Norman Hendrich

HADES Tutorial

Contact:

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

hendrich@informatik.uni-hamburg.de

Contents

List of Figures

1   Introduction

This section introduces the concepts behind Hades, the "Hamburg Design System", a portable Java-based visual simulation environment. While Hades can be used for any type of discrete-event based simulation, the main focus is currently on the simulation of digital logic systems on gate-level up to system level, including efficient hardware-software cosimulation.

visual simulation

The remainer of this introduction is organized as follows. For the impatient reader, section 1.1 sketches the contents of the later chapters of this tutorial. It also recommends the order of reading for first-time users, experienced users, and developers.

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 1.4. Subsection 1.3 contains a summary of the functionality of Hades, together with a list of the features planned for the near future.

1.1   How to read this Tutorial

For a quick start, browse through the installation instructions in chapter 3 and then read section 4, "Hades in a Nutshell". It presents the user interface and step by step explanations of the basic editor commands and the *std_logic* system for digital logic simulation. Should you encounter trouble, browse through the FAQ in chapter 10 and use the index to locate more detailed information.

getting started

Chapter 5 builds upon chapter 4 and includes 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 the printing and export options. Chapter 6 explains the Hades waveform viewer, chapter 7 lists some of the available simulation models, and chapter 8 demonstrates scripting.

advanced users

Chapter 9 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 is shown to demonstrate the interaction between simulation components and the simulator. The third model, a clock generator, is used to explain access to the simulator, graphics, mouse input, I/O, and configuration.

developers

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]—includes an interactive simulation mode, where circuit inputs can be toggled via mouse-clicks or the keyboard in real-time. This allows to set the input values while the simulation is running, without the need to write an external stimulus file, and without lengthy edit-compile-simulate-analyze cycles. 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 fully supports hierarchical designs, and all functions can be scripted, so that experienced users can create and simulate complex systems without the need for (possible expensive) design software.

Unlike some other systems, where simulation models have to be written in some specialized internal programming language, all Hades simulation models are directly written in Java. This gives the designer full access to a modern object-oriented programming language with a rich class library, including full network access and portable graphics. Despite the lack of special syntax, simulation models written in Java are not necessarily more verbose than models written in simulation languages like VHDL [IEEE-93a]. See chapter 9 about how to write your own Hades simulation models.

The combination of a powerful language, a visual editing environment with animation support, and full access to the simulation kernel makes for new design and debug possibilities.
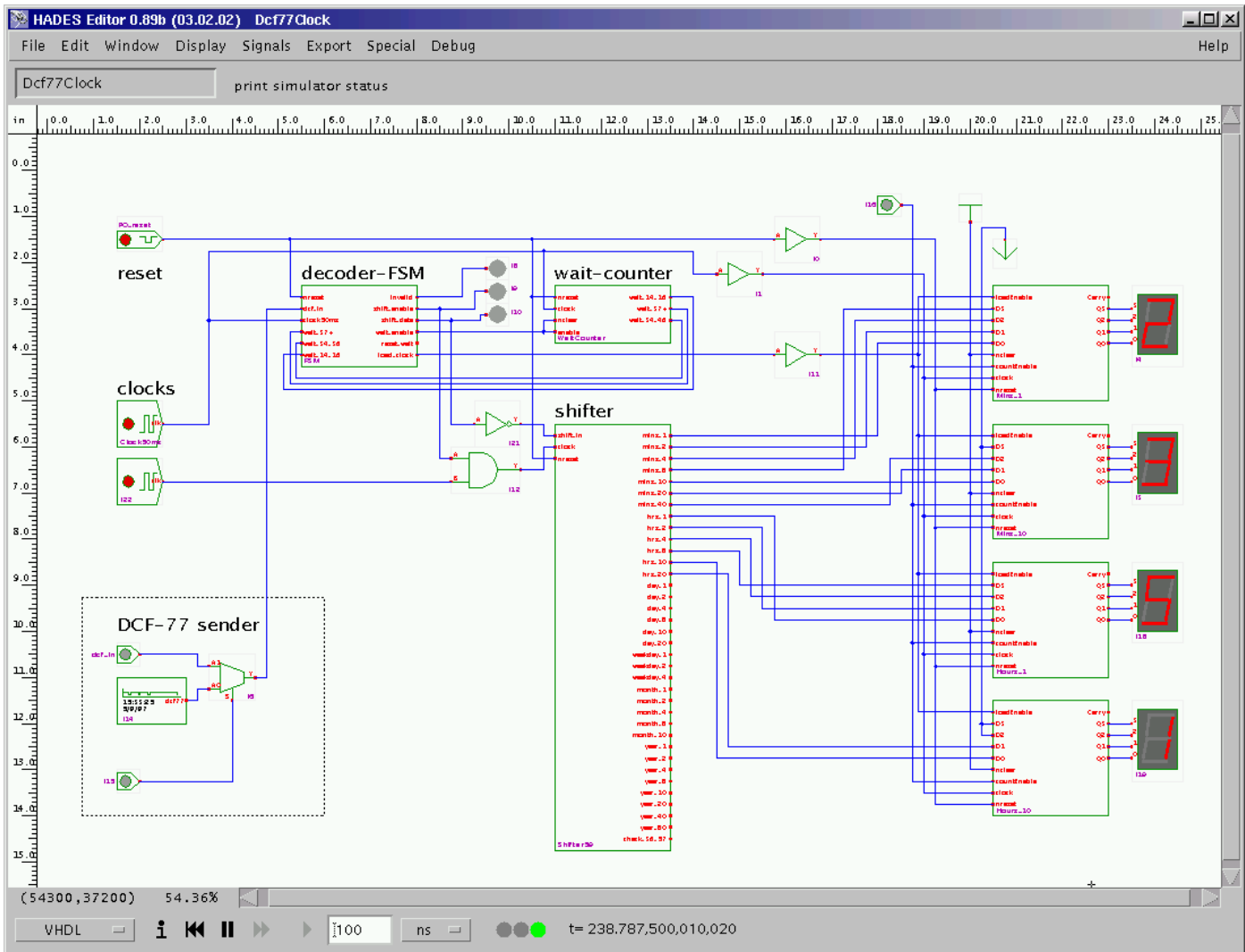
Figure 1: A Hades example design: The toplevel schematic of a DCF-77 radio controlled digital clock, showing basic gates, interactive switches, animated seven segment displays, subdesigns, and behavioral components including the DCF-77 sender.

For example, writing complete and interactive system-level testbenches for digital system simulation can be much easier than with other design systems. Additionally, the Hades framework offers full access to its simulation kernel, which can also be replaced by user-defined simulation algorithms. For example, the *hades.models.pic.SmartPic16C84* microprocessor core uses this feature to efficiently synchronize its internal period-oriented operation with external logic running in the discrete-event based simulation kernel. Compared to the traditional way of running the microprocesor core under control of the discrete-event based simulation kernel, speedups of a factor of 5 have been observed using this feature.

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. For example, the *hades.models.imaging* includes a few image-processing operators written as Hades simulation components.
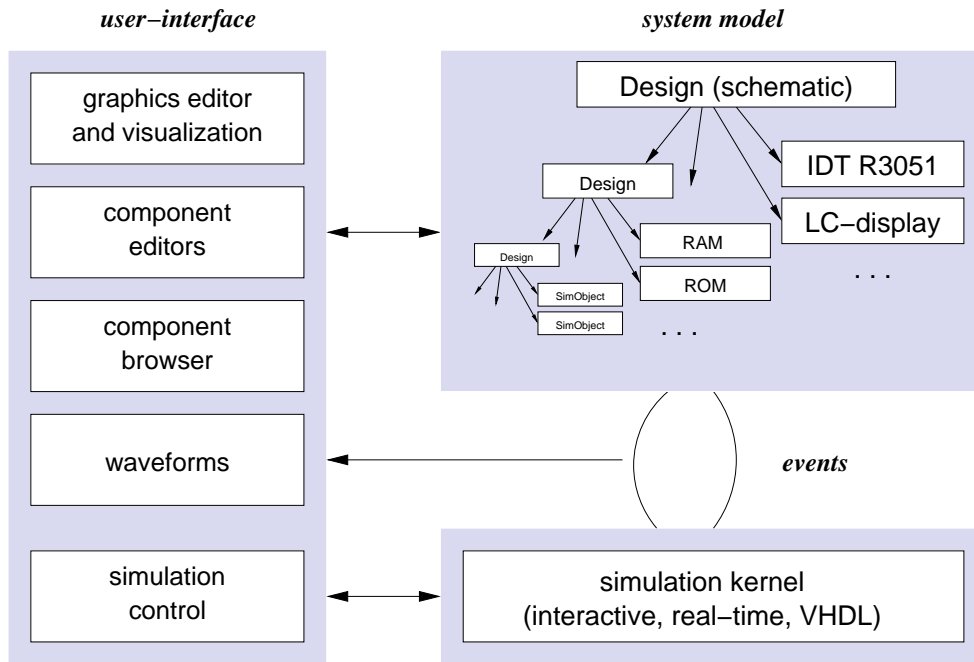
Figure 2: The software architecture of the Hades framework. A *design* represents a hier-archical collection of simulation models, which interact under control of the event-driven *simulation kernel*. The user interface consists of the main graphical *editor*, the component and library *browser*, simulation component *property dialogs* and editors, a *waveform viewer*, and the *simulation control*.

1.3   What is Hades?

The major components of the Hades framework are shown in figure 2. It consists of the simulation engine, the simulation experiment built from a set of pre-defined simulation components, the graphical editor and several other user interface blocks. With the current version of Hades, you can:

- design and simulate digital electronic circuits,
- on gate level, RT level, and system level,
- based on the *std_logic* and *std_logic_vector* multi-level logic system, including buses with resolution functions,
- 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 jfig 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 0.92 of Hades, the following models are available:

- basic gates and complex-gates with up to six inputs,
- all standard flipflops,
- a variety of I/O models, including interactive switches and animated components like LEDs or seven-segment displays,
- behavioral models for many 74xx TTL circuits,
- a large library of RTL components (based on std_logic_vector),
- memories (RAM, ROM, microcode) with graphical memory editors,
- LC-display models, a text terminal (VT52/VT100), UART,
- interfaces to our state-machine editor applet,
- PIC16Cxxx and MIPS R3000 family microcontroller cores,
- LFSR- and BILBO-registers for circuit selftest and signature analysis.

At the moment, all models use a simplified generic timing as might be typical for a $1.0\mu m$ CMOS library. If necessary, subclasses with detailed timing data could be generated to model a specific ASIC library accurately. See chapter 9 on how to write your own Hades simulation models.

1.4   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.

2 Showcase and demos

This chapter presents a few Hades design examples as an overview of the editor and simulator features. The screenshots used in the following pages were taken on both Windows and Linux systems, with several different windows managers and versions of Java, to underline the portability of the Hades software.

The first example is a simple gate-level circuit, which illustrates interactive simulation and glow-mode. The carry lookahead adder from [Hennessy & Patterson] shows how to use design hierarchy and custom symbols. The next examples demonstrate the interactive state-machine editor and a typical 74xx series circuit.

gate-level

Three examples are then used to explain the register-transfer level simulation models available in Hades. A simple 16-bit counter and a user-defined ALU show the individual components, while the third example presents a complete 16-bit microprocessor with microprogrammed control, multiport register bank, and external memories.

register-transfer level

Next, a MIDI controller based on a PIC16C84 microcontroller demonstrates hardware/software-cosimulation. Due to the small number of pins on the microcontroller, the keypad and display are multiplexed, which poses interesting problems for visualization.

cosimulation

Finally, an asynchronous micropipeline circuit is presented. The complex handshaking used in the micropipeline is an excellent demonstration of the use of interactive simulation. Under the hood, the simulation models for the C-gates and the registers were written in the Jython scripting language.

scripting

Further design examples are used and explained in the later chapters. Archive files with most of the examples presented here are available for download at the Hades homepage. See section 3 for details on downloading.

The chapter closes with a short overview of the interactive applet collection on our Hades website, which includes over 300 ready-to-run circuits ranging from basic gates and flipflops to full system-level simulation of 32-bit MIPS-based systems.

online applets

2.1   Hamming Code
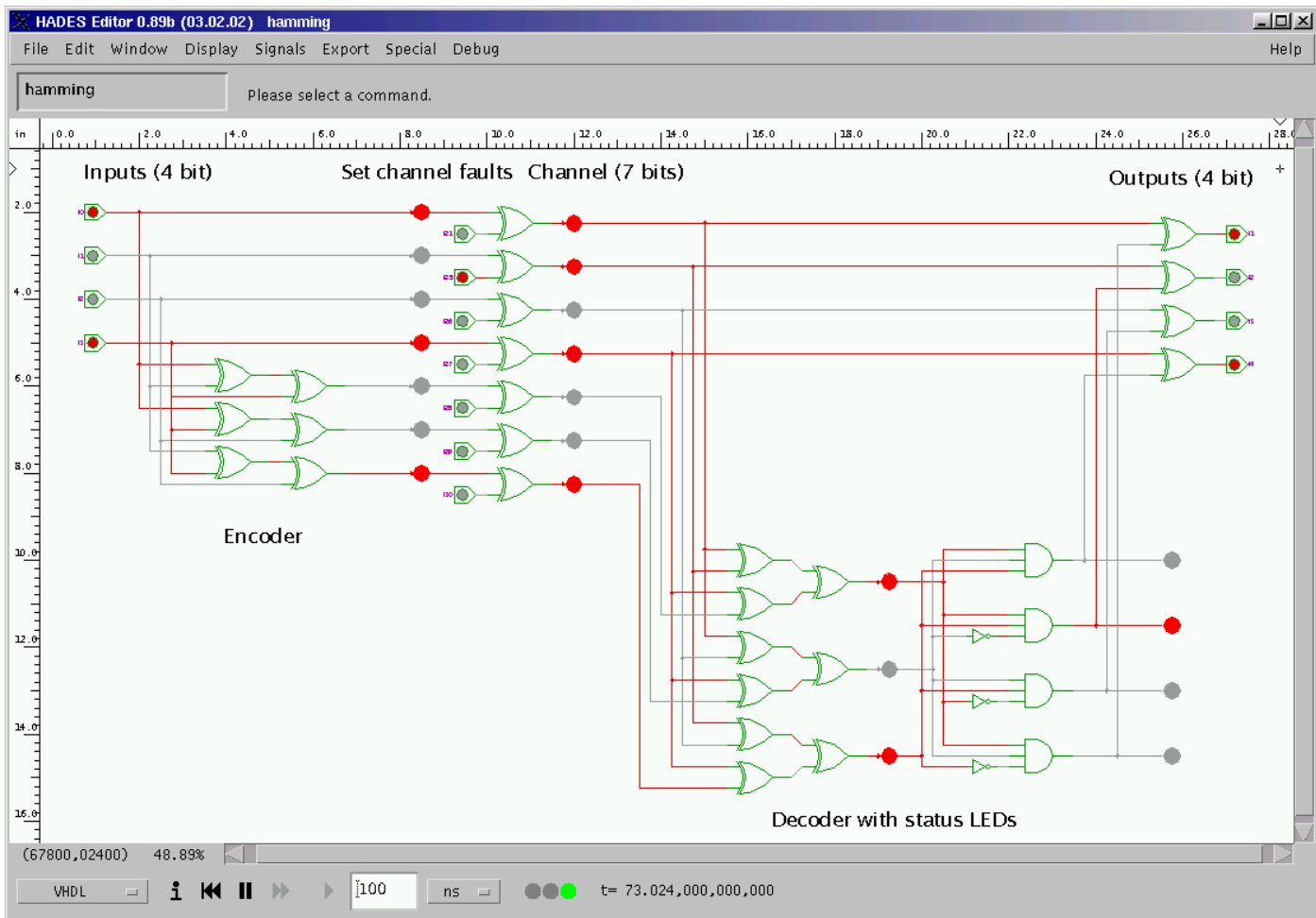
Figure 3: Four-bit Hamming code demonstration. The circuit consists of three parts. On the left are the input switches and the XOR-gates for the Hamming code encoder. On the right is the decoder with the output and status LEDs. The XOR-gates and switches in the middle of the screen allow simulating channel faults on each of the seven transmitted signals.

gate-level circuits   This simple gate-level circuit realizes a 4-bit Hamming-code encoder and decoder. It demonstrates the use of interactive simulation and the visualization with glow-mode.

interactive simulation   Use the four switches on the left to set the input value for the encoder, and observe the decoded value on the four output LEDs. A simple click on the switch will toggle the corresponding signal between the 0 and 1 values. In glow-mode, the logical value of a signal is *glow-mode*   indicated by the signal's color, which provides immediate visual feedback about the circuit behaviour. Different colors are used for the nine values of the *std_logic* multi-level logic system used by Hades for gate level simulation. For example, cyan color indicates the X value, allowing to detect un-initialized parts of the circuit at one glance.

The switches and XOR-gates in the middle of the screen allow to invert the signal transmitted to the decoder, simulating a channel fault on each of the seven transmission signals. Note that the circuit will correct any single channel-fault, but is unable to detect or correct all double (or higher) faults.

real-time mode   Another feature of the simulator is the real-time simulation mode which intentionally slows the simulation down. This can be used to demonstrate gate-delays and hazards.

2.2 Carry-Lookahead Adder
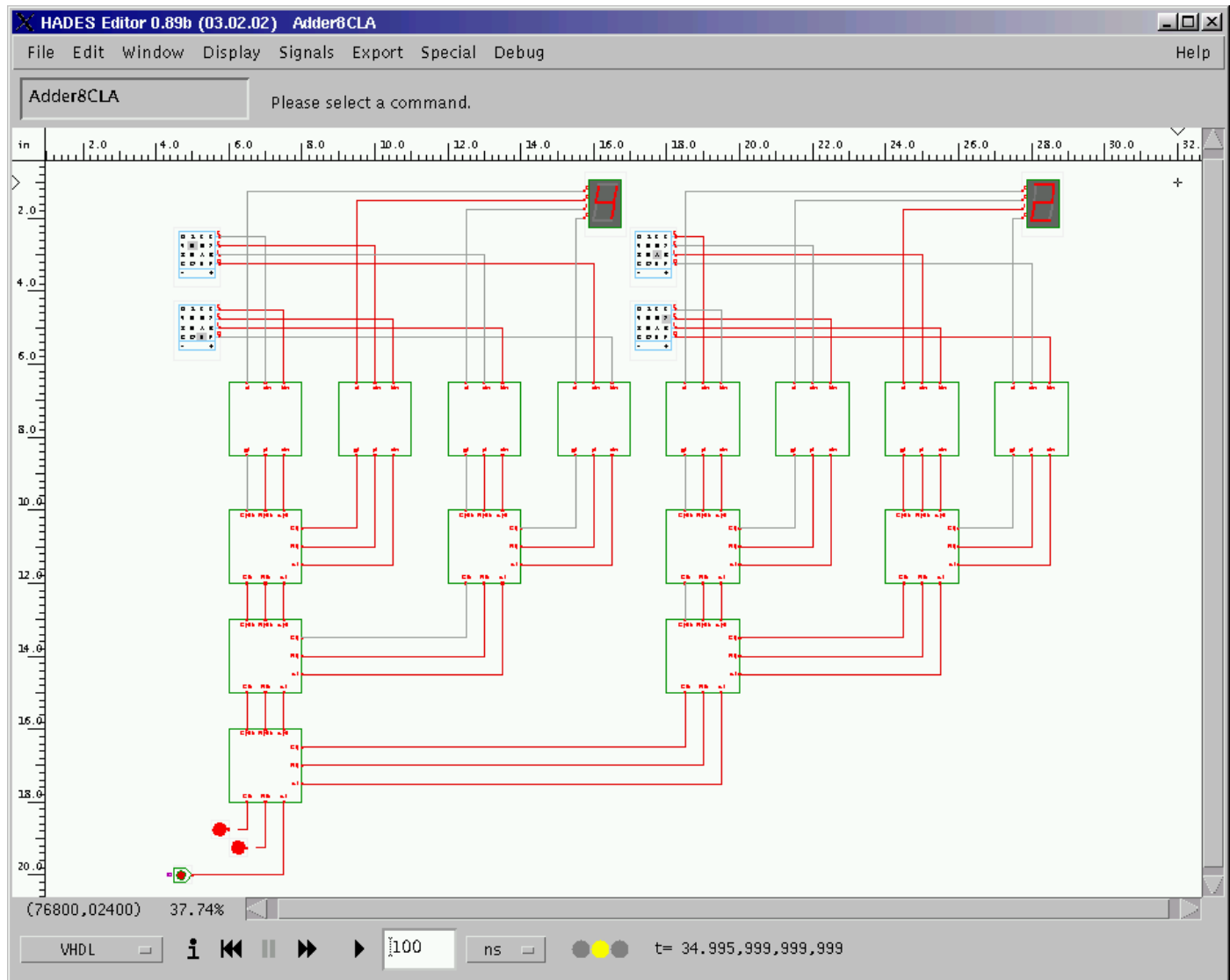
Figure 4: Eight-bit carry lookahead adder circuit.

This eight-bit carry lookahead circuit is the interactive version of the carry lookahead adder presented in [Hennessy & Patterson]. The example illustrates a design hiearchy, consisting of eight instances of the `sum.hds` and seven instances of the `cla.hds` sub-designs. *hierarchical designs*

A bottom-up design style was used to build the circuit, as the 1-bit adder and 1-bit carry-generate blocks where designed first. The graphical symbols, required to integrate the sub-designs blocks into the top-level schematics, can be created automatically. However, custom graphical symbols are used in the example, to help with algorithm visualization. The steps required to design hierarchical circuits in Hades are presented in section 5.1 on page 43. *bottom-up design*

The CLA circuit also includes animated hex-switches and -displays for input and output. Use the four HexSwitches to select the summand values a<7.4>, a<3:0> and b<7:4>, b<3:0>. Then click the `carry_in` switch to set the input value for the adder, and observe the output value on the seven-segment displays and the `carry_out` and `carry_propagate` LEDs. *switches and displays*

2.3   Traffic-light controller
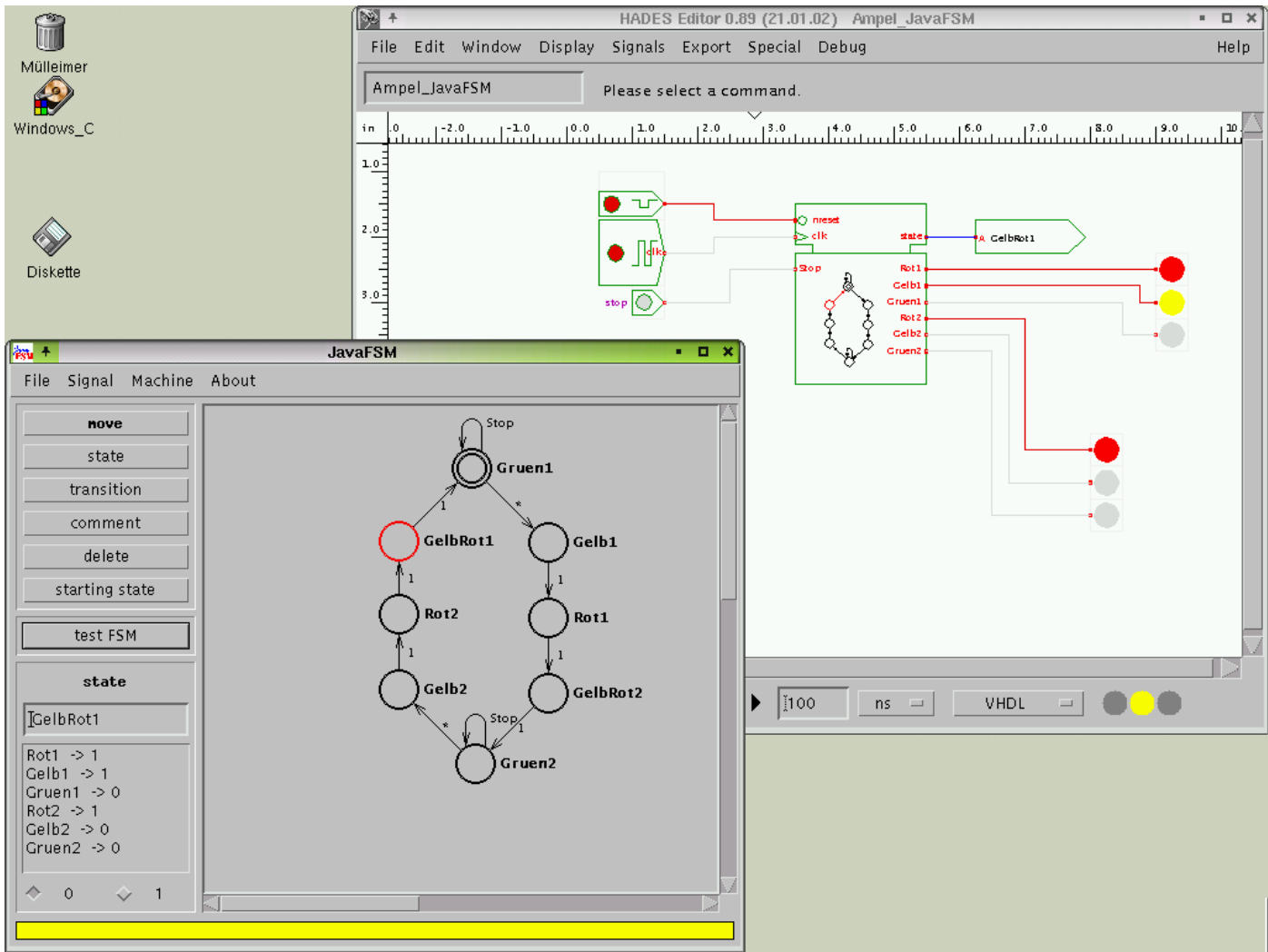
Figure 5: Traffic-light controller, realized using the Hades interactive state-machine editor.

state-machine editor   This example shows a simple traffic-light controller, realized as an instance of the Hades state-machine editor. This interactive editor is also available standalone as an applet on `tech-www` `.informatik.uni-hamburg.de /applets/java-fsm/`.

The GUI of the editor is available after selection of the state-machine type (Mealy or Moore) and the definition of the inputs and outputs of the machine. It is shown in the left part of the above screenshot and allows to create, edit, and delete states and transitions. Note that the graphical symbol for the state-machine used in Hades highlights both the active state and the active transitions of the FSM.

clock generator   The example circuit also includes the `ClockGen` and `PowerOnReset` components, as well as colored `LEDs`.

2.4 TTL-series 74xx components
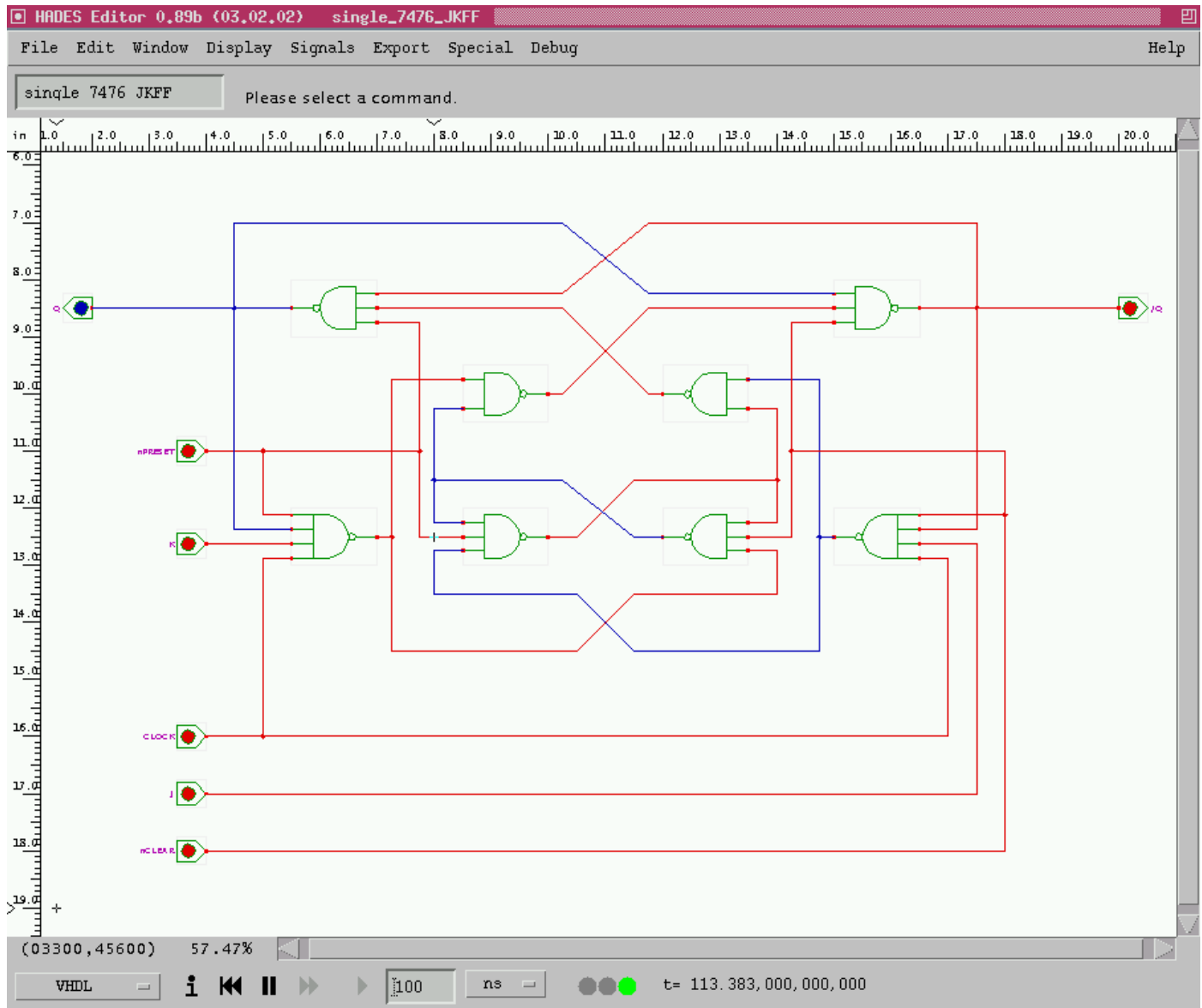
Figure 6: One example of the 74xx series components available in Hades: The internals of the 7476 JK flipflop.

The above screenshot shows the internal structure of (one half of) the 7476 dual-JK flipflop. Many of the TTL 74-series components are available as Hades simulation models, some of which are realized as standard sub-designs, while others are written as behavioural models.

TTL 74xx series models

This also allows to design and simulate many legacy circuits, which are often based on the 74xx series. The only complication is that some circuits don't use explicit reset logic and rely on random initialization of the real electronics. Such circuits require either additional reset logic in Hades, or the use of the *metastable* flipflops.

legacy circuits

Other system-level simulation models in Hades include RAM and ROM memories, a VT52-compatible terminal (with serial or parallel interface), standard text and graphics LCD displays, and measurement equipment like counters or hazard-detectors.

system-level models

2.5   RTLIB 16-bit counter
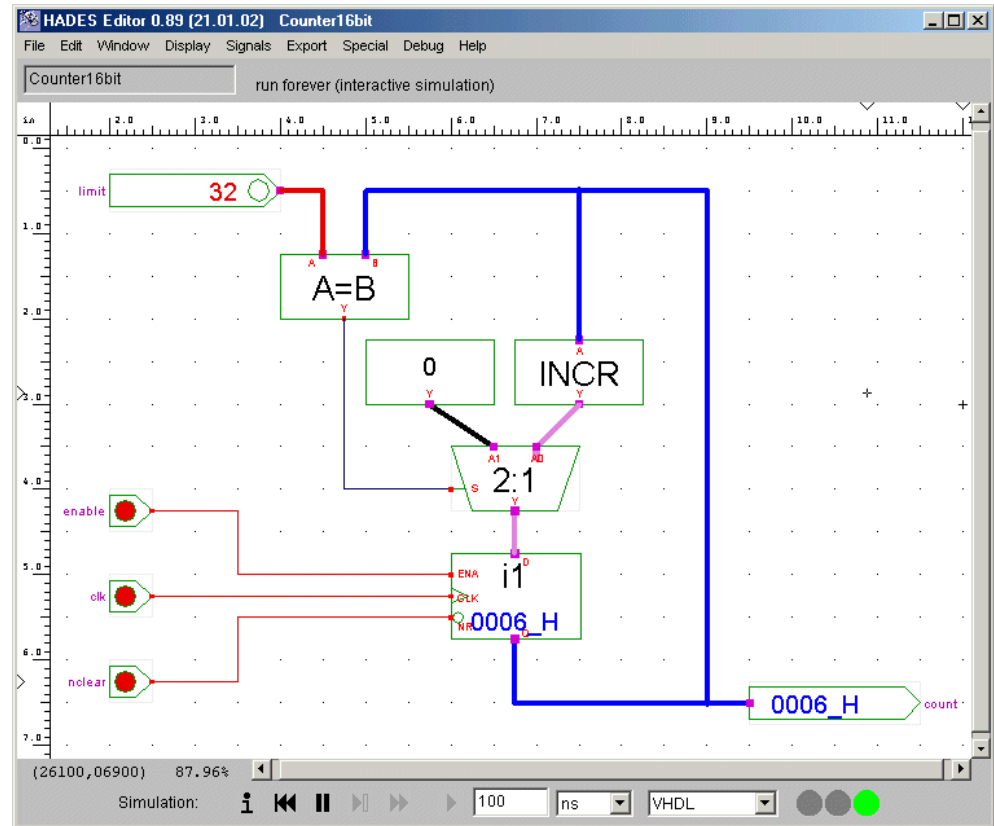
Figure 7: A simple counter circuit built with the RTLIB simulation models.

register-transfer level   Besides the standard gate-level simulation models like basic gates and flipflops, Hades also includes a variety of simulation models for register-transfer level operations. The screenshot above shows a simple 16-bit synchronous counter with selectable maximum value, built from a register, an incrementer, a multiplexer, and a comparator.

The input switches used in the example allow incrementing or decrementing their current value via mouse-clicks. Also, the input values can be set directly via the switch property sheets. All RTLIB components support bus-widths in the range of 1 to 63 bits.

RTLIB glow-mode   A special feature is the glow-mode for the bus signals. As with glow-mode for single-bit signals, the color is used to encode and visualize the value on the signal. Naturally, using 65536 different colors for a 16-bit signal is not a realistic options. Instead, the encoding scheme selects a color based on the the last decimal digit of the integer value of the signal. The colors are chosen as the DIN/IEC code used for resistor marking. For example, integer values 2, 12, 22, ... will be encoded with red color, 3, 13, 23, ... with orange, etc. This very simple algorithm works surprisingly well and also allows to follow datatransfers in quite complex circuits (see below).

2.6   RTLIB user-defined ALU
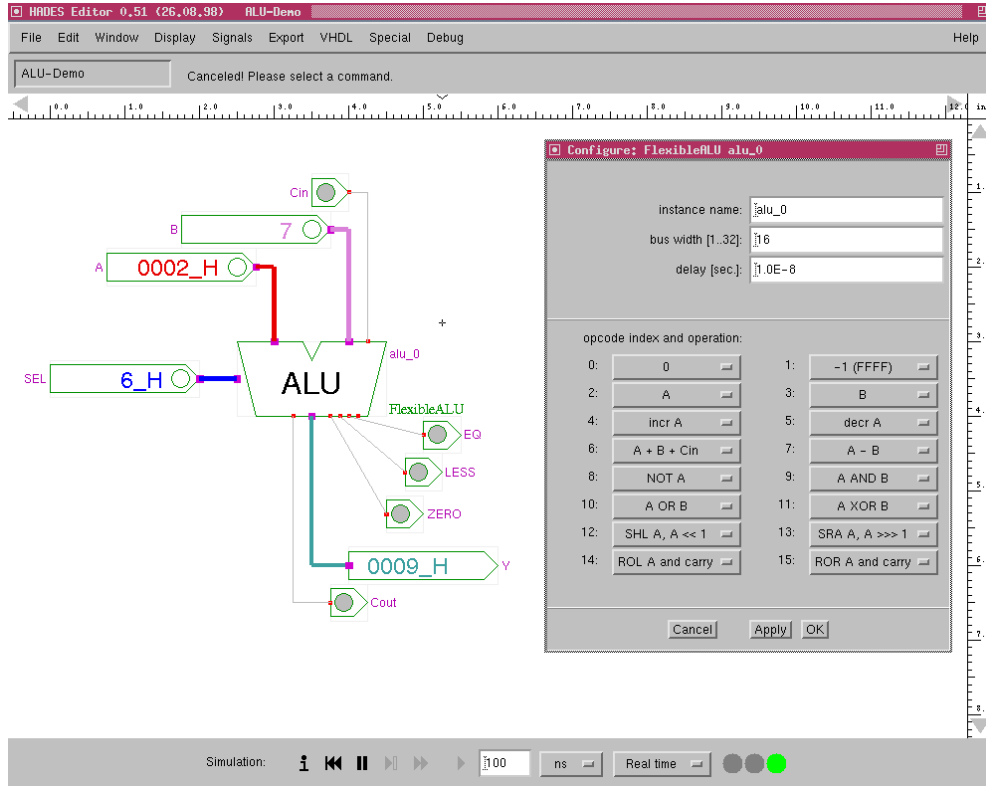
Figure 8: Interactive demonstration of an user-defined arithmetic-logical-unit. All functions of the ALU can be changed interactively via the ALU property sheet.

Simple demonstration of a multi-function arithmetic-logical unit. The interactive switches are used to generate input values for the data inputs `A`, `B`, `Cin` and the control input `SEL`. The ALU model supports a 4-bit control input to select 16 different operations out of a set of about 30 possible operations.

ALU operations

The screenshot also shows the property editor of the ALU simulation model. It allows to specify the component name and the global parameters bit-width and propagation delay. The rest of the GUI is used to select the mapping of ALU function codes (opcodes) to the actual operations. For example, the opcode 9 selects the `A AND B`.

property sheet

A similar but more complex example is the datapath circuit used in our T3 lab-course which combines the user-defined ALU with a multiport register-file.

2.7   D*CORE processor
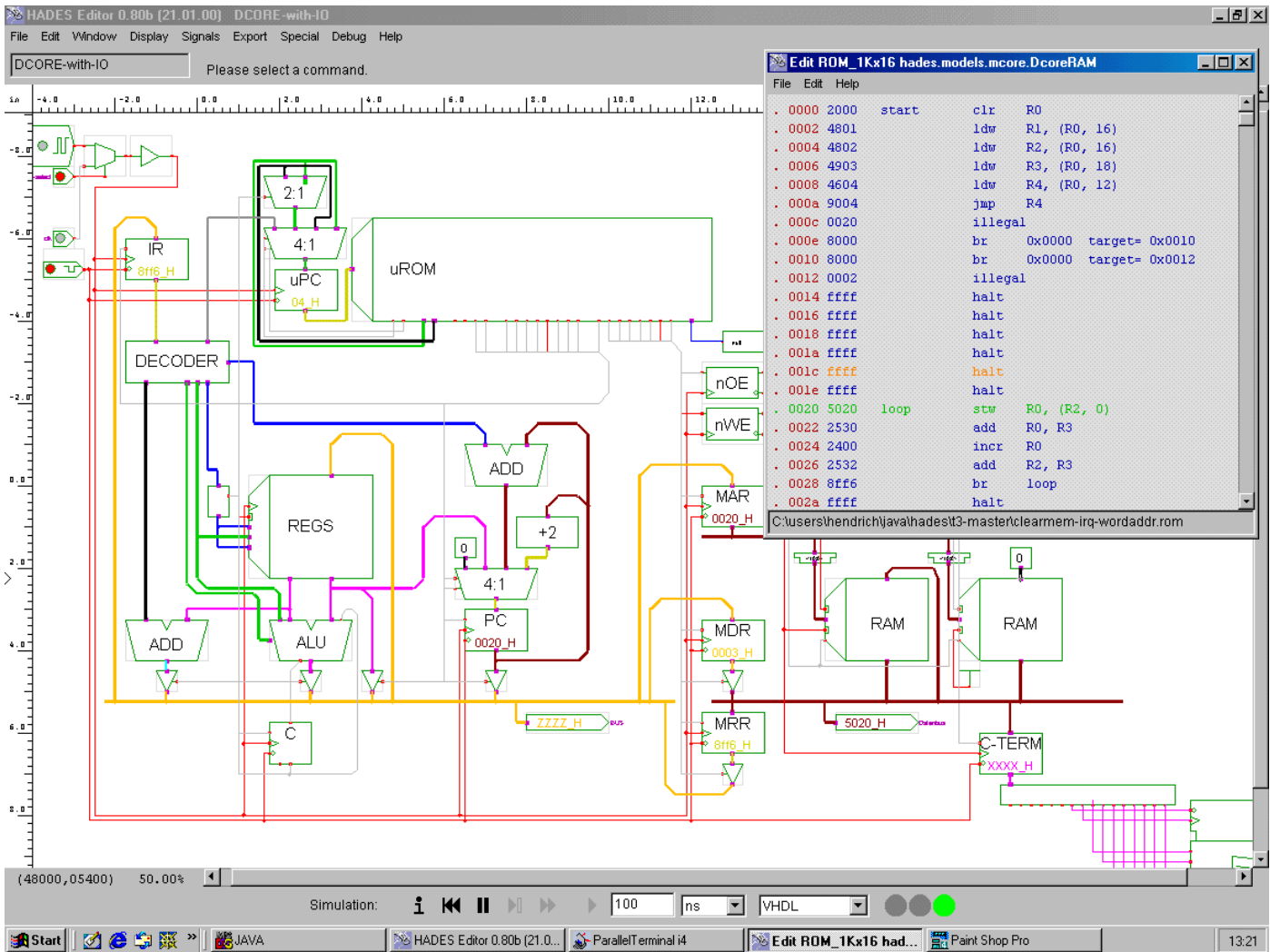
Figure 9: A 16-bit microprocessor used for teaching. This circuit features a complete 16-bit microcprocessor with microprogrammed control unit, three-port register file, multi-function ALU, bus-interface, address-decoder, RAM, ROM, and I/O components.

RT-level   This example demonstrates a complex register-transfer level design, built from dozens of RTLIb components. It implements a complete 16-bit microprocessor with datapath and microprogrammed control, as well as the external memories.

microcode   The processor is currently used in our third-semester lab-course on computer architecture and operating system principles. After a set of initial exercises which explain the individual parts of the processor (e.g. the datapath alone), the students are given the complete hardware structure shown in the above figure. However, the microcode is initally blank and must be *visualization*   written by the students. Unlike text-based processor simulators, Hades allows us visualize all datatransfers required for the instruction cycle, and the integrated editor for the microprogram memory allows to modify the microcode without recompilation or restarting the simulator.

The students then proceed to write some assembly programs for their processor, thereby learning about all levels of machine and program architecture. The memory editor in the top right corner of the screenshot shows includes the option to disassemble the memory contents, and it highlights current read and write accesses.

overclocking   Also, the circuit includes the bus-interface registers and models typical gate-delays and memory access times, which can even be used for an overclocking experiment.

2.8   MIDI-controller using a PIC16C84 microcontroller
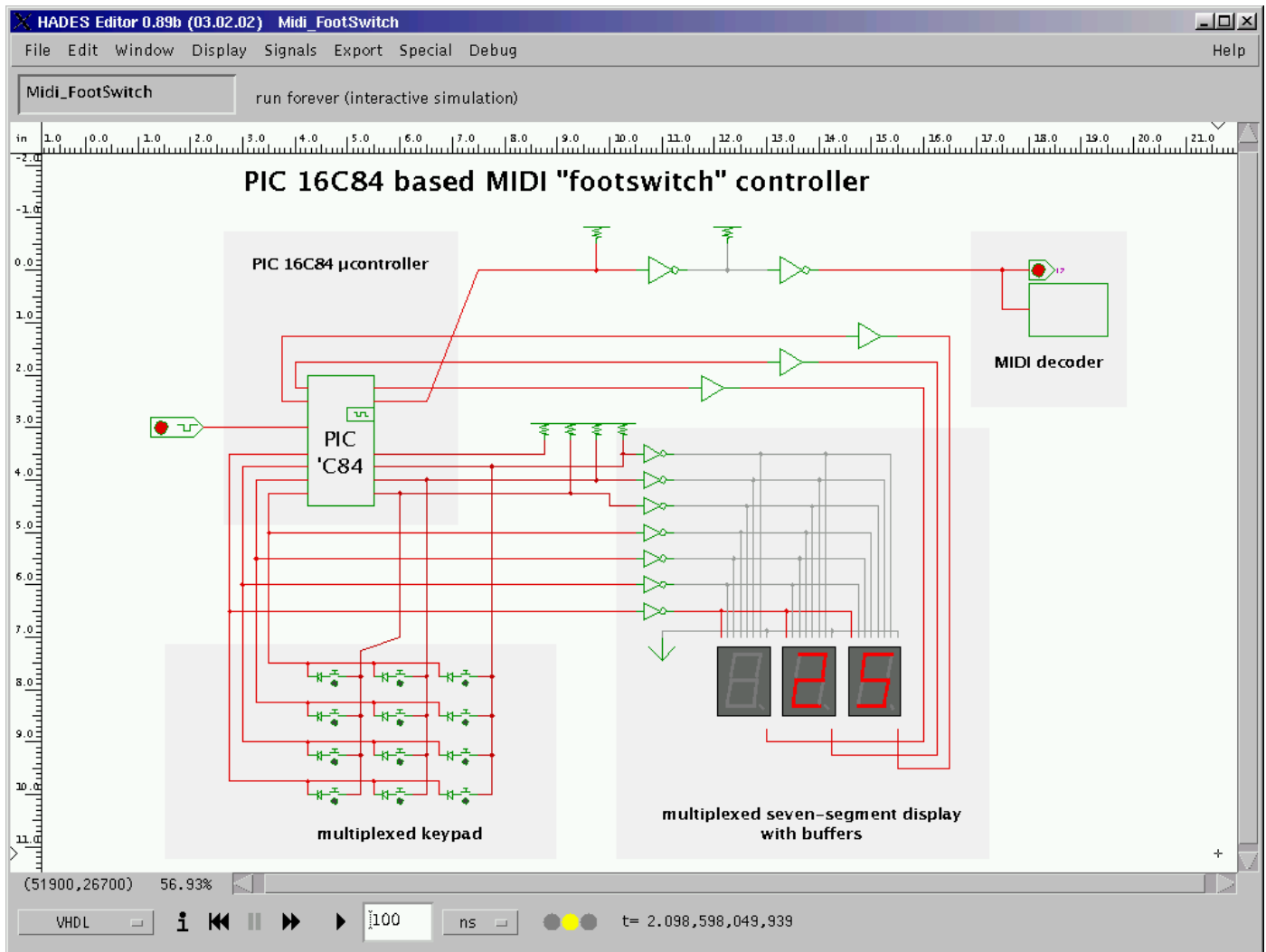
Figure 10: A MIDI-controller circuit based on the PIC 16C84 microcontroller core with multiplexed bus, interactive switches and displays, and a software-controlled RS232 (32.125 KHz) serial communication to the MIDI port.

Based on a PIC16C84 microcontroller, this MIDI (musical instruments digital interface) „footswitch" controller demonstrates cosimulation in Hades. The software running on the microcontroller periodically samples the keypad including debouncing, displays the selected program number, and generates the corresponding MIDI control messages. The serial communication at 32.125Kbaud is software controlled. The PIC16C84 controller is attractive for low-cost prototyping because it is EEPROM based and in-circuit programmable. Also, very cheap programmers are available.

cosimulation

To improve simulation performance, several variants of the PIC16C84 core model are provided. The slowest models uses the external clock input and provides fully accurate timing for all instructions and interrupts. The faster models use a cycle-based implementation to run the processor core independently of the event-driven simulation kernel for the I/O signals. The necessary synchronization is possible using direct method calls to the simulation kernel.

cycle-based simulation

The design also demonstrates the use of the `std_logic` logic system to model the buses with weak pullups and multiplexed open-drain switches. A special simulation model is used for the seven-segment displays, in order to integrate the multiplexed input values for continuous display.

multiplexed displays

2.9   Micropipeline
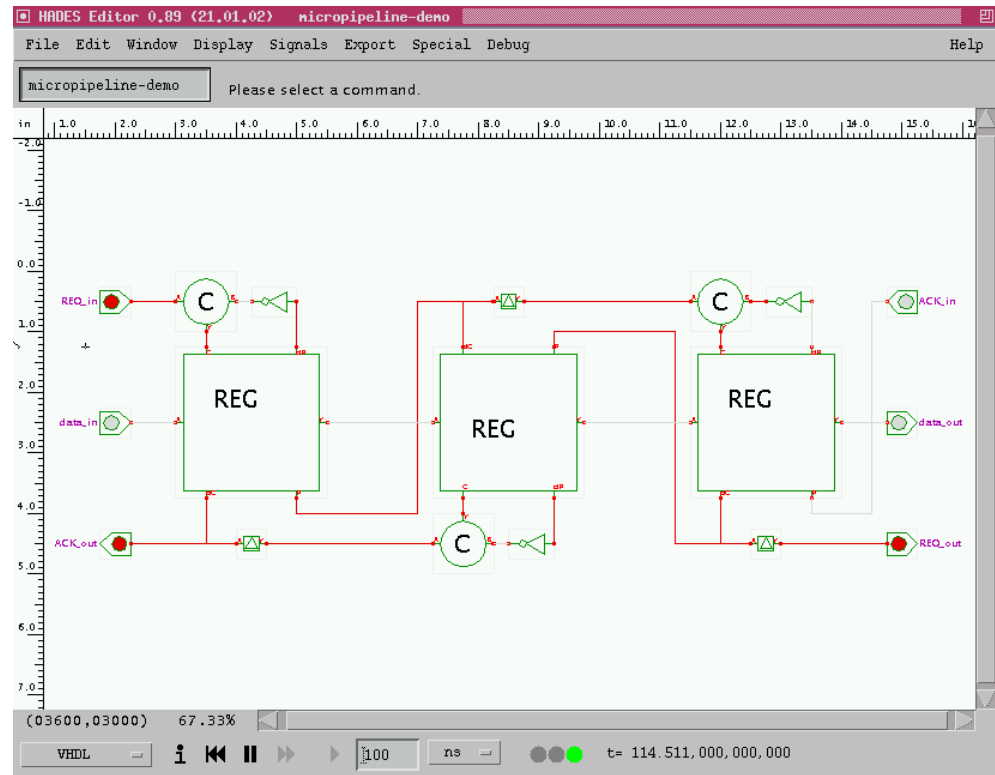
Figure 11: An asynchronous micropipeline built from Muller C-gates. The simulation models in this example were written in the Jython scripting language, illustrating the seamless integration of external simulation models into Hades.

asynchronous   This circuit demonstrates part of an asynchronous micropipeline and allows to play with the complex handshaking protocol between the pipeline stages. It also shows the initialization problems inherent in asynchronous system design. Using the default simulation models, with their undefined (`std_logic 'U'`) initial state, the circuit can only be made to work by explicit initialization via the command shell or a script.

Jython   While this can not been seen on the above screenshot, the simulation models for the micropipeline were written in the Jython scripting language. A similar approach is possible with all programming languages that provide a binding to the Java object model. The advan-
rapid prototyping   tage of a scripting language like Jython is the rapid prototyping style of development. Unlike Java-based simulation models, which usually cannot be changed during a simulation run, the scripts can be re-read and modified even at simulator runtime.

2.10 Switch-level Simulation
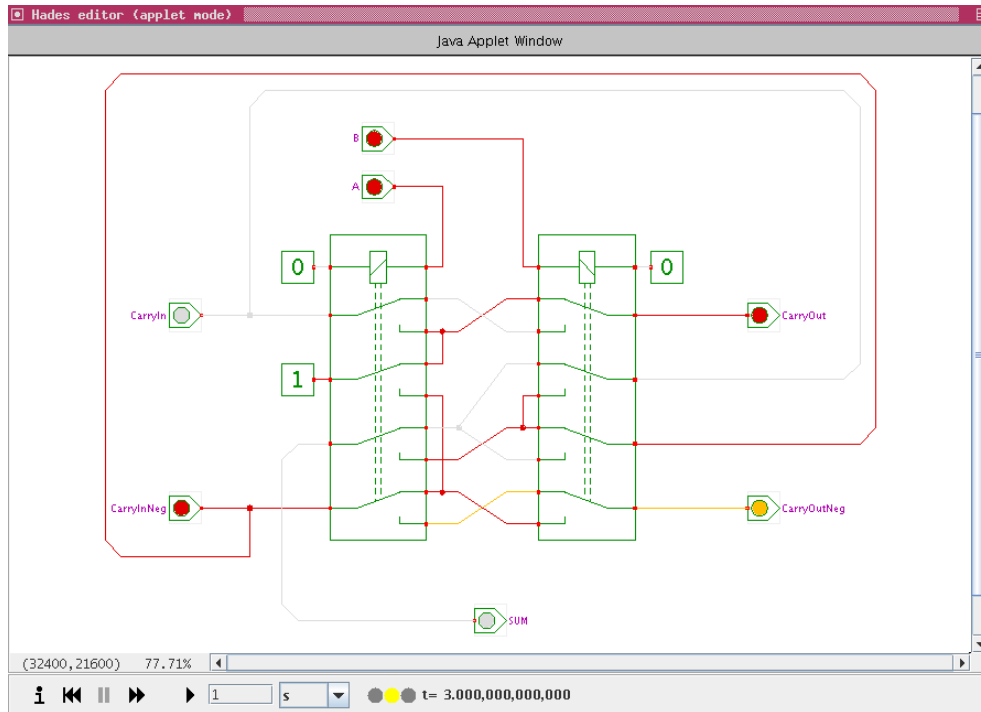
Figure 12: One stage of the adder circuit used by Konrad Zuse in his Z2 and Z3 relais machines. The dual-rail encoding and wiring ensures that the external carry-input signal propagates instantly through all stages of the adder. Therefore, the total delay of a multi-stage adder is just one cycle.

Since version 0.98, the Hades framework also integrates support for *switch-level simulation*. Based on an improved simulation algorithm, the new switch-level components (including switches, relais, and transmission gates) can be mixed freely with all existing simulation components. Also, switch-level components can use the whole std_logic_1164 logic model and arbitrary gate-delays. No separate simulator or a complex coupled-simulator setup is required. Instead, the editor ensures that switch-level components are connected via special signals which manage the extra bookkeeping when switches open or close. *switch-level simulation*

The screenshot above shows a one-bit adder built from two relais. The circuit is based on the clever scheme used by Konrad Zuse in his historical Z2 and Z3 machines. Based on a dual-rail encoding (both positive and negated polarity) of the carry signal and clever connection to the relais contacts, multiple stages of the adder can be cascaded without extra logic. Because the propagation delay through closed contacts is much faster than the switching time of a relay, the total delay of a multi-stage adder is practically the same as for the one-bit adder. *Zuse adder*

The screenshot also shows the *view-mode* variant of the user-interface, where the main menu-bar and most edit-controls are hidden from the user. This mode is useful when students are expected to study and analyse the behaviour of given circuits instead of building their own circuits. *view-mode*

2.11   Applet Website
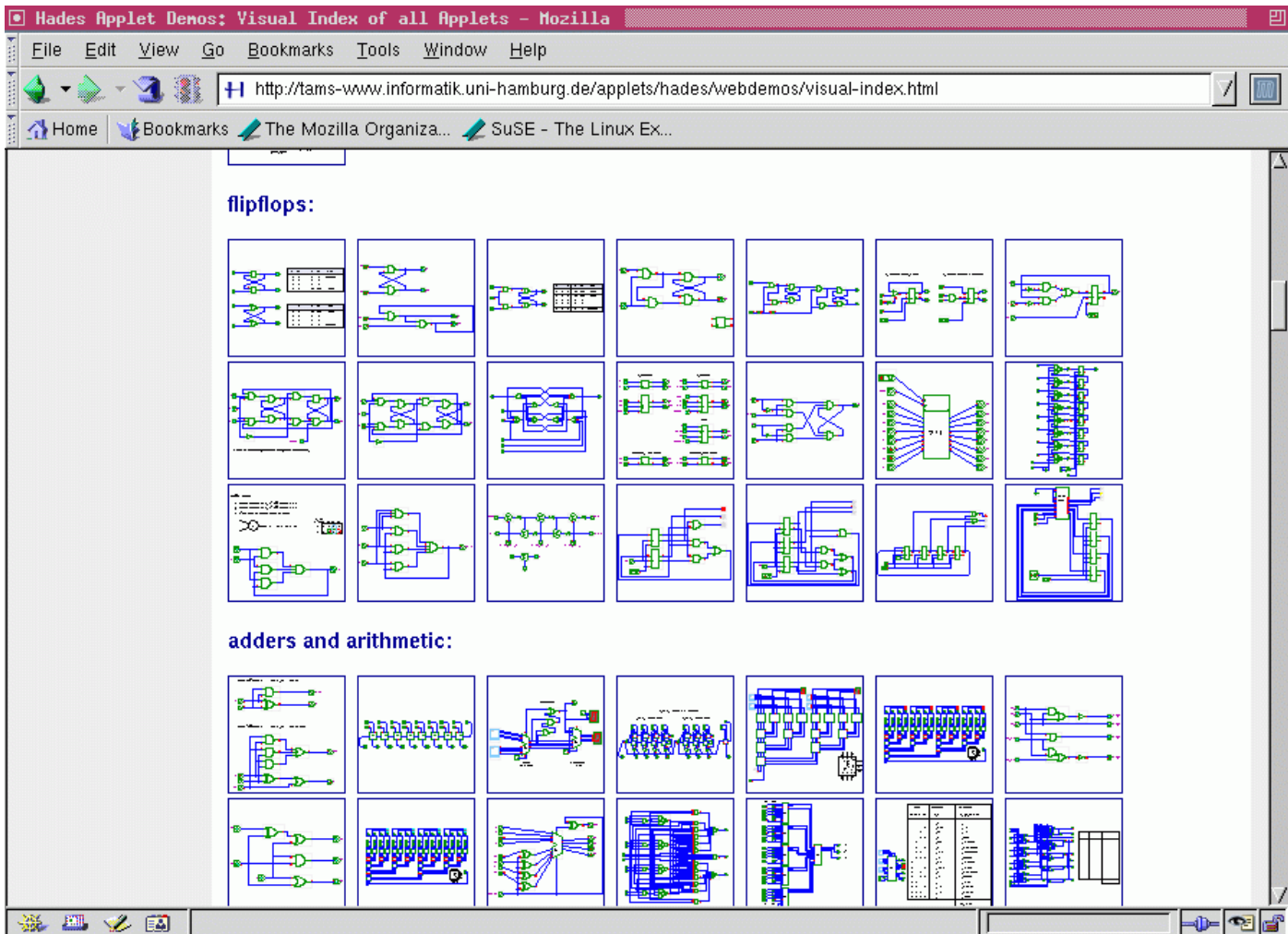
Figure 13: Screenshot of the Hades applet website index page. The applet collection on the Hades homepage currently includes well about 300 different circuits, each ready to run and with its own documentation.

applet version Since the widespread availability of high-speed internet access via DSL and similar techniques, it finally proved practical (during 2005) to allow users running the Hades simulator as applet or via the Java webstart protocol. Our website currently hosts a collection of about 300 individual interactive demonstration applets, each embedded into a webpage with detailed explanation and cross-references to other applets and external web content. The applets can be accessed with every Java-enabled web-browser and should run without installation.

To keep the downloading times as short as possible, the applet version of Hades consists of a core software archive of about 2 Megabytes only. This includes editor, simulator, and a set of basic simulation components. The remaining simulation models and utility classes are packaged into separate archive files which are downloaded on demand, when the user visits the corresponding webpages.

In addition to the webpage with embedded applet, we also offer a separate *Java webstart* link for each demonstration circuit. This allows the user to automatically download, install, and run the full Hades editor without the manual installation (described in detail in chapter 3).

visual index The screenshot above shows part of the *visual-index* page of our website collection, where thumbnail images allow the user to quickly find and access particular applets. Clicking a thumbnail opens the corresponding web-page.

3   Installation

This section presents all steps required to get the Hades editor and simulator to work. It first discusses the system requirements and the selection of a suitable Java virtual machine. Sections 3.4 and 3.5 describe the software download and the setup of a working directory for your designs. Two examples show how to register the Hades .hds files as a new file type on Windows systems. Finally, section 3.11 and 3.10 describe multi-user setup and the steps involved to script Hades from Jython.

installation overview

To get Hades running on your system, follow these steps:

1. check that your computer meets the system requirements.
2. verify that you have a suitable Java virtual machine, namely JDK/JRE 1.4.2 or higher.
3. visit the Hades homepage and download the software, documentation, and examples.
4. create a suitable working directory for the design examples and your own designs.
5. optionally, create script files to start the Hades editor with your favorite options. You may also want to register .hds design files with your operating system.
6. use the software. See the next chapter 4 for a walk-through on a simple design example.
7. if you experience problems, see the FAQ (section 10) for help.

The Hades editor can also be used as an applet in any Java compatible web browser without extra installation. However, due to the tight security checks for applets, several functions of the editor are disabled unless you explicitly override those security settings. The details of changing the security settings for applets are discussed in section 3.12.

3.1   Quick Start

If a suitable Java virtual machine (JDK/JRE 1.4.2 or higher) is already installed on your system, you can download and start Hades very easily. Unless you want to modify the simulator and write and debug your own simulation models, it is probably best to first try the *Java Webstart* installer. If Java Webstart is setup and enabled in your browser:

Webstart installation

- Visit *http://tams-www.informatik.uni-hamburg.de/applets/hades/*
- Follow the link to the *webstart* page,
- Click the *download and run Hades* button, and wait until the download has completed.
- When prompted by the Webstart installer, agree to run the editor. The installer might also offer to create desktop and start-menu entries for Hades.

Otherwise, download the hades.jar software archive manually:

manual download

- Visit *http://tams-www.informatik.uni-hamburg.de/applets/hades/*
- Follow the link to the *download* page,
- Use your browser to download the hades.jar archive file (right click, then select *save link as* or similar). Remember where you save the file.
- Open a command shell, change to the directory that contains the hades.jar file and run the editor via the command

      java -jar hades.jar

  Double-clicking the hades.jar file might also start the editor (depending on your desktop settings).

3.2   System requirements

Hades is written in the Java programming language [Sun 97] and uses features first introduced
with JDK/JRE 1.4. It should run without modification on every computer system that offers
a compatible Java virtual machine (JVM). This includes PCs or workstations running either
Windows XP, Windows 95/98/ME, Linux, most commercial versions of Unix including So-
laris or AIX, and the Apple Mac OS X. In principle, Hades should also work on systems like
fast PDAs and organizers if a JVM is available for that system.

screen size   Because of the user interface with the graphical editor, a large and high-resolution monitor is
recommmended for running Hades. A display of 1024x768 pixels should be ok, but the higher
the better. Depending on your font setup, some dialog windows may not fit on the screen at
resolutions of 800x600 or lower. However, the simulator can also be run as a standalone
application in text-mode without user interface if necessary.

performance   For acceptable performance, a fast processor (Pentium, Athlon) and 256 MBytes of main
memory are recommended. However, a Pentium-II 300 system with 64 MBytes is more
than adequate to run most of our educational designs. Naturally, the simulation of larger
circuits with thousands of gates may require 256MB of memory or more. On a modern
PC or workstation with a current JVM, the simulator performance should surpass one million
events per second. While Hades should also run on older hardware like a 486 or microSPARC
processor, you probably won't like the performance.

OS versions   When running a simulation, Hades stresses many parts of the JVM and operating system,
especially the graphics system due to frequent full-screen repainting, but also the thread syn-
chronization and memory management including garbage-collection. Should you experience
problems with the simulator, please ensure that your operating system libraries and device
drivers (graphics card) are up-to-date. Note that no stable Java virtual machines are available
for Windows 3.11 and older version of Linux (e.g. kernels 2.0.x).

3.3   Choosing a Java virtual machine

Since its introduction in 1995, the Java platform has evolved dramatically from the first
JDK 1.0 release to the current JDK/JRE 1.5 (Java 5) and the upcoming next release (Java 6).
The current version of Hades uses large parts of the Java class libraries, including some
methods and classes first introduced with JDK/JRE 1.4. Therefore, you need a Java virtual
machine that supports Java 1.4 or higher to run the Hades software.

recommendations   The following short list shows which JVMs are known to work with Hades on Windows,
Linux/x86, and Solaris systems:

```
     Windows 95/98/ME        Sun JDK 1.4.2
                             IBM JDK 1.4.1

     Windows 2000/XP         Sun JDK 1.5.0
                             Sun JDK 1.4.2
                             IBM JDK 1.4.1

     Linux (x86)             Sun JDK 1.5.0
                             Sun JDK 1.4.2
                             IBM JDK 1.4.1
                             Blackdown JDK 1.4.2

     Solaris (Sparc)         Sun JDK 1.5.0

     MacOS X                 Apple JDK 1.4.2
```

Current versions of the JDK/JRE are available for download from the Sun Microsystems Java
website, `www.javasoft.com`. For other systems, please contact your system vendor for the
availability of a suitable Java virtual machine.

3.4 Hades Download

Unless you receive the Hades software on CDROM or magnetic media, please visit the Hades homepage, `http://tams-www.informatik.uni-hamburg.de/applets/hades/` for information, design examples, and software downloads. *Hades homepage*

Follow the link to the *download* page, then download and save a copy of the `hades.jar` file. This archive is in executable JAR/Zip-format and includes the complete Hades software and simulation components. *hades.jar*

After downloading, *do not unpack* the archive file, unless you want to modify the software. All current Java virtual machines work much faster when accessing classes and resources from the packed archive file, instead of loading hundreds of individual files. Also, the signatures and main-class attributes stored in the archive will not work after unpacking. However, if you suspect that the `hades.jar` file was damaged during download, you can use your favorite packer tool to list the archive contents. For example, try to open the archive in WinZip, or use the `jar`-utility program from the JDK, `jar tf hades.jar`. This should print or list several hundred files.

The download area on the Hades homepage also offers some documentation and design examples. All recent documentation files use the PDF document format, while older files are available in (compressed) Postscript format. The design examples archive files are in ZIP-format and should be downloadable with all current browsers. *docs and examples*

3.5 Recommended file structure

The next step is to create a directory for your own design files as well as the examples design files. Figure 14 shows the recommended file structure with a subdirectory called `hades` below your home directory on a Linux system. In the screenshot, the newly created directory is called `/home/hendrich/homework/hades` and is used to hold the `hades.jar` executable JAR-archive with the Hades software. Below that directory, the `t3` subdirectory holds some `.hds` design files, e.g. `/home/hendrich/homework/hades/t3/datapath.hds`. Another subdirectory, `examples` holds a few other subdirectories with Hades design examples, e.g. the `bilbo/bilbo.hds` file. *Unix/Linux*

A similar directory structure should be used on Windows systems. The screenshot in figure 15 shows the recommended setup with the `\hades` subdirectory, the location of the `\hades\hades.jar` executable JAR-archive, and the `\hades\examples` and `\hades\t3` example directories. *Windows*

If possible, try to avoid special characters or spaces in directory and file names. Such names are difficult to enter via the command line, and some versions of the JDK/JRE have bugs that may result in obscure problems. For example, the default location for a user's home directory on a German Windows XP system is `C:\Dokumente und Einstellungen\username\ Eigene Dateien\`, which is very long and contains three space characters. A new directory like `C:\users\username\` might be a better alternative.

Unlike the `hades.jar` archive, which should be kept packed, the design examples archive files must be unpacked before using the examples. When unpacking with a Windows based packer program like WinZip, always select the option to preserve the directory structure from the archive. If you uncheck the option, the Hades editor may not be able to load subdesign `.hds` files in hierarchical designs or resources referenced by design files. See the above screenshots for examples on how the final directory structure should look like. *unpacking examples*

Note that the directory structure used in the example archives is not always consistent (but cannot be changed easily for compatibility with older Hades versions). For example, the standard `hades-examples.zip` file includes the full directory structure, `/hades/ examples/x/y.hds`. When unpacking this archive, please avoid to create a nested directory structure (`hades/hades/examples/`), because this will not work with (older) versions of Hades. For example, with the directory structure shown in figure 15, you should unpack the `hades-examples.zip` file to the `C:\Eigene Dateien` directory. *avoid hades/hades/*
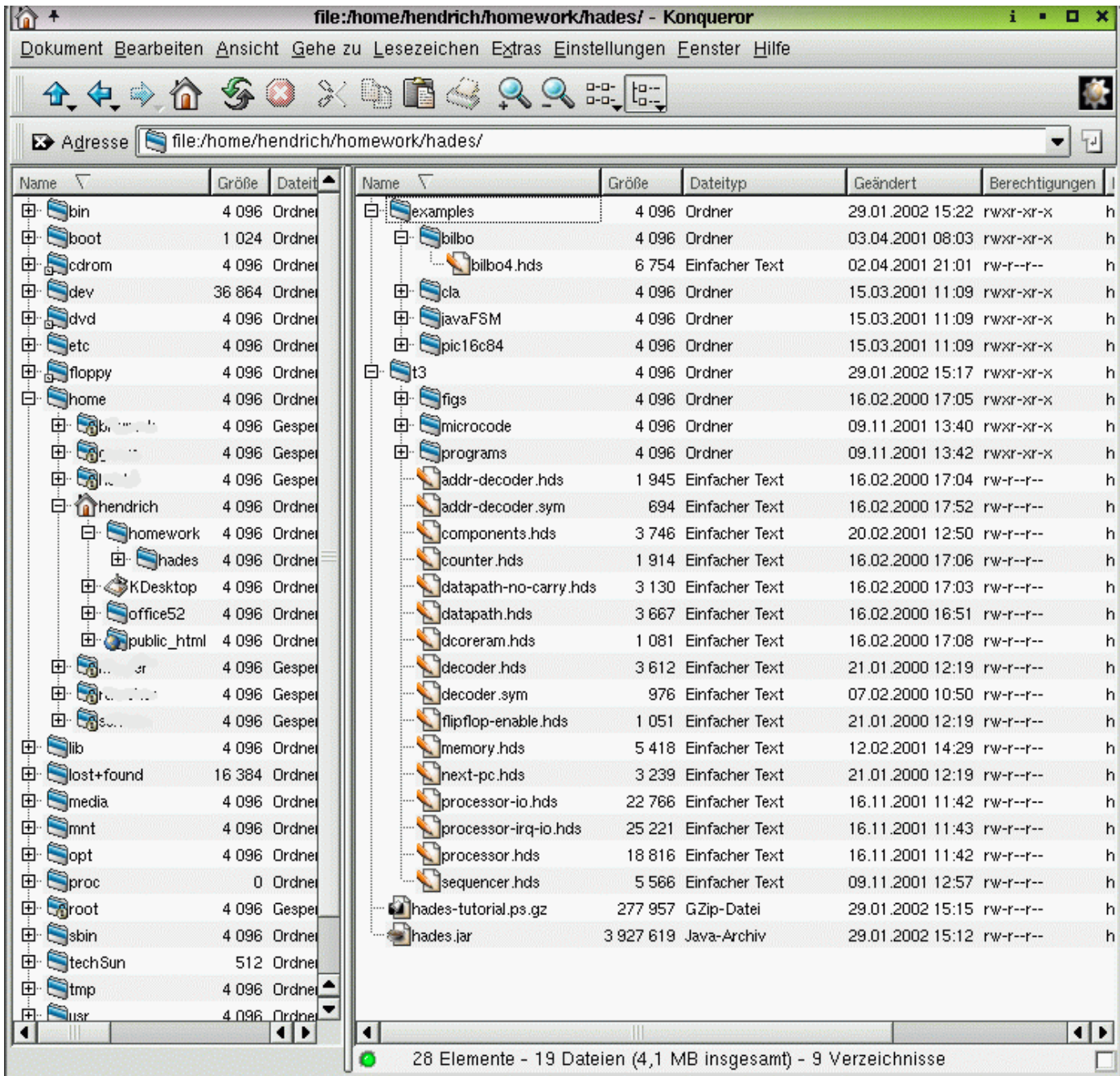
Figure 14: Example directory structure for single-user Hades installation on Linux.

3.6   Installation with JDK/JRE 1.4

Default installation

Windows   The default installation of the JDK or JRE on Windows changes the *Windows search path* to include the main Java executables (`java.exe` and `javaw.exe`, the latter without console window). Therefore, you can call the Java executable without further settings. For example, with the directory structure shown in figure 15 and the JDK 1.4 installed, you would start the Hades simulator from a command (DOS) shell with the following commmand line:

```
java -jar "C:\Eigene Dateien\hades\hades.jar"
```

The double-quotes are required because of the space character inside the directory name.
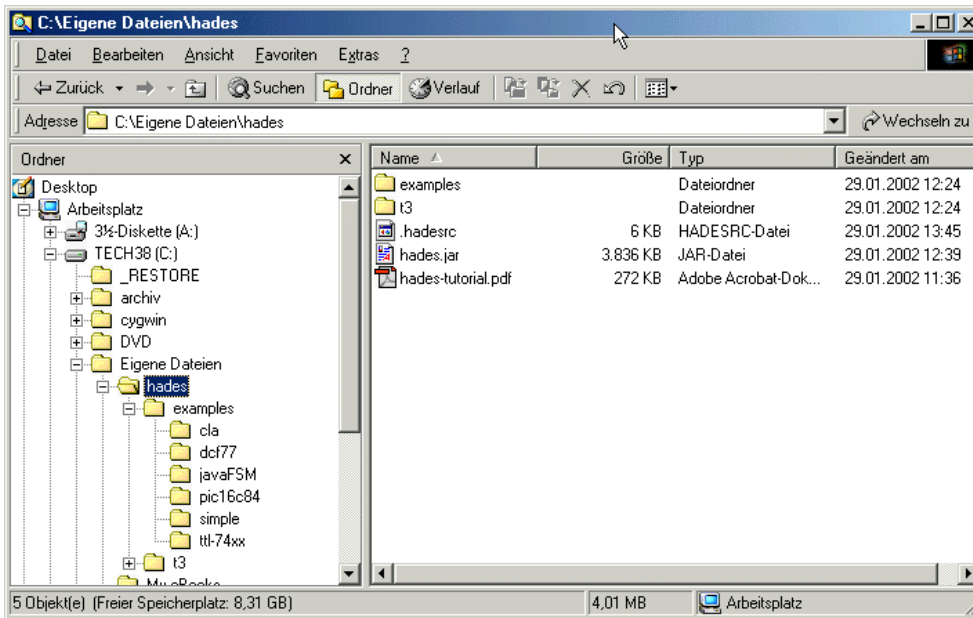
Figure 15: Example directory structure for single-user Hades installation on Windows.

If you call the `java` executable directly from your design directory, you can avoid to type the full path to the `hades.jar`, because the VM will then find the file in the current directory. Please check the release notes for your JDK to learn about additional command line options. For example, in order to increase the memory limit for the virtual machine, you could use the following command:

```
cd "C:\Eigene Dateien\hades"
java -Xmx256M -jar hades.jar
```

For long-running simulations, it might also be useful to select the so-called *server* virtual machine instead of the default *client* VM, if the former is available on your system. While the client VM is is optimized for quick application startup and desktop performance, the server VM uses more aggressive optimizations which often results in higher simulation speed:

```
java -server -Xmx512M -jar hades.jar"
```

On Windows systems with JDK 1.4 or higher, you should also be able to run JAR-archives by double-clicking the archive file. Therefore, you can drop a link from the `hades.jar` archive to your desktop or start-menu, allowing you to start the Hades editor just like any other windows application.

double-clicking hades.jar

If double-clicking the `hades.jar` archive does not work, please check that the JAR-archive itself is intact — see the FAQ (10.1.6). Second, another program might have registered itself as the application for the `.jar` filename extension. This frequently happens when installing a packer program after installing the JDK. If such cases, you may have to un-install and re-install the JDK (this is probably easier than editing the Windows registry directly).

The situation on Linux and Unix platforms is a little bit more complicated, due to differences between the vendor's virtual machines and the directory structure used by your operating system. For example, the JDK 1.4 could be installed in `/opt/jdk1.4`, `/opt/sfw/j2sdk1.4.2`, `/usr/lib/jdk1.4`, etc.

Unix and Linux

Also, the `java` executable might not even be included in your default search path, especially when multiple versions of the JDK/JRE are installed. In that case, simply use the absolute path to reference and call the Java virtual machine executable. For example:

```
cd /home/hendrich/homework/hades
/usr/lib/jdk1.4.2/bin/java -jar hades.jar
```

Using the JDK extension package mechanism

Java extension packages   Instead of manually setting the CLASSPATH environment variable, all versions of the JDK since 1.2.2 also support the *Java extension mechanism*. Basically, the trick is that the JDK automatically searches a special directory for `.jar` archives when looking for Java classes. Therefore, it is not necessary to set the CLASSPATH environment variable for JAR archives copied to the extension directory.

JDKDIR/jre/lib/ext   On current Windows and Unix versions of the JDK or JRE, this magic directory is called `<JDKDIR>/jre/lib/ext` and is initially empty after a default JDK or JRE installation (where `<JDKDIR>` means the base directory of the JDK installation, e.g. `C:\jdk1.4.2` or `/opt/j2sdk1.4.2`). Just copy the `hades.jar` file into this directory and change the file permission flags to allow read access for all users, if necessary. Note that the `ext` directory may only be writable for the system administrator. If you plan to use Jython for scripting you may also want to copy the `jython.jar` to the ext directory, see section 3.10.

For example, the resulting JDK directory structure may look like the following on a Windows system:

```
C:\jdk1.5.0
C:\jdk1.5.0\jre
C:\jdk1.5.0\jre\bin
C:\jdk1.5.0\jre\bin\java.exe          -- Java executable with window
C:\jdk1.5.0\jre\bin\javaw.exe         -- Java executable w/o window
C:\jdk1.5.0\jre\bin\...
C:\jdk1.5.0\jre\lib
C:\jdk1.5.0\jre\lib\ext
C:\jdk1.5.0\jre\lib\ext\hades.jar     -- Hades and jfig
C:\jdk1.5.0\jre\lib\ext\jython.jar    -- Jython (optional)
C:\jdk1.5.0\jre\lib\ext\...           -- more .jar files
```

other applications   You could then run the Hades editor and any other applications from the `hades.jar` archive
in hades.jar   just by giving the main class name as the argument to the `java` executable. For example, you could enter the following commands into a command (DOS) shell:

```
java hades.gui.Editor                     -- Hades editor
java hades.models.pic.PicAssembler        -- PIC 16 series assembler
java jfig.gui.Editor                      -- jfig graphics editor
...
java org.python.util.jython               -- jython interpreter
...
```

windows script   Naturally, it is possible to write short script files to execute the above commands, to avoid typing the command again and again. For example, to start the Hades editor you could write the following file and save it as `hades.bat`,

```
rem file hades.bat
rem run the Hades editor with JDK (1.4+), no CLASSPATH required,
rem because hades.jar is in JDKDIR\jre\lib\ext\hades.jar,
rem allow up to 256 MByte of memory
rem
javaw -Xmx256M hades.gui.Editor
```

Unix   Here is a similar script for Unix. Set the execute permission (x) bits for the script file and copy it to one of the directories in your search path:

```
# hades.bin
java -Xmx256M hades.gui.Editor -file $1
```

The obvious advantage of using the extension mechanism is that all classes in the extension packages are instantly useable by all Java applications, without any further complicated CLASSPATH setup. The disadvantage is the possible conflict between classes installed locally and the classes in the extension directory. Therefore, the extension mechanism should only be used for stable packages, but not during development.

3.7 Installation with other JVMs

The current version of Hades uses large parts of the Java class libraries, including the so-called Swing-based user-interface and Java2D-based graphics. Also, it relies on a few methods only introduced with release 1.4 of the JDK/JRE. Despite several ongoing efforts to provide alternative Java virtual machines, none of those projects has yet delivered a runtime that fully and reliably provides all functions rerquired by Hades.

The following list summarizes the situation:

- *JDK 1.3* and older: these versions of the Java runtime are obsolete and have been replaced by newer versions. Please use the JDK 1.4 or higher.

- *Microsoft VM* (jview): This VM originated as a complete Java runtime environment for Windows, with good performance and several interesting features, but lacking compatibility. Discontinued due to legal issues. Not suitable for running Hades anymore.

- *Kaffe*, *jamvm+classpath*, etc: The *classpath* project strives to provide a complete reimplementation of the original Java class libraries as free and open-source software. The classpath libraries can be combined with several different virtual machines to provide a full Java runtime environment.

  Despite major improvements during 2005, it is not yet possible to reliably run Hades on top of the *classpath* libraries, mostly due to a few subtle bugs in Swing and Graphics2D.

- *gcj+classpath*: The *gnu compiler for Java* allows compiling Java classes to native executable programs on a variety of platforms. As *gcj* is also based on the *classpath* libraries, the same restrictions are explained in the preceding paragraph apply.

3.8 User preferences and configuration

At application startup, the Hades editor first reads a global configuration file included in the *editor startup* `hades.jar` archive, and then searches for user-defined configuration files. This allows to change the default configuration and editor behaviour to your preferences. Currently, the editor uses the following sequence to read the configuration data. It first reads the default configuration from the file /hades/.hadesrc contained in the `hades.jar` archive. (If necessary, you can use the `jar` tool from JDK with the update option to change the global configuration file inside the `hades.jar`, but you should be careful to keep sensible default values.) Next, the editor tries to read a file called `.hadesrc` in your home directory. Finally, it tries to read a `.hadesrc` file in the current working directory. This hierarchy means that you can have global (default) attributes, your user preferences, and also local preferences for each Hades design or project.

For example, you can decide whether the simulator should start automatically after a design file has been loaded, you can specify the initial window size and position, colors to be used for glow-mode, etc. See appendix A on page 116 for the list of the available property keys and their default values.

To edit the configuration files, start the Hades editor and then select *Menu ▷ Special ▷ Show properties...* to open the properties viewer. The screenshot in figure 16 shows the Hades properties viewer window, together with some property keys and values. The bottom part of the properties viewer window shows the filenames for your user- and local- (working directory) properties files. Clicking on one of the buttons will export all properties from the dialog window to the corresponding file. Note that the propertiews viewer only exports property keys whose values start with `"hades"` or `"jfig"`. Naturally, you can also write and edit the `.hadesrc` startup files directly with your favorite editor.
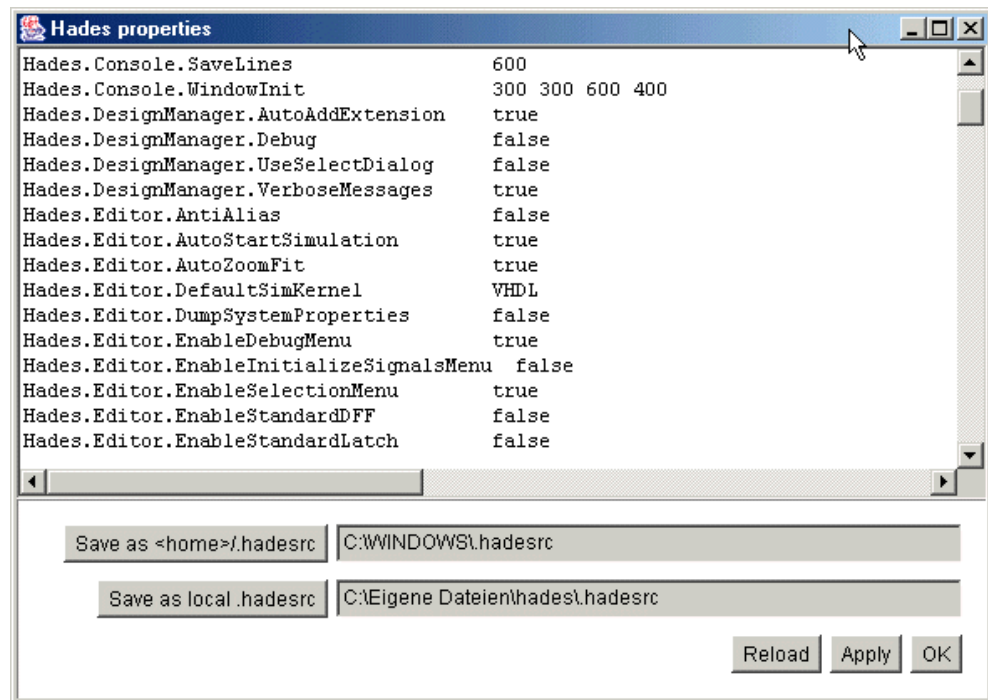
Figure 16: Screenshot of the Hades properties viewer, showing a few of the properties which control the editor and simulator startup behaviour.

3.9   Registering .hds files on Windows

If you prefer a document-oriented style of working, you may want to register a new file type for Hades design files and to associate that file type with your JVM and the Hades editor. Upon double-clicking a `.hds` design file, your operating system would then start the Java virtual machine, initialize the Hades editor and then load the design file into the editor.

Naturally, the actual GUI settings required to change the file associations depend on your version of Windows, your language settings, and your Java virtual machine. As examples, figures 17 and 18 show the GUI settings for a Windows 98 system (English) and a Windows ME system (German language), but the steps are similar for Windows XP. You first open the *Folder options* dialog window from an *Explorer* window or the *Control Panel*. In the dialog window you select the *File Types* panel. Select *New Type* to create the `.hds` file type or *Edit* if it already exists. Enter the file type description, possibly a MIME type, and then select the *Open* action. Provide the corresponding command line, depending on your JVM, JVM installation path, `hades.jar` location, and any additional options like the maximum memory limit for the JVM.

Note that the Hades editor only understands a few command line options and exits with an error message if the options or parameters are wrong. This can be hard to debug, because the error messages will not be visible when running with the window-less `javaw.exe` JVM. If the editor works when started manually, but instantly dies when called via the file-type mechanism, be sure to check the command line (especially the `-file` option).

You can also directly use `regedit` to create the necessary registry keys, but the required steps depend on your version of Windows and cannot be described here. For other operating systems, consult the systems documentation about how to create and register file types with your file manager and desktop environment.

3.10   Jython

scripting   While many scripting languages are available for Java, Jython is certainly one of the best. It is an implementation of the high-level, dynamic object-oriented language Python written in
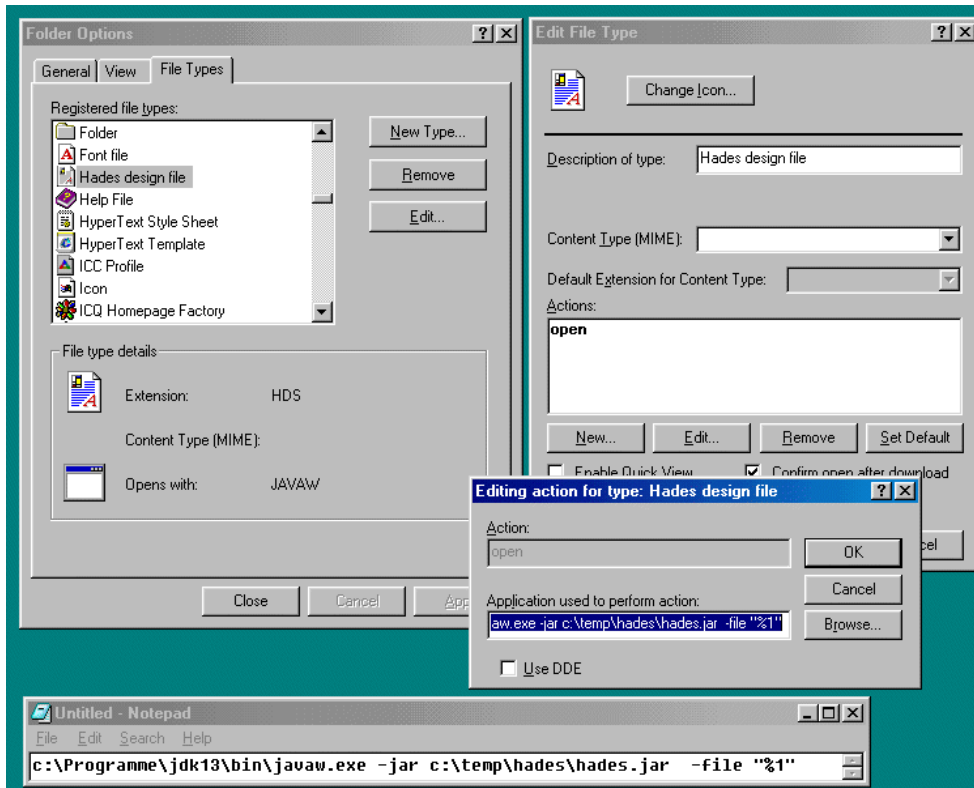
Figure 17: Screenshot showing the Windows folder options and file types dialog. Follow the instructions in the text to register the Hades design files (.hds) as a new Windows file type.
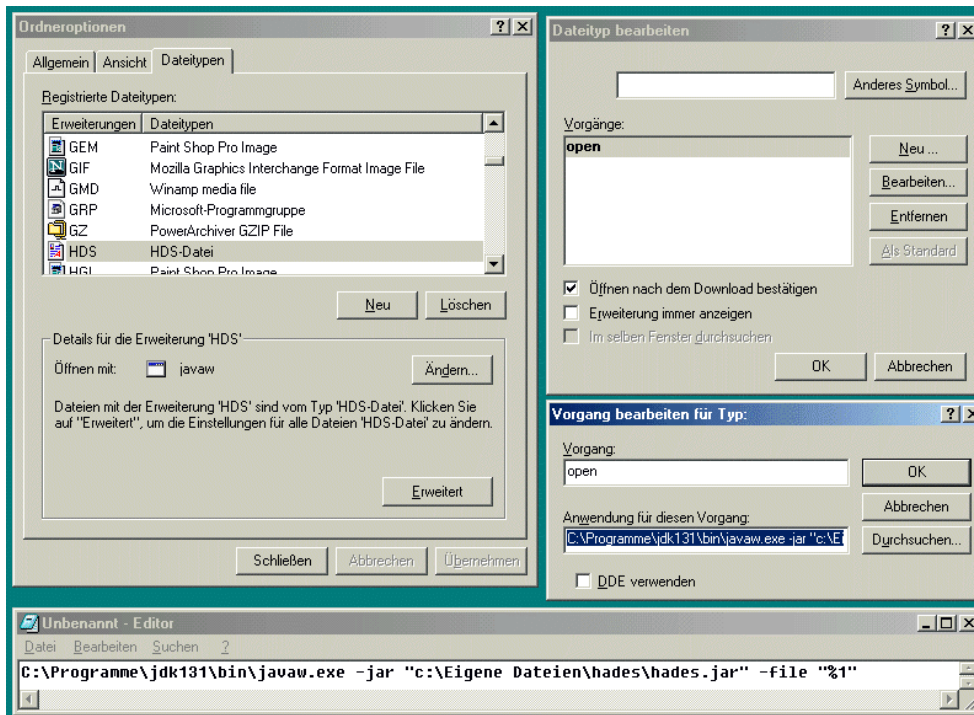
Figure 18: Screenshot showing the Windows folder options and file types dialog. Follow the instructions in the text to register the Hades design files (.hds) as a new Windows file type.

100% Pure Java, and seamlessly integrated with the Java platform. See chapter 8 for details about the integration of Hades and Jython.

To install Jython, visit the project homepage at `www.jython.org` and download the installer. Follow the instructions from the `www.jython.org` website through the installation process. It creates a directory structure, holding a `jython` startup script, the main `jython.jar` software archive, and several examples and library directories. The startup script will include the platform- and JVM-dependent CLASSPATH settings. In order to use the Jython with Hades, you will have to edit the startup script to include the `hades.jar` archive with the (possibly quite long) CLASSPATH statement. To start the Hades editor, substitute `hades.gui.Editor` as the main-class for the `org.python.util.jython` class.

When using Jython regularly, you might consider copying the `jython.jar` archive to the extension package directory, which makes most functions of Jython available to all your Java software without extra CLASSPATH settings. See section 3.6 on page 22 for details.

3.11   Multi-user installation

sharing resources   The installation instructions from the above sections are meant for single-user machines. Naturally, installing a separate copy of the `hades.jar` archive for each user on a multi-user machine will work, but this is not always the optimal solution. A similar arguments holds for the Hades example files, which might also be shared among many users.

The standard solution is to install the shared files, namely the `hades.jar` and the default example files, into a shared directory, e.g. a Windows network drive. Each user would still set up a private design directory.

When using the JDK, the administrator could copy the `hades.jar` archive file into the extension directory, which makes the software available to all users. Otherwise, it might be a good idea to provide a script to start the editor, and to put this into a system directory (like `/usr/bin`).

3.12   Applet installation and issues

browsers   The Hades editor and simulator can also be run as an applet inside most current web browsers, including Internet Explorer, Mozilla, Firefox, or Opera. However, as explained above, your browser will have to support least Java version 1.4 in order to run Hades. We recommend to install and use JDK/JRE 1.5.0 (or higher) as the Java plugin for your browser; see your browser documentation for details. Please note that running the Hades applets will not work with obsolete browsers like Netscape 4.x. If you are running and older version of the Internet Explorer, please check that you are using the external Java plugin instead of the built-in Microsoft virtual machine (*jview*).

download times   When you first visit a web page that includes an applet, the browser will automatically download the required applet class archive file(s) and then attempt to start the applet. In the case of Hades, this implies an initial download of about 5 MB size for the full `hades.jar` class archive file. While this corresponds to only a few seconds of delay over a broadband connection, it translates into several minutes of download time over a typical modem connection. Please be patient! After the first download, your browser should have cached the class archive file, so that subsequent visits of the applet pages will load much faster.

To reduce download time, we use a special stripped version of the simulator packed into a file called `hades-applet.jar` in our demonstration applets. This reduces the initial download to about 2 MB filesize.

security and file access   The default security settings used by the JDK/JRE for applets prohibit all potentially dangerous operations, including all access to files on your own computer. Therefore, you cannot load or save design files when running Hades as an applet, unless you change the security settings. If you want to enable file access for the Hades applet, you will have to edit the *Java Policy file* called `.java.policy` and located in your home directory. The JDK includes an extra program, `policytool`, to edit the policy file. If you don't like the GUI of policytool,

you can also edit the policy file directly with your favorite text editor. However, the JVM SecurityManager is sensitive to syntax errors in the policy file and will (silently) deny access to resources.

The following example shows how to enable full file access (read, write, delete, and execute) *policy file*
to all files on your local machine. Naturally, these settings are not recommended due to the security problems:

```
/* AUTOMATICALLY GENERATED ON Thu Oct 04 11:12:41 CEST 2001*/
/* DO NOT EDIT */

grant {
  permission java.io.FilePermission "<<ALL FILES>>",
                        "read, write, delete, execute";
};
```

A much more secure way is to grant applet access only to applets from specified servers, and only to specified files, for example:

```
grant codeBase "http://tams-www.informatik.uni-hamburg.de/applets" {
  permission java.io.FilePermission "/home/hendrich/homework/hades/",
                        "read, write";
};
```

Substitute the corresponding names for the applet server, the permissions (e.g. read only), and the files (e.g. your Hades design directory).

3.13 Developer Installation

While the Hades framework includes the most common simulation models for digital system simulation, you might want to add new, specialized simulation models or even editing functions. Given the Java concept of dynamic class loading, it is not only possible but very easy to extend the framework with your own classes.

If you are using the JDK, you even have all the required tools, including the *javac* Java *JDK or IDE?*
compiler and the *jar* archiver. While the JDK tools are fine for small projects, you may prefer to use your favorite integrated development enviroment (IDE). Popuplar examples of Java IDEs are Eclipse and Netbeans. Please consult the documentation for your IDE about how to import existing Java classes and how to setup the build environment. Most IDEs provide the option to directly import Java classes and packages form JAR- or ZIP-archive files. After the import, the class and source files should show up in your IDE's class browser.

If you are working with the JDK, the following setup has proven useful.

- Due to the Java class naming convention, it is possible to have a single root directory for all your Java projects. Create such a directory, if you don't already have it, e.g. `C:\users\hendrich\java` (Windows) or `/home/hendrich/java` (Unix, Linux)

- Unpack the *hades.zip* archive file into that directory. This will create several new subdirectories, namely the `hades` (simulation framework and models) and `jfig` (graphics) directories.

- Set the *CLASSPATH* environment variable to point to the root directory. That way, the Java virtual machine will be able to find and load all classes in your development directory tree. You may also want to include the current working directory, e.g.

  `set CLASSPATH=c:\users\hendrich\java;.`

- Edit Java source files with your favorite editor:
  `cd c:\users\hendrich\java`
  `emacs hades\models\gates\Nand2.java`

- Compile the Java sources with the *javac* compiler from the JDK or any other compatible Java compiler. For example, the *jikes* is much faster than *javac*.

  ```
  javac hades\models\gates\Nand2.java
  ```

- On Unix platforms, you can use the Hades *makefile* as a template to automate the build process. Just add new entries for all your new packages and classes.

debugging   Naturally, when using an IDE you would use its integrated debugger to test and debug your new classes. Because all classes in the *hades.zip* archive are compiled with full debug information, this will also allow you to traceback and analyze errors that occur inside the core Hades classes. Debugging with the plain JDK is a little more difficult, because the debugger included in the JDK is neither very powerful nor user-friendly. Therefore, it is often preferable to use an external Java debugger like *jswat* (*www.bluemarsh.com/java/jswat*) to analyze the behaviour of your code.

When testing your new classes, it might also be useful to switch-on the debug messages (and exception traces) from the Hades editor. The corresponding flag can be set at runtime via calling the static *setEnabled()* method in class *jfig.utils.ExceptionTracer*. From the editor, you can use the *'!'* bindkey to toggle the status. Another way to activate the debug messages is to start the editor with the `-vv` command line option, `java hades.gui.Editor -vv`. Some Hades classes additionally provide their own debug options. Check the list of properties in the editor, via *menu ▷ special ▷ show properties*, and look for properties like *Hades.DesignManager.VerboseMessages*.

Unfortunately, the error and warning messages are not handled fully consistently throughout the Hades framework. Some classes will log their messages to the Hades console window (open via the editor menu, *menu ▷ special ▷ show messages*), while other messages are directly written to the stdout and stderr output streams. To see all such messages, you should start the Hades editor from a shell.

4 Hades in a Nutshell

This chapter introduces the basics of creating and running a Hades simulation, including the most common editing and simulation commands available via the Hades editor user interface. In order to present a simple but complete example, a D-latch flipflop circuit will be constructed and simulated throughout the following sections. These topics are covered in this chapter:

- starting the editor
- loading a design or creating a new design
- basic editor settings like zoom or glow-mode
- adding simulation components to a design, for example logic gates, flipflops, switches
- wiring the components
- interactive simulation and simulation control
- using waveforms

4.1 Running Hades

Please install and start the Hades editor following the steps in the previous chapter, for example by double-clicking the `hades.jar` archive file. If everything goes well, the main editor window of Hades will appear on the screen after a few seconds.

Initially, the Hades editor window should look similar to figure 19. 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 via the *Edit ▷ Cancel* menu item or by pressing the *ESC* key. Note that the editor window needs to have the keyboard focus in order to react to keystrokes.

editor window explained

The panel below the object canvas contains several buttons that control the simulation engine. From left to right, the buttons have the following functions:

simulation control

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

In the default setup, the simulation will automatically be started in interactive mode directly during application 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. The remaining, leftmost control on the simulation control panel can be used to select one of several available simulation algorithms. For the simulation of digital systems, the preferred settings are *VHDL* or *VHDL (batch)* mode.

4.2 Using the Popup-Menu

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 *bindkeys* (shortcut keys) for the most important functions. While bindkeys 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, a few commands are only available via the popup-menu. Because the Hades editor relies on the Java AWT window toolkit, which

menu, popup, bindkeys
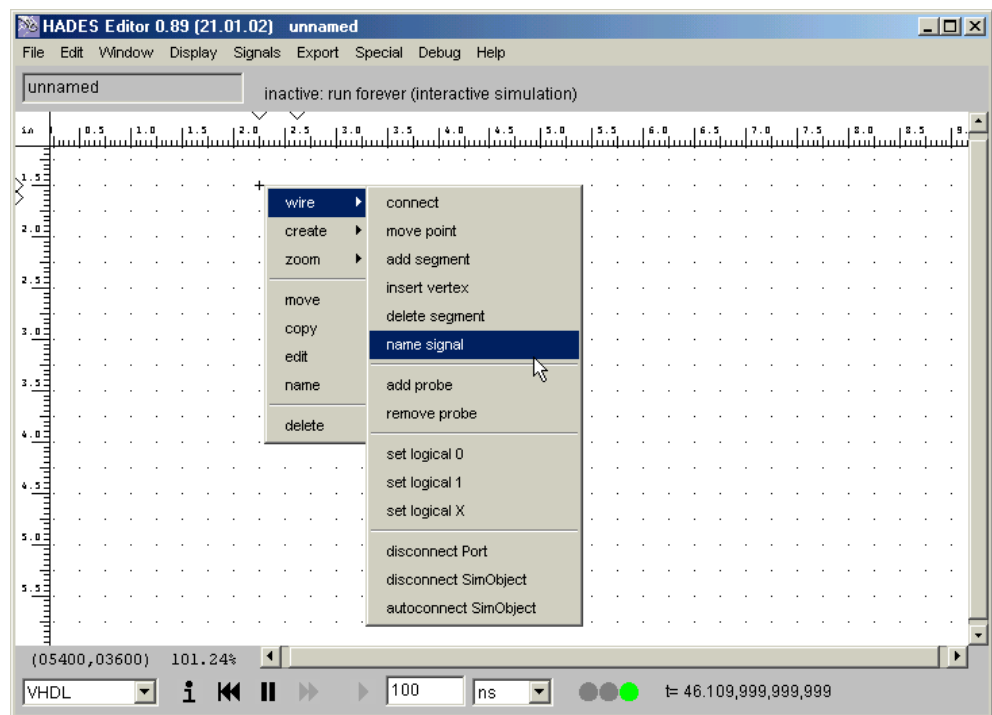
Figure 19: The Hades editor at start-up, with activated popup-menu. The main portion of the window is used by the *object canvas* which displays the current simulation setup (circuit). The panel above the object canvas displays the design name and status messages. The bottom panel is used to control (start/stop) the simulation.

tries to provide the native look and feel for each platform, the way to activate a popup-menu is platform dependent. 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, click and release the right mouse button. The popup menu will appear and you can select an item by clicking with left mouse button. On most platforms, the popup menu can also be activated with a combination of pressing a *modifier* key (e.g. *Meta* or *Alt*) and then clicking the left mouse button. Please consult your system documentation for the details on your system and Java virtual machine.

Now try to activate the popup-menu. It will appear at the current mouse position and should look similar to figure 19. It provides the following items:

wire   invokes a submenu used to create and edit wires (signals). See section 4.7 below for details.

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.

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 dialog window that allows to edit all user-settable properties of the given component.

name   change the name of a component.

delete   delete the component at the mouse position.

nested submenus   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. To show the selection of commands from the nested menus a notation with little arrow symbols is used in this tutorial. For example, the *popup ▷ name signal* command (compare figure 19) is used to rename a signal in the current design.
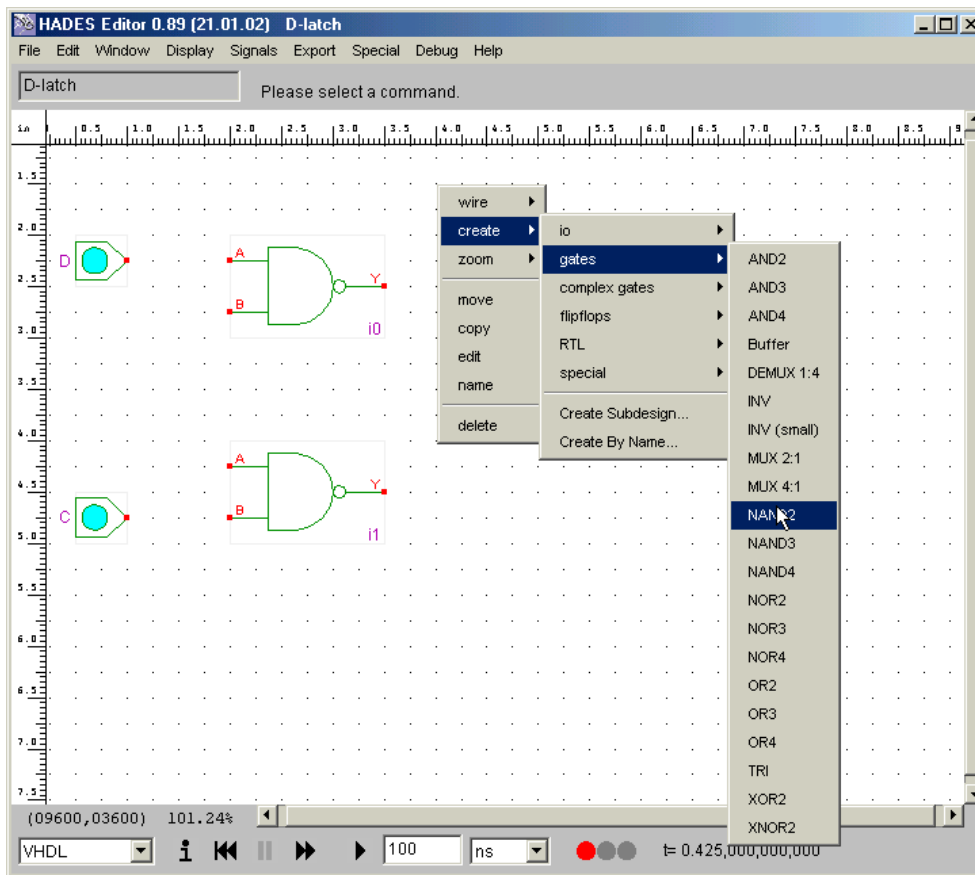
Figure 20: Selecting a two-input NAND gate from the editor popup-menu.

4.3   Creating Components

As usual, digital circuits in Hades are constructed from predefined simulation objects or *components*. The complexity of components may vary in a wide range, from simple logic gates and flipflops to complex system-level models like memories or processors.

In order to build a D-latch flipflop, just four NAND gates are needed. (the finished complete D-latch circuit is shown in figure 28 on page 42). To create a NAND gate, activate the popup-menu and select the create submenu. A new popup-menu should appear which in turn contains additional submenus called *IO*, *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 20. This instructs the editor to load both the simulation model (the Java code) and the symbol (graphical representation in the object canvas) of the requested gate.   *creating components*

Note that the creation of the first simulation component can take a few seconds, because Java enforces lazy program initialization and the virtual machine might have to load and verify several dozen classes when the first simulation model is requested. Naturally, overall performance depends largely on processor speed, but subsequent object creation should be much faster.

The popup menu will disappear, and the editor will switch to a rubberbanding mode, displaying a small rectangle at the mouse position. This rectangle indicates the bounding box of the gate symbol and can be moved to the desired position using the mouse. Click the left mouse button to place the NAND gate (top left corner) at the selected position, which will then be shown with its complete symbol.   *placing components*

Often, several instances of the same simulation model type are required for a design. To speed up this frequent operation, the editor will automatically create a new instance of the selected type (here the NAND gate), once the previous one has been placed. Therefore, just   *multiple instances*
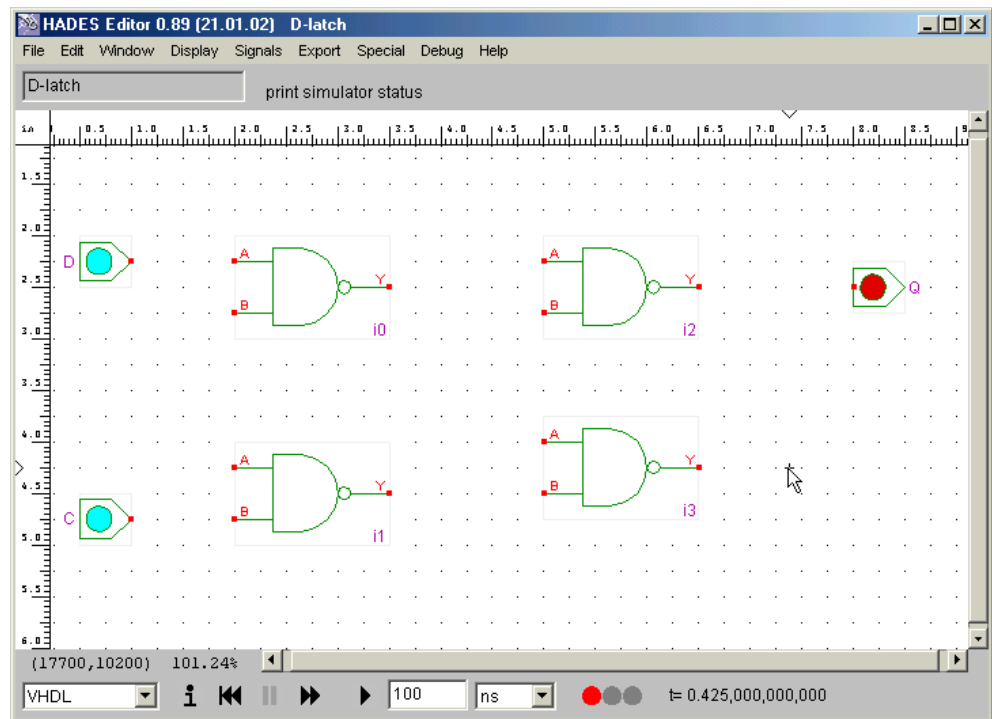
Figure 21: The D-latch design with all components before wiring

move the mouse to the target position for the next gate and click the left mouse button again to add another component to the current design. Finally, click the right mouse button or press the *ESC* key to cancel the *create* operation, as soon as you have created all required instances.

moving components   If you have dropped a component in a wrong position, you can always move it to a new
bindkey m   position. If necessary, cancel any ongoing command (like *create*). Then, position the mouse on one of the component symbol corners and select the *popup ▷ move* command or type the *m* bindkey. 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 one of the NAND gates a little bit to the right, and then back to its original position.

copying components   Another way to create multiple components of the same type is to use the *copy* command instead of the *create* command. Move the mouse to the source object and select *popup ▷ copy* or type the *c* bindkey. The editor 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.

deleting components   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 and select *popup ▷ delete*. The editor will delete the component without further notice, but in an emergency you should be
undo and redo   able to undo the delete command by selecting *undo* from the *Edit* menu. In theory, the editor supports unlimited recursive *undo* and *redo* operations. Unfortunately, *undo* does not work reliably under all circumstances yet.

4.4   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. The most important of these are *Ipin* for an input pin and *Opin* for an output pin of a design, which are accessed via the *popup ▷ create ▷ IO ▷ Ipin* and *Opin* menu items. 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 an
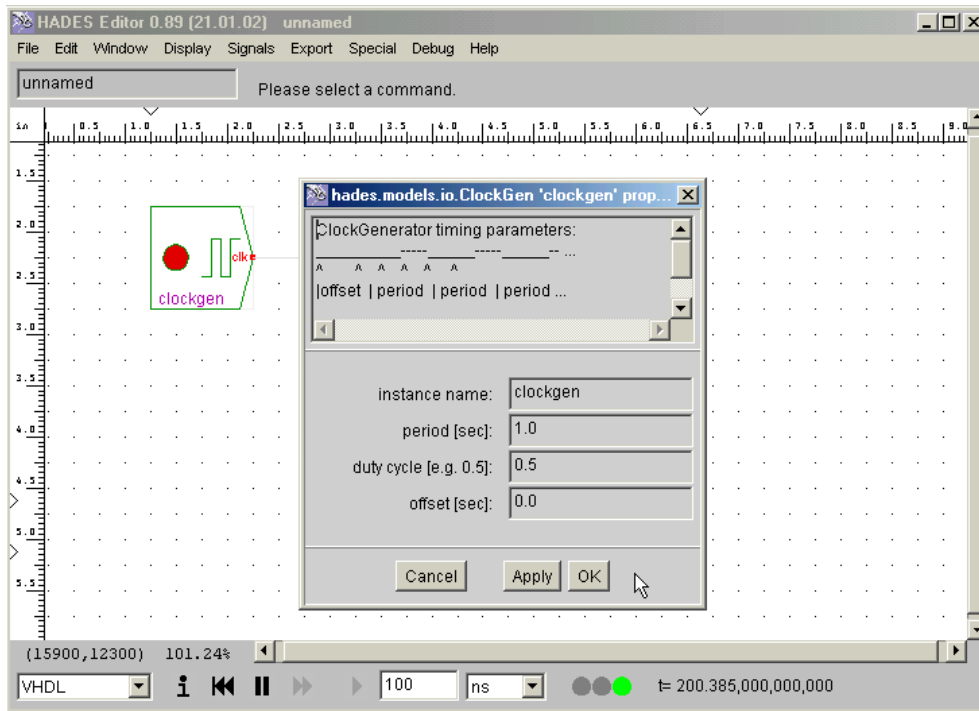
Figure 22: An example of the visual editing capabilities in Hades: A clock generator and its property sheet, allowing you to set and change its name and its timing parameters.

Opin component for the Q (stored flipflop data) output. Another Opin can be used for the NQ (inverted flipflop data) output. 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 *popup ▷ 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 normally called *Q* and *NQ*. Change the names of the Ipins and Opins correspondingly. Your design should now look similar to figure 21.

create by name

In order to keep the popup menu simple, not all available simulation models are also listed in the *popup ▷ create* submenus. One way to access and create any component available to Hades is to use the *popup ▷ create ▷ create by name* command, which takes a Java class name and uses the Java reflection mechanism to create instances of that class. See section 7.2 on page 59 for a detailed explanation.

4.5   Component Properties

properties and configuration

Most simulation components in Hades offer their own user interface to access and specify their (user-settable) parameters. Just position the mouse over a component and select the *popup ▷ edit* popup menu item to open the *property sheet* dialog window for that simulation component. The dialog window is not modal, so that you can open many property dialogs (for different components) at once. 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.

One example of the property dialog for a *clock generator* component is shown in figure 22. 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.
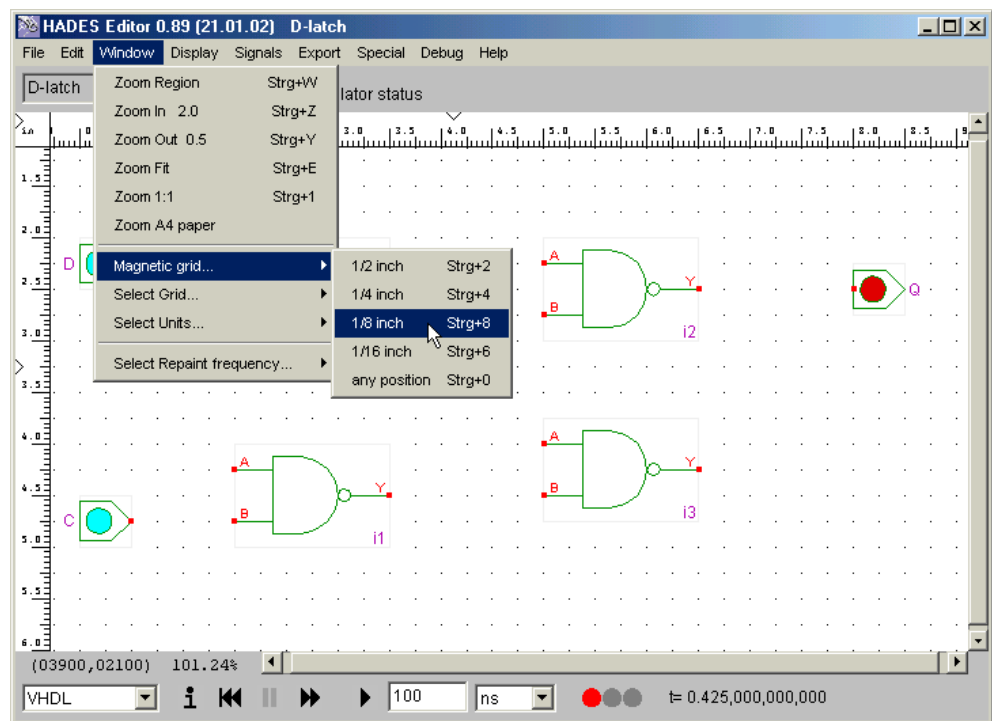
Figure 23: Selecting the magnetic grid

4.6   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 *magnetic 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).

magnetic grid   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 magnetic grid via the main editor menu. Select the *Window* menu and either the *Select Grid...* or the *Magnetic Grid...* submenus. The default is to snap the cursor to points 1/4th of an inch apart, which is useful for almost all 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 magnetic grid to 1/8th of an inch, compare figure 23.

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 *popup ▷ 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 keys or the scrollbars.

negative coordinates   While the Hades editor allows you to place objects anywhere you choose, you should probably restrict objects to the positive quadrant. For example, older versions of the `fig2dev` utility program used to export Hades schematics to output formats like Postscript or PDF silently clips objects with negative coordinates. If necessary, select the *Edit ▷ Move/fit design to positive quadrant* command to prepare your schematics for printing.

One very effective visualization aid towards the understanding of digital systems is the color-encoding of the logical values on wires during simulation. This idea, *glow-mode*, was borrowed from [Gillespie & Lazzaro 96]. Hades supports *glow-mode* color encoding for all values of the *std_logic* simulation model and for buses, too. To toggle glow-mode for all signals in the current design during a simulation, just select the *Display ▷ Glow Mode* menu item or press the *g* bindkey anywhere in the object canvas. Sometimes, it is useful to select glow-mode for a subset of signals only, for example to emphasize to critical signals or to switch off animation for unimportant signals. To this end, position the mouse over the target signal and press the *h* bindkey to toggle glow-mode for that signal only. Note that scripting can be used to automate the glow-mode settings instead of manual selection, see chapter 8.

glow-mode

bindkeys g and h

Unfortunately, the disadvantage of using *glow-mode* is the greatly reduced simulator performance, because the frequent screen updates easily take more processor cycles than the simulation itself. Also, graphics operations are still very slow and even unreliable on many Java virtual machines. Therefore, the editor uses a separate thread to minimize the number of total repaints. Use the *Windows ▷ Repaint Frequency* submenu to select the desired *frame rate* of the editor.

performance

4.7   Creating Wires

After some components are placed in a design, it is time to connect them. In Hades, the logical connection of simulation components is called a *signal*, while the name *wire* is used for the graphical representation of a signal. A wire consists of one or many *wire segments* and possibly *solder dots*, which indicate *wire segment* junctions. The graphical user interface is used to connect and manipulate *wires*, while the editor automatically updates the corresponding internal signal data structures.

signal vs. wire

A connection of a simulation component to a signal is only possible at 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 left mouse button. This creates a new wire and connects its first end to the component port. If you move the mouse, you will see that it traces a rubber-band to indicate the current position of the wire. Click the mouse on any intermediate points (*vertices*) you need to route the wire, then move to the destination port for that wire and click the mouse again. This attaches the wire to the port, and the signal is ready to use.

ports

vertices

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 magnetic grid, it should be quite easy to move the mouse to the component ports and correctly place intermediate points.

At the moment, the Hades editor does not support autorouting; it relies on the user to create wire vertex points where needed. Also, note that Hades won't enforce Manhattan geometry for wires. For example, a diagonal routing of the feedback lines in a flipflop circuit can greatly improve the readability of a circuit schematic.

no autorouting

As described above, the editor will automatically create a new wire (and signal), when it detects a mouse click on an unconnected simulation component port. This reaction is not always desirable, the more so as the risk of inadvertent mouse clicks may be high at small display zoom factors. Use the *special ▷ disable create signals* menu command to toggle the editor behaviour.

4.8   Adding Wire Segments

Many wires, however, are not just a point-to-point connection, but require connection to multiple 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 of the start, end, or intermediate points—of the wire, activate the popup-menu and select *popup ▷ wire ▷ add segment*. The editor will create a new wire segment attached to

complex wires
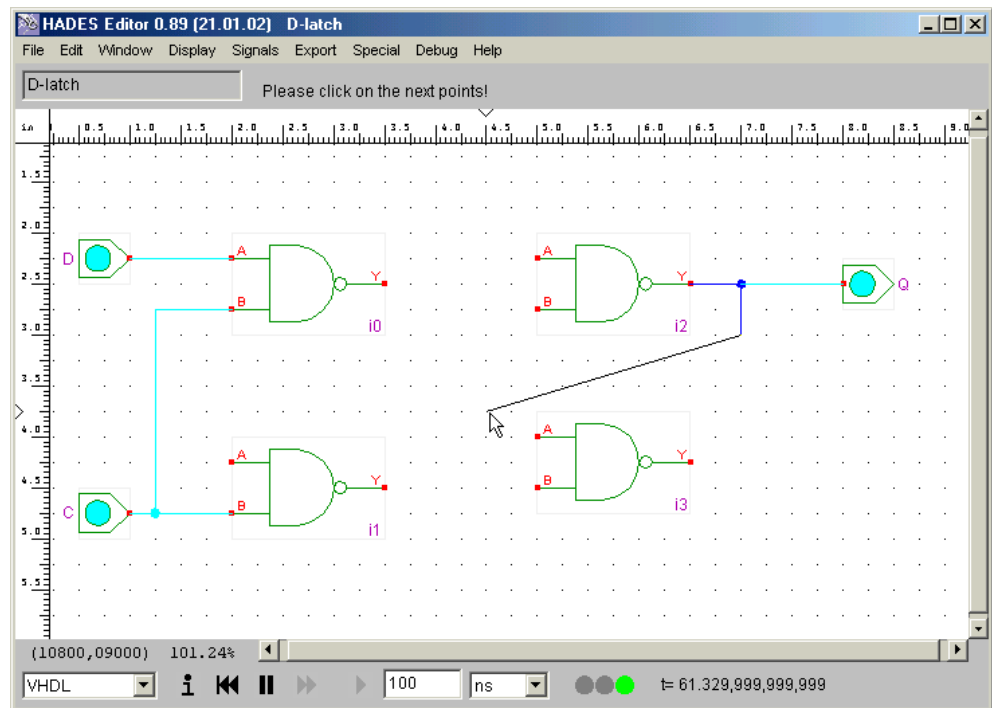
Figure 24: The completed D-latch design with all components and wires. During interactive simulation, you can change the input values to your circuit by clicking on the *Ipin* symbols.

the original wire and switch to the rubberbanding mode. As above, specify any intermediate points needed, then click on the component port to connect the wire.

bindkey w   Another, and faster, way to trigger the *wire ▷ add segment* operation is via the bindkey *w*. Just move the mouse to the position where to connect the new wire segment, press the *w* key, and use mouse clicks to specify signal vertices and finally the target port.

4.9   Connecting existing Wires

shift+mouse click   Sometimes, it is necessary to connect two existing wires together. To this end, you would move the mouse to a vertex of the first wire and then select *popup ▷ wire ▷ add segment* (or use the *w* bindkey) as described above. Next, move to the mouse to an intermediate vertex or an endpoint of the second wire. Finally, hold down the *SHIFT*-key and click the left mouse button to connect the two wires. Note that a simple mouse click (without the shift key) will not work, as the editor creates a new signal vertex and waits for another mouse click. (The reason for enforcing *shift+click* to connect wires is that this makes it more difficult to inadvertently create short-circuits between wires. Also, it allows to route many wires one over another without automatically connecting them, which can often greatly inprove the clarity of complex schematics. For example, all of the microprogram output signals in figure 9 on page 12 are routed as a single line, to keep the visual „overhead" of the control lines to a minimum.)

4.10   Moving Wire Points

moving wire points   You may also want to move signal vertices after initial placement. This is possible with the *popup ▷ wire ▷ 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.

bindkey o   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. The rubber-banding for the *move point* command is implemented only partially yet.

If you want to insert a new *vertex* point into an existing wire, for example to improve the signal routing, just move the mouse onto or near to (nearer than the magnetic grid spacing) that position. Then, select *popup ▷ wire ▷ insert vertex* or use the *v* bindkey, which instantly creates the new vertex. Of course, the new vertex will only be visible if it is not exactly in-line with the previous wire routing.

bindkey v

Note that the Hades editor will try to keep all wire segments connected if you move a component (this feature can also be useful to check 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.

4.11   Deleting Wires or Wire Segments

To delete a wire completely (including all its wire segments and all its connections to simulation components), move to a vertex or endpoint of the wire and select *popup ▷ delete*. Please make sure to position the mouse exactly, as the editor searches for the nearest graphical object to decide which object to delete (wire or simulation component). If necessary, you can try the *edit ▷ undo* menu item to restore a deleted signal and its previous connections to simulation objects. Unfortunately, the *undo* operation does not work reliably in all circumstances, so you might have to save and re-load your design. In any case, as the simulator cannot restore the events for the signal, you will have to restart the simulation after an *undo* operation.

deleting wires

Use the *popup ▷ wire ▷ delete segment* command or the *x* bindkey to delete a single wire segment nearest to the mouse position. The editor recognizes if the wire is split into two separate parts as a result of deleting the wire segment, and it will automatically update its internal data structures (creating a new split *signal* if necessary).

deleting wire segments, bindkey x

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 use the *w* bindkey to initiate the *add segment* operation. Once you are finished with all wires, your schematic should look similar to figure 28.

completing the example

4.12   Changing Signal Names

Whenever a new signal is created, the editor will also automatically assign a unique name to it. The default algorithm just numbers the signals based on the prefix *n* (for net) and the current count of signals, so that the initial signal names are *n0*, *n1*, *n2*, etc.

default signal names

In many cases, you might not care about the exact name of a signal, and the default names are as good as any. However, descriptive signal names are essential when you plan to use the waveform viewer (see chapter 6) to analyse and debug your circuits. Even in small circuits, meaningful names like *clk*, *nreset*, *carryout* or *addr.enable* are much better to work with than *n213*, *n10*, and *n42*. But even worse, the numbering scheme is applied to each subdesign. If you debug a circuit with many subdesigns, the waveform viewer will list many signals called *n* or *n17*, which are only distinguished by their full hierarchical name.

Therefore, it is good practice (but not required) to select meaningful descriptive names at least for all essential signals in your designs. To do this, position the mouse near to a wire segment of the signal, and type the bindkey *n*, or activate the popup-menu and select *popup ▷ name*. Enter the new name for the designated signal, and press *OK* to apply the new name.

naming a signal

The editor also includes a utility method called *renameToplevelSignalsAfterDrivers()* that can be called via scripting (see section 8) or from the editor menu via *menu ▷ options ▷ Call a method*, and then entering the string *renameToplevelSignalsAfterDrivers null* into the dialog window. This automatically renames all signals based on the name of the component that drives the signal. The idea is that names like *flipflop3.Q* or *xor2.Y* might be better than *n7* and

n112. Still, you should ensure that all your simulation components have descriptive names (see section 4.4 on page 33) before calling this method.

4.13   Editor Bindkeys

The following table lists the default bindkeys for the most important Hades editor operations. Note that the editor only reacts to keypresses when it has the keyboard focus. Also, some Java virtual machines still have bugs in the keyboard input, especially in combination with non-US keyboard layouts.

```
ESC    escape           cancel current operation
DEL    delete           delete component or wire
BSP    backspace        delete component or wire

c      copy             copy component
e      edit             edit component parameters
m      move             move component
n      name             name component

w      wire             add wire segment
v      vertex           insert vertex into wire
o      mOve vertex      move wire vertex
x      delete segment   delete wire segment
N      Name             name wire
W      Wire             create new wire at mouse pos.
                        (without port connection yet)
```

4.14   Loading and Saving Designs

saving designs   This is a good time to save your design. At the moment, Hades stores design data as plain ASCII text files, one file per *design* or schematic, which means that you can also use your favorite text editor to view or modify the design files. 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. Therefore, enter a suitable filename, e.g. `dlatch.hds`, and click the *OK* button. For a larger project with many subdesigns, you might want to create and use subdirectories. By convention we use the extension `.hds` for Hades design files,
.hds extension   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`.

Once you have saved your design, the editor will remember the filename, so that you can use the *File ▷ Save* menu command to save the design again. The *Save* operation does not overwrite the previous version of a design file, but automatically renames the existing file before saving the current design. The naming pattern used for the backup files is *designname.hds*_`<number>`. For example, the first two backups of *dlatch.hds* will be called *dlatch.hds_1* and *dlatch.hds_2*. By the way, the default filename is `unnamed.hds` in the current working directory.

loading designs   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 delete all current editor objects and start a new 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*.

Despite the overhead of parsing the design files and object creation, loading a small design file should only take a few seconds on current PCs or workstations. Naturally, loading larger hierarchical circuits with many subdesigns and thousands of simulation components can take much longer, especially when the Java virtual machine runs short of memory and needs to start garbage collections during circuit initialization.

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 25: The logic values used in Hades, based on *std_logic* and their associated default colors in *glow-mode*.

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 26: The NOT and AND functions of the std_logic system. Note that both functions include the standard boolean operations with arguments 0 and 1 as a subset (shown in bold-face). Also note that many input combinations result in U or X outputs.

4.15   Digital Simulation and StdLogic1164

modeling issues

While the Hades framework can be used for all types of discrete-event simulation, most of the available simulation components circuit as well as the different simulation engines are targeted towards digital circuit simulation. The goal of every simulation is to study the behaviour of some real system from a set of idealised models. This allows for a variety of descriptions with greatly differing levels of abstraction and detail. One very successful abstraction is switch-level simulation, where the complex, non-linear and time-continous behaviour of electronical circuits is modeled with a set of discrete logical values that change at discrete times.

The most simple simulation model is the Boolean algrebra with its two logical values 0 and 1 and the well-known switching functions (AND, OR, etc.). Unfortunately, this model often is too restrained; for example it is impossible to model important situations like short-circuits or undefined input values. Therefore, multi-level logic systems are often used to model and simulate digital circuits. Several tools support a three-level model with the values 0, 1, and X, where the X level is included to model undefined values during circuit initialization or to indicate errors like a bus with multiple conflicting drivers.

std_logic

However, even more complex logic systems are required to model situations like open-collector output gates or buses with tri-state drivers and pullup resistors. Therefore, all gate-level simulation components in Hades use the de-facto industry standard logic model, the nine-valued *std_logic* system [IEEE 93b] used in the VHDL simulation language [IEEE-93a]. Please skip the next few paragraphs if you are already familiar with digital simulation and the *std_logic* multilevel logic.

Figure 25 shows the nine logic values from *std_logic* and their meanings; these are implemented in class *hades.models.StdLogic1164* together with the basic logical operations. At
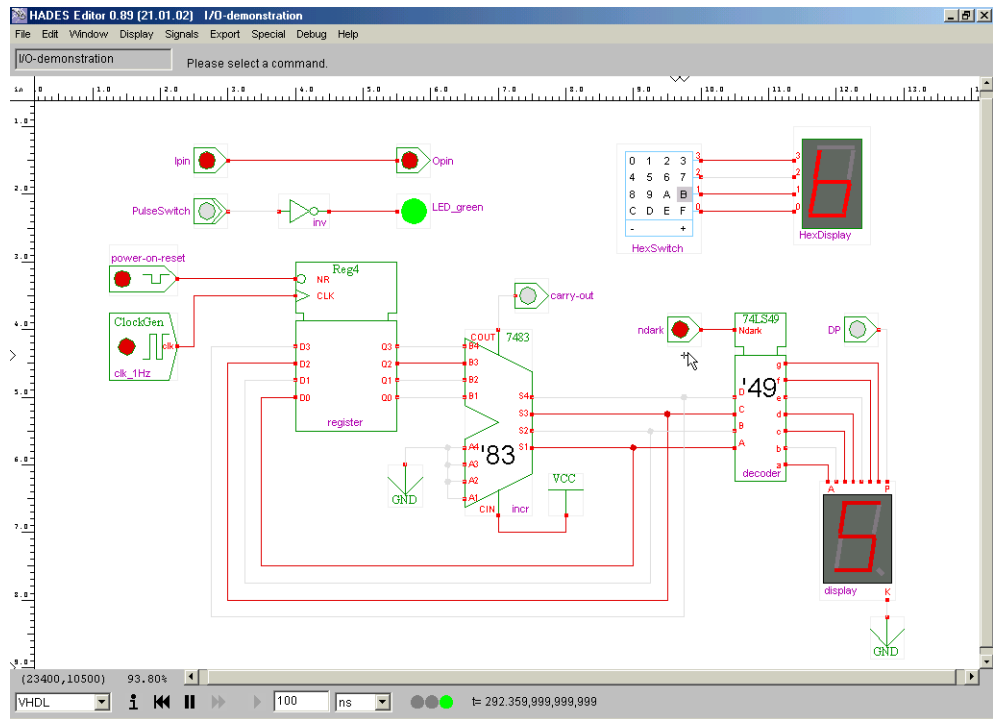
Figure 27: A collection of interactive I/O components: Ipin (switch), pulse switch, power on reset, clock generator, hex switch, Opin (LED), LED, and hex display.

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.

4.16   Interactive Simulation and Switches

Most digital simulators run the simulation in batch-mode. After designing your circuit you have to compile it into the internal representation used by the simulator. Next, you have 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, all *Ipin* components and several other components from the Hades *I/O* library work like small switches. Figure 27 shows a collection of the *interactive* input and output simulation 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 the left mouse button on an *Ipin*, 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.

Note that all *Ipin*s are initialized to the *undefined* (U) state at the start of the simulation. 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. However, any std_logic value can be selected as the default start value for an *Ipin* via its property dialog, when appropriate. 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 - 0

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 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.

clock generator

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.

Opin = LED

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.

hex switch,
hex display

4.17   Waveforms

Naturally, the Hades framework also includes functions to collect and display waveform data for logical signals. Figure 28 shows the D-latch circuit with probes added to the signals. Please refer to chapter 6 on page 51 for a detailed description on how to use waveforms and the waveform viewer.

Again, please remember to change the default signal names to descriptive names before using the waveform viewer, especially when your circuit contains nested subdesigns. See the tip in section 4.12 on page 37 on how to rename the signals.

4.18   Tip: Restarting the Simulation

While the Hades editor and simulator try to fully support all edit operations during a running simulation, this is not always possible. For example, restoring the simulator event list after deleting a component and then undoing the delete is problematic. Also, some *static* simulation components create events only under special conditions or only at the start of a simulation. The latter is true for the *VCC* or *GND* components, which only generate their (constant) output value once at the beginning of a simulation. Edit operations that involve such simulation components, for example connecting a wire to a *VCC* instance during a running simulation, may result in inconsistent signal values. Often, such conditions are easily to spot in *glow-mode*, because un-initialized signals are highlighted in *cyan* (U) or *magenta* (X) colors. Use the simulation control panel to re-initialize the circuit by first pressing the *rewind* button and then the *run* button.

VCC, GND problems
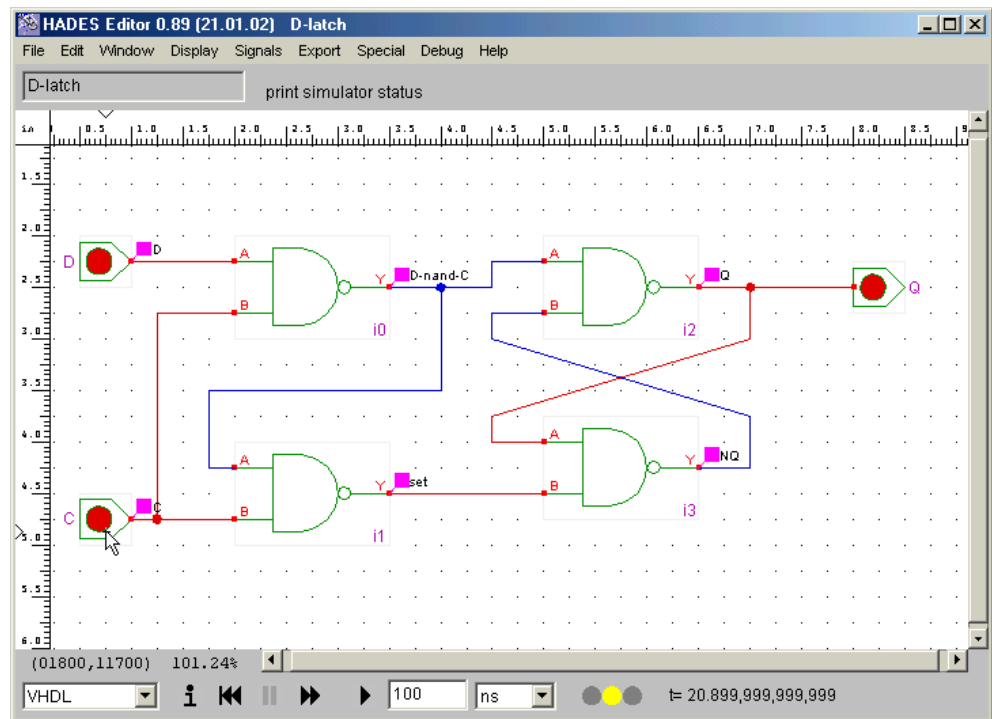
Figure 28: The D-Latch circuit with active waveform probes

4.19   Tip: Unexpected Timing Violations

synchronisation During interactive simulation, all events generated internally by your design are scheduled
problems together with the user-events generated by clicking on input components like *Ipins*. On any
user-generated event, the current simulation time is used as its time stamp. This behaviour
may result in unexpected faults in your simulation whenever the events feeds the input of
a flipflop or similar component, which might detect a setup- or hold-time violation with
respect to its clock signal. Usually, the flipflop will then generate a X value on its output
which propagates to the rest of the circuit.

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 simulation time. Note that real circuits also suffer
from these timing problesm, because the timing of input changes 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.

5   Advanced editing

This section introduces a few of the advanced editor commands and features of the Hades framework. First, section 5.1 motivates and explains how to create hierarchical designs in Hades. Next, section 5.2 lists the most useful editor bindkeys. Section 5.3 demonstrates printing via both the Java native printing functions and the external fig2dev program. Finally, section 5.4 shows how to export gate-level circuits to VHDL for simulation and logic synthesis with external EDA tools.

5.1   Hierarchical Designs

For circuits with more than a few dozen gates, most designers will prefer a *hierarchical* design, where the complete circuit is partitioned into smaller blocks. These blocks, often called *subdesigns*, 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 displayed in the circuit schematic of the parent design.

subdesigns

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.

advantages

Often, 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. Often, 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, while a *top-down* design methodology is more difficult to achieve.

top-down
bottom-up

As an example, the remainder of this section will use the 8-bit carry lookahead adder first presented in figure 4 on page 7. An excellent description of this circuit including the theory and performance aspects of the different kinds of adders is found in [Hennessy & Patterson]. The hierarchical version of the adder circuit shown in the figure consists of eight instances of the 1-bit adder and seven instances of the carry generate and propagate block. While the 8-bit adder is still a very small circuit, the *flat* design variant would already fill one very large schematic with several dozen gates.

8-bit CLA adder

The *bottom up* design of the 8-bit CLA adder starts with the small building blocks, here the 1-bit adder and the 1-bit CLA block. The very simple schematic for the 1-bit adder is shown in figure 29. The *Ipin* switches and connectors define the external inputs *ain*, *bin*, and *cin*, while the *Opin* connectors define the *si* (sum), *gi* (generate) and *pi* (propagate) outputs. The two XOR gates calculate the one bit sum of the three inputs while the AND or OR gates are used to calculate the *gi* (generate) and *pi* (propagate) signals.

1-bit adder

Once the above 1-bit adder circuit is complete, save the design under a suitable name, for example *sum.hds*, using the *File ▷ Save as* menu command in the Hades editor. Next, select the *Edit ▷ Create Symbol* editor command, which automatically creates the *graphical symbol* required to display a subdesign in the toplevel design. The symbol data is written to a text file whose name is constructed from the current design name and the *.sym* extension. In the example, the symbol file for *sum.hds* is called *sum.sym* and written to the same directory as the *sum.hds* design file. The automatically created symbol file includes port connections for all

symbol creation
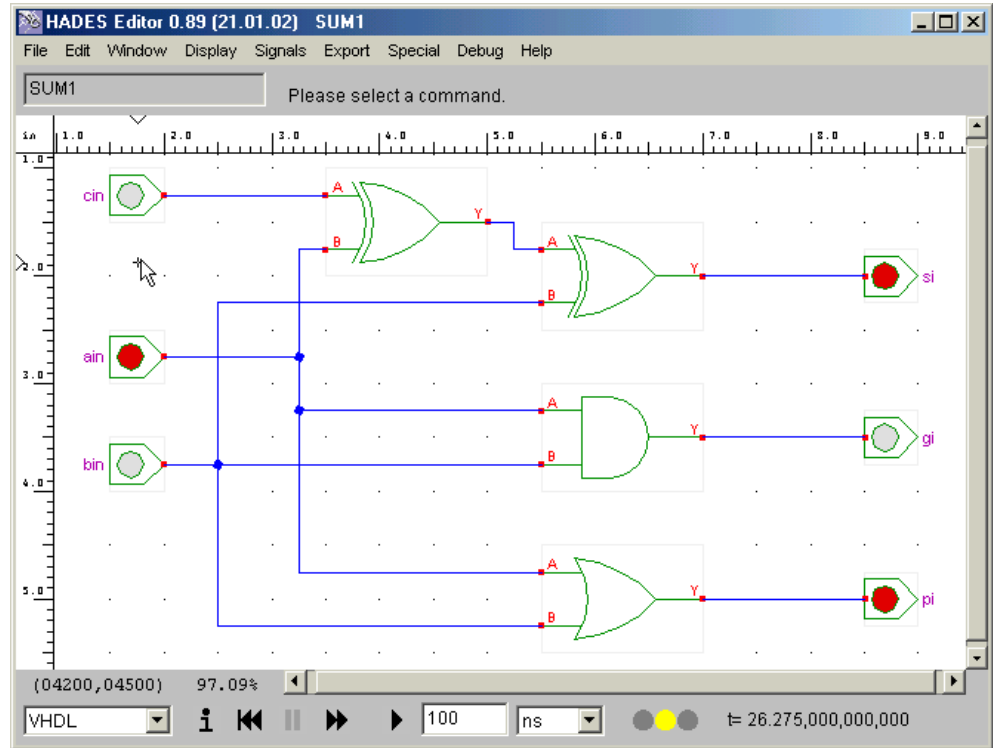
Figure 29: 1-bit adder with carry generate and propagate signals
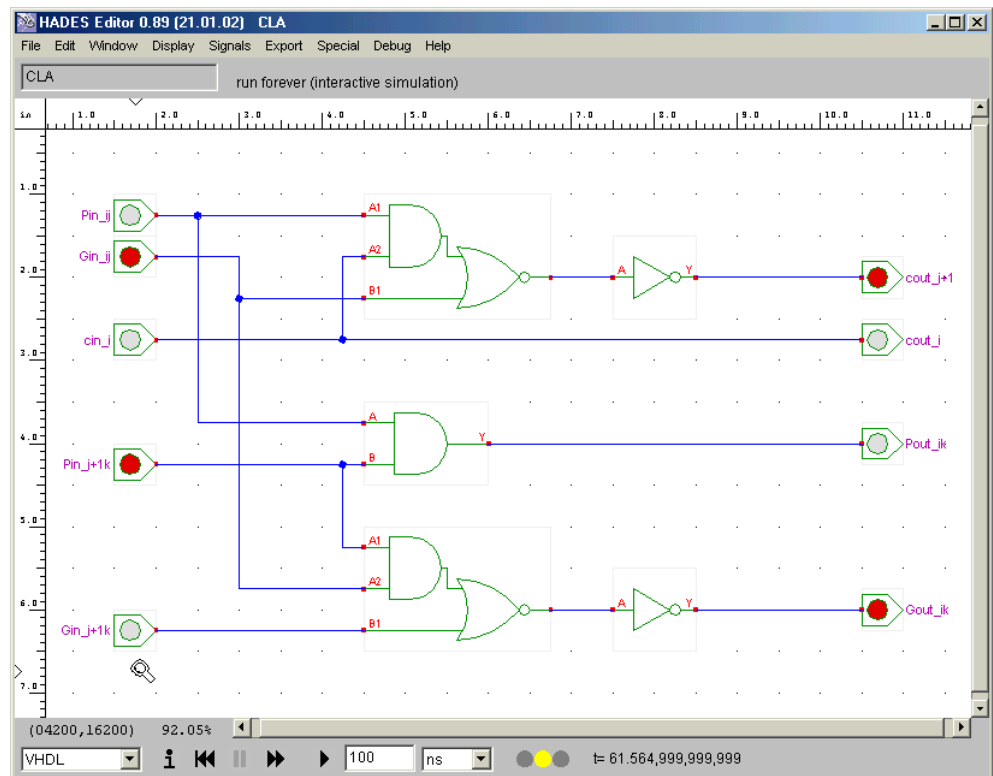
Figure 30: CLA block for the 8-bit CLA adder
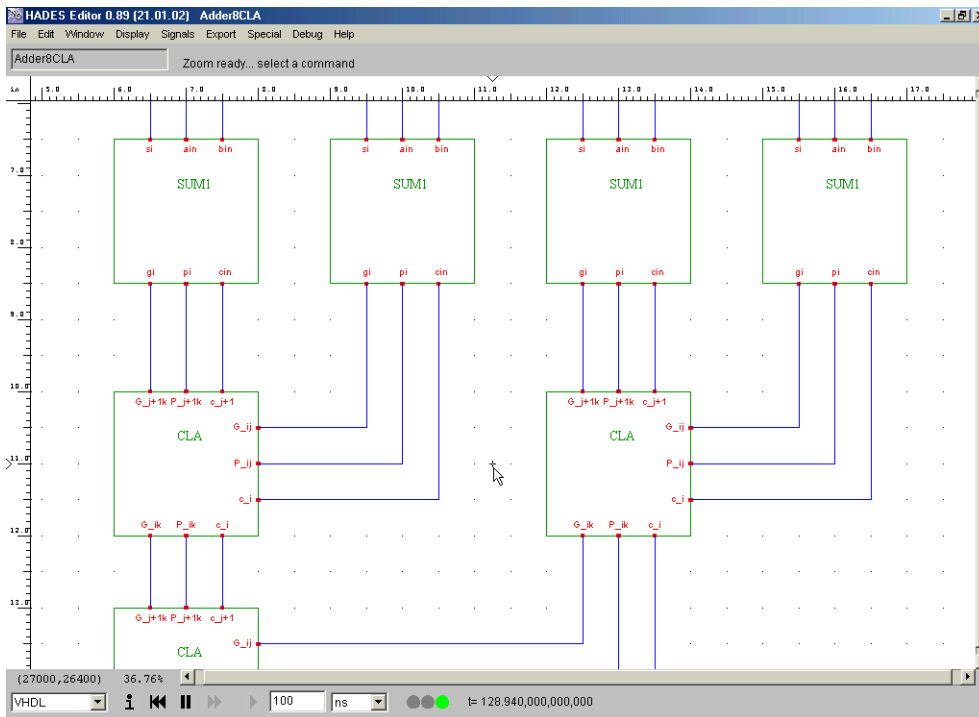
Figure 31: A region of the full CLA adder schematic, showing the connection of the adder blocks (on top) and the cascade of CLA blocks (below). The figure also shows the custom designed block symbols and the 2D placement on the schematic.

Ipin switches and *Opin* displays. (Technically, the symbol creation algorithm creates ports for all simulation components that implement the *hades.models.IpinConnector* and *OpinConnector* interfaces.) The port positions on the default symbol correspond to the relative positions of the *Ipin* and *Opin* components in the circuit schematic. After finishing the *sum.hds* design and creating a the *sum.sym* symbol, the next step is to create the carry lookahead block shown in figure 30. The circuit consists of two AOI21 gates, one AND gate, and two inverters. The input and output names, like *cin_i*, *Pin_ij* or *cout_i* indicate the function and the cascading of the corresponding signals. Again, save the design as *cla.hds* and afterwards select *create symbol* command to create the *cla.sym* symbol.

If necessary, you can edit the symbol file with a text editor, to update or change the symbol representation. Unfortunately, the planned symbol editor is not available yet. The default layout of an automatically created symbol uses the standard left-to-right dataflow direction, with all inputs on the left side and all outputs on the right. This simple layout is not optimal for the CLA circuit, because a two-dimensional layout shows the underlying tree-structure much better. Therefore, the symbol files shown in figure 31 were actually hand edited.   *editing symbols*

You are now ready to design the complete eight-bit adder circuit. First, select the *Edit ▷ New* menu command to start with a new, blank design. To create a subdesign component, move the mouse to an appropriate position and select *popup ▷ create ▷ create subdesign*. Shortly, the file selector dialog will appear. Change to the correct subdirectory, if necessary, then enter the name of the subdesign (*sum.hds* or *cla.hds*) and select the *OK* button. The Hades editor will now load the subdesign and its symbol, and it will place the symbol at the selected position. Each subdesign is labeled with an unique instance name which can be changed via the *popup ▷ name* item from the popup menu. To create the remaining adder and carry lookahead block subdesigns, you can either repeat the above steps, or use the *popup ▷ copy* command. Position the subdesigns with the *move* command to find a visually appealing layout. Finally, create *Ipin* and *Opin* components (or the hex-switches and displays) to drive the subdesigns.   *creating subdesigns*

Despite the additional steps like symbol creation, using the hierarchical design style will already be much faster than drawing a *flat* design even in this small example. The advantages of hierarchical designs become much more noticeable for larger projects. Note that you can   *hierarchical or flat?*

easily repeat the process by creating additional levels of hierarchy, for example by combining four 1-bit adders to one 4-bit adder, four 4-bit adders to a 16-bit adder, etc. 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.

Here, again, are the steps to build a hierarchical circuit in *bottom up* order:

1. Design the subdesign and save it (*subdesign.hds*).

2. Execute *Edit ▷ Create Symbol* the menu command.

3. Design the toplevel design.

4. Exceute the *popup ▷ create ▷ create subdesign* command and select the subdesign file name (*subdesign.hds*) to create an instance of the subdesign.

Because the editor can only refer to available components, a pure top-down design style is impossible in Hades (or any other component based simulation framework). Naturally, you can use empty designs as placeholders:

1. Use the editor to create an otherwise empty design with just the I/O-components for your subdesign. For example, create *Ipin* and *Opin* components for all toplevel connections of your subdesign.

2. Execute *Edit ▷ Generate Symbol* from the menu to generate a symbol for the (still empty) subdesign.

3. Include the subdesign into your toplevel design and edit the toplevel design.

4. Fill in the remaining components of the subdesign(s).

5.2   Editor bindkeys

efficient editing
To improve the usability of the schematics editor, Hades provides several standard *bindkeys* as shortcuts for the most important editing and drawing functions. Similar to the popup-menu behaviour, all commands are automatically applied to the object nearest to the mouse position, without the need to expcicitly select an object. 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, many of which require both hands on the keyboard. Therefore, the Hades editor uses only a few *ALT+X* combinations, while most shortcuts are directly available via simple key presses. Figure 32 shows the table of the available predefined bindkeys. In the current version of Hades, the mapping is hard-coded in class *hades.gui.KeyHandler* and can not be changed at runtime.

5.3   Printing and fig2dev export

printing issues
While the Java runtime environment includes a printing function, the quality of the resulting output is still far from optimal and depends on the Java version being used. Older Java virtual machines up to 1.1.x only support the base AWT graphics calls and use a fixed graphics resolution of 72 dpi. The resulting output should be readable, but several objects (like the curved symbols used for OR and XOR gates) will look very poor. The Hades editor automatically detects, when the newer and higher quality Java2D graphics library is available. If so, the Java2D printing system will be used, which theoretically results in better output quality. In some cases, however, the printing may still produce bad output or even crash, due to Java2D bugs and printer driver issues.

Key	Function
0	set a signal to '0' level
1	set a signal to '1' level
c	copy a component
e	edit a component's properties
f	zoom fit
F	zoom full (100%)
g	toggle glow-mode (all signals)
h	toggle glow-mode (one signal)
I	create an Ipin component
m	move a component
M	move a signal vertex
N	change a signal name
n	change a component name
o	move a signal vertex
O	create an Opin component
p	add a probe to a signal
P	delete a probe from a signal
r	force a redraw
U	undo the last command
v	insert a new vertex into a wire
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
y	zoom out (71%)
z	zoom out, centered (71%)
Z	zoom in, centered (141%)
BSP, DEL	delete a component/wire
ESC	interrupt and cancel any command

Figure 32: Important bindkeys in Hades, sorted alphabetically

To test the Java native printing, simply select the *File ▷ Print* menu command from the editor menu bar. This will open the platform specific print options dialog, which allows to select the printer, paper size and orientation, etc. Click the *OK* button in the dialog window to start the printing.

The recommended way to print Hades schematics or to convert them into other graphical formats is via the FIG-format export functions. The schematics editor relies on the graphical objects provided by the *jfig* graphics editor class libraries. For example, the graphical *symbols* for logic gates are implemented as a subclass of the *FigCompound* class, which in turn collects several individual graphical objects like polylines, rectangles, or labels. See section 9.3 on page 86 for an overview over the available graphics object classes. While several Hades graphics objects add support for animation, it is very easy to convert Hades schematics into FIG graphics files, and you can use several external tools to further process the FIG files. The most important of these tools is probably the *fig2dev* converter program included in the *transfig* program package. As the name indicates, the *fig2dev* program reads a FIG file and translates this into one of several important output formats, including Postscript, Encapsulated Postscript, and PDF. Other options are LaTeX drawing commands, EPIC commands, or the HP-GL language used to control plotter devices.

FIG format, transfig

On most Unix and Linux machines, the *fig2dev* program will usually be available as part of the default installation. The most recent version can be downloaded from *ftp.x.org/contrib/applications/drawing_tools/transfig*, if necessary. A pre-compiled binary for Windows is available for download on the *jfig* homepage, *tech-www.informatik.uni-hamburg.de*. Run the *fig2dev -help* command to list which output languages and options are supported by your version of fig2dev. For printing, you will probably select either Postscript *-L ps* or PDF *-L pdf* format. For example, the following command converts the *dlatch.fig* FIG file into encapsulated Postscript, scaling to 33% of the original size:

fig2dev options

```
fig2dev -help
fig2dev -L eps -m 0.33 dlatch.fig > dlatch.ps
```

export options dialog Several export options are directly available via the Hades editor user interface. Select the *Export ▷ Settings* menu command to open the export options dialog window. This window includes user interface components to select the output format and several options for the fig2dev export. Usually you would fill in the options from top to bottom.

color options First, select the output language and the color options. The three settings control the conversion of the Hades internal graphics to the intermediate FIG format file. The full color option exports the current schematics without any color modification. The black and white option substitutes the all colors used for component symbols and signals (even in glow-mode) with black. This improves the image quality when using a b/w laser printer, where the default raster algorithm often result in too bright and low contrast for the full color schematics. The third option keeps the colors only for the input and output components (*Ipins* and *Opins*), while all other components and signals are rendered in black and white. This setting is sometimes useful to produce a high-contrast b/w output which still allows to read input and output values.

size, orientation The remaining export options control the output page orientation, size, and offset. Use the *A4, landscape, centered* button to rescale the current schematics to A4 paper size. Next, type in a filename or click the *browse* and use the file dialog to select the output filename. Finally, click the *export now* button to actually start the export process.

FIG editing Note that the output filename does not change when you load a new design or press the *export now* button many times. Always make sure to select a new filename unless you actually want to overwrite a previously generated output file! The editor does not delete the intermediate FIG file created as the first step of the export process. This allows you to use a FIG editor like *xfig* or *jfig* to edit the graphics file, for example to include additional annotation, to highlight parts of the schematics, to change colors, or fonts, etc.

screenshots It is also possible to use *fig2dev* as a filter to generate an output in one of several well-known image formats, including PPM, PCX, PNG, TIFF, and GIF. Please check your version of *fig2dev* to see which image formats are supported. Another way to generate a screenshot is the editor *Export ▷ GIF* menu command which translates the current object canvas 1:1 into a GIF format image file. However, this command will only work if you have the required *GIFEncoder* classes installed on your system (and included in the classpath setup). Due to licensing issues around the GIF format, it is impossible to include the *GIFEncoder* classes in the *hades.jar* archive.

5.4 VHDL export

VHDLWriter As the main focus of Hades is on simulation, it does not include logic synthesis functions. Therefore, you may want to export your circuits to another design system, for example to realize your design as an FPGA prototype. To this end, the package *hades.utils.vhdl* includes a few utility classes that allow converting Hades design files into VHDL descriptions. The main class is *hades.utils.vhdl.VHDLWriter*, which reads a Hades design and writes an equivalent VHDL description including the full design hierarchy and structural (netlist) information. Naturally, due to the complexity involved, the automatic conversion of Java program code (as used by Hades simulation components into equivalent VHDL code is impossible. However, the export generates behavioral architectures for almost all gate-level components, which can then be used for both VHDL simulation and logic synthesis.

At the moment, there is no user interface for *VHDLWriter*. Instead, you have to use the shell (or a script) to call the export. The export process generates one VHDL file per Hades design and subdesign and one behavioral file per simulation component used. Therefore, the *VHDLWriter* expects the name of the VHDL working directory as its first argument, and the name of the (top-level) Hades design file as its second argument:

```
# Unix csh/tcsh example command
# VhdlWriter <directory> <designname> [<classmapfile>]
```

```
#
cd /home/joe/hades/examples
setenv CLASSPATH /home/joe/hades/hades.jar
java hades.utils.vhdl.VHDLWriter /tmp/dlatch dlatch.hds
```

Signal and component names in Hades designs may be of arbitrary length and can use the
full Unicode character set. The external name encoding in the *.hds* design files uses the
\uxxxx escape notation to represent such names in ASCII compatible files. When nec-
essary, the utility class *hades.utils.NameMangler* can be used to encode or decode such
names. On the other hand, names in VHDL are restricted to a subset of ASCII, are case-
insensitive, length-limited, and certain character combinations (like a double underscore) are
forbidden. Therefore, the conversion of Hades designs often requires a complex renaming
of signal, component, and design names into valid and unique VHDL names. The *VHDL-
Writer* class uses a built-in algorithm to this end. The name mangling algorithm knows the
VHDL language keywords and tries to detect and avoid name collisions if possibly, but the
name mangling may result in very long VHDL names. See the class documentation for class
hades.utils.vhdl.VHDLNameMangler for details.

name mangling

Unless you want to write your own VHDL packages with gate-level simulation models,
you may need to map the Hades component names to the names of the equivalent simula-
tion models from your favorite VHDL library. This requires a mapping from Java class-
names like *hades.models.gates.Nand3* or *hades.models.flipflops.Dffr* to VHDL names like
ecpd10.NAND3 or *cmos7x.DFFR*, but possibly also a mapping from Hades subdesign names
to VHDL package and entity names. Both mappings can be specified in a text file, which
is then added as the third command line parameter to *VHDLWriter*. The following lines
are taken from the example mapping file, */hades/utils/vhdl/vhdl-classmap.txt* included in the
hades.jar archive:

library mapping

```
# example Java classname to VHDL entity name mapping file.
# Note that the Java class names do not carry an extension.
#
hades.models.gates.And2        gate_and2
hades.models.gates.And3        gate_and3
hades.models.gates.Inv         inverter
hades.models.gates.InvSmall    inverter_small
hades.models.gates.Nand2       gate_nand2
#
# and so on, e.g. some flipflops
#
hades.models.flipflops.Dffr    dffr
hades.models.flipflops.Latchr  latchr
#
# or hades.models.io components:
#
hades.models.io.ClockGen       CLOCKGEN
hades.models.io.HexDisplay     hex_display
hades.models.io.LED            led
#
# use the following to re-map Hades designs/subdesigns.
# note that we map a file name (with .hds extension)
# to VHDL names:
#
examples/simple/dlatch.hds              examples_dlatch
/hades/examples/dcf77/Shifter59.hds  shifter_59
```

During the export process, the *VHDLModelFactory* class is used to actually generate VHDL
behavioural descriptions for each of the simulation components. Currently, *VHDLModelFac-
tory* supports the basic gates with up to four inputs (AND2..XOR), complex gates with up two
six inputs (AOI21..OAI33), and a few flipflops (DFF, DFFR, Latch, LatchR). Also included

VHDLModelFactory

are the *VCC*, *GND*, and *Pullup* connections, the configurable *ClockGen* clock generator, and the *PowerOnReset* reset pulse generator from the *hades.models.io* package. Naturally, *Ipin* switches and *Opin* outputs are converted to the external interface in the design entity declaration. For other simulation components that are not implemented in *VHDLModelFactory*, a valid VHDL entity declaration is automatically generated from the Hades *SimObject* port information, and an empty VHDL architecture is written.

For example, the following VHDL description is created for a *hades.models.gates.Nand2*:

```
-- VHDL for Nand2: /D-latch/i3

library IEEE;
use IEEE.std_logic_1164.all;

entity hades_models_gates_Nand2 is
  port (
        Y :         out  std_logic_1164;
        A :         in   std_logic_1164;
        B :         in   std_logic_1164
  );
end hades_models_gates_Nand2;

architecture SIMPLE of hades_models_gates_Nand2 is
begin
  Y <= not (A and B);
end SIMPLE;

configuration cfg_hades_models_gates_Nand2
of hades_models_gates_Nand2 is
  for SIMPLE
  end for;
end cfg_hades_models_gates_Nand2;
```

VHDLExportable   To include a VHDL export option into your own Java written simulation models, you have to implement the *hades.utils.vhdl.VHDLExportable* interface and implement its methods like *writeEntity()*, *writeArchitecture()* etc. During the VHDL export, *VHDLModelFactory* will check whether a simulation component implements *VHDLExportable* and call the corresponding methods to generate the VHDL instead of using its default implementation.

6 Waveforms

Signal *waveforms*, or the plot of signal values as a function of simulation time, are an essential tool for the understanding and debugging of digital circuits. Most importantly, waveforms allow tracing and analysing the timing of signal changes and their dependencies during the simulation. However, it is generally not practical to record the signal changes for all signals, due to the sheer amount of generated data. Therefore, waveform data is only kept for signals explicitly marked by so-called *probes*.

waveforms: signals vs. time

Section 6.1 first describes the overall concept of signal probes and section 6.2 lists the different types of waveform data currently supported. The user interface and basic operation of the waveform viewer is desribed in section 6.3, while section 6.4 explains how to search the waveform traces for specific events and values, and section 6.5 sketches how to save and later reopen waveform data. Next, section 6.6 lists the predefined bindkeys of the waveform-viewer. Finally, section 6.7 shows how to use scripting functions to automatically activate probes for signals and to control the waveform viewer.

6.1 Probes and the waveform viewer

While the Hades simulator needs to keep track of the most recent value of each signal to propagate this value to all components connected to a signal, the simulator normally does not store any previous values. The obvious reason for this behaviour is the enormous amount of data generated during a simulation run. On a current workstation, the simulation may process up to a few million events per second, where each event might need a dozen bytes (or more) to represent it. This translates into memory requirements of several MBytes per second to store the full simulation data, which is simply not practical in most cases.

Therefore, you have to explicitly mark each signal that you want to observe by adding a *probe* to it. As usually only a few signals are selected with probes, this greatly reduces the amount of data collected by the simulator. Internally, adding a probe to a signal automatically creates a corresponding *waveform* object, which is then used to record all subsequent changes on that signal. The relation between signal and waveform is unidirectional. While a signal has a reference to its waveform, the waveform has no reference back to the signal. This architecture allows to save waveform data to a file and to restore and browse the waveforms later without the need to also fully restore the simulator.

probes

As explained above, the Java runtime has to store all event data generated by the simulator for all probed signals. Obviously, this is no problem in small circuits, where input events are generated by interactive switches every few seconds, but a single probe on the output signal of clock-generator component set to 1 MHz will consume about 12 MBytes of memory per second. In such situations, the default upper memory limit of about 64..256 MBytes enforced by the Java runtime can be consumed very quickly, and you should consider to start the Java virtual machine with an increased value of the memory limit. For example, on most current Java virtual machines, the -Xmx512m command line option increases the upper memory limit to 512 MBytes, e.g.:

memory usage and memory limit

```
java -Xmx512m hades.gui.Editor -file clock.hds
```

Naturally, even with an increased upper limit, all available memory will be consumed sooner or later when probing signals with lots of activity during a long simulation. To avoid crashing, the waveform viewer will automatically delete the oldest part of waveform data whenever it detects an out-of-memory situation. This means that the waveform viewer keeps as much recent event history as possible, but earlier simulation traces are lost.

purging

To avoid the purging of old waveform data, you can run the simulator for specified time intervals via the *run for* option and command instead of running continuously in interactive mode. That way the simulator will stop one the specified time interval is over, and you can clear the waveform data manually after analysing it.
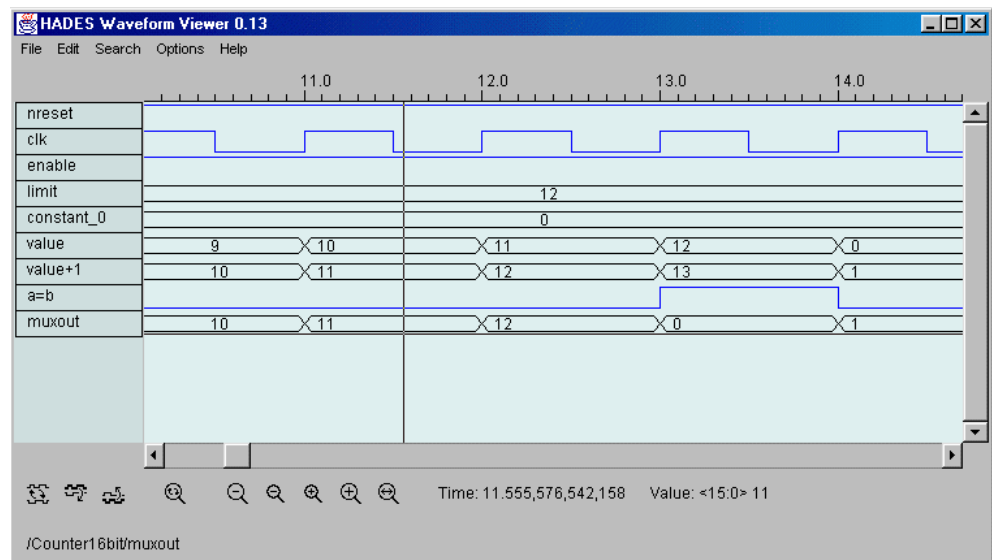
Figure 33: Screenshot of the Hades waveform viewer, showing the *StdLogicVector* wave-
forms for the RTLIB 16-bit counter circuit presented in figure 2.5 on page 10.

6.2   Waveform types

Naturally, the waveform representation for signals of different types should be optimized for
that signal type. For example, a waveform for a *std_logic* value should graphically represent
the nine logic levels of *std_logic*, while a waveform for a bus carrying an integer value should
indicate this integer value. Also, different search options corresponding to the signal type
should be provided. In Hades, this is achieved by a class hierarchy consisting of an abstract
base class *hades.styx.Waveform* and separate subclasses of *Waveform* for individual signal
types. Currently, Hades provides four waveform subclasses for the following signal types:

- *WaveInteger*, a waveform to store and render integer values
- *WaveStdLogic1164*, used for *std_logic* values
- *WaveStdLogicVector*, for *RTLIB* buses of type *hades.models.StdLogicVector*.
- *WaveString*, a waveform which stores string values.

The screenshot shown in figure 33 demonstrates the waveforms for both scalar *std_logic* sig-
nals and *std_logic_vector* buses. The waveforms in the example are taken from a simulation
of the 16-bit counter presented in section 2.5 on page 10.

Should you want to support a new waveform type, you can just write a new subclass of *Wave-
form* with corresponding methods to store, search, and paint the actual payload of the signal.
Unfortunately, the current implementation of *WaveformViewer* uses a hardcoded mapping of
signal types to waveform classes instead of the factory design pattern. To add a new wave-
form type, you will also have to edit and recompile class *hades.styx.WavwformViewer*.

6.3   Using the waveform viewer

window layout   A screenshot of the Hades waveform viewer is shown in figure 34. The main window uses a
conventional „waveform viewer" layout with a menu bar, the time panel at the top, the stack
of waveforms with scrollbars, and a set of control buttons. At the bottom of the window is
the status message panel. A cross-hatch cursor is used in the main waveform canvas, which
helps to compare the timing relationship between waveforms across the screen.

waveform names   The name panel at the left part of the main window displays the names of the waveforms.
When a waveform is first created, its name is set form the full name of the corrsponding sig-
nal. Starting from the root of the design hierarchy (/), the full name lists the instance names of
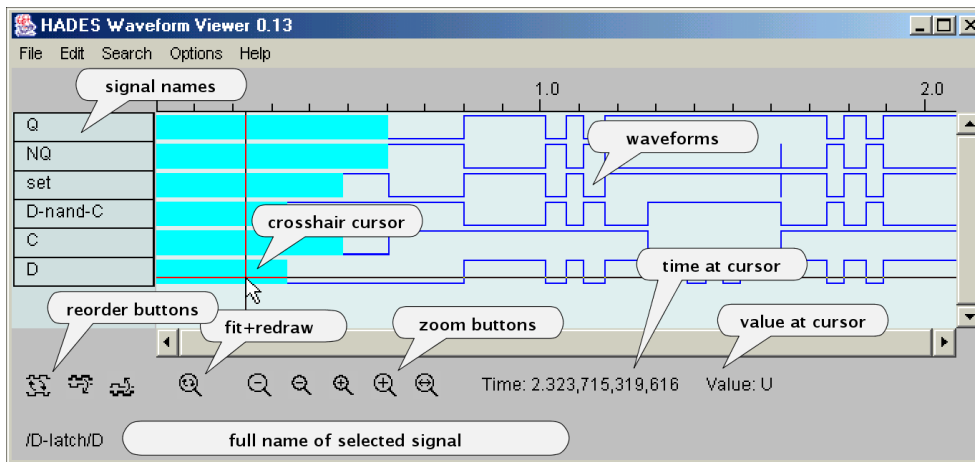
Figure 34: Annotated screenshot of the Hades waveform viewer window. The waves are for the D-latch circuit from figure 28. The main panel shows the waveform names, the waveform data, and the time scale. The status panel shows the value and time corresponding to the current mouse position (or the full signal name). Use the control buttons to select zooming and panning, and to re-order the waveforms.

any intermediate subdesigns and finally the signal name inside the leaf subdesign. Originally written long before Swing with its was available, the size of the name panel is fixed, and only the last part of the full waveform name is displayed in it. For example, short name *cout* would be displayed for a signal with the full path */cpu/alu/adder8.3/adder1.1/cout*. However, the full (hierarchical) name of the waveform at the current mouse position is displayed in the status message panel on the bottom of the waveform viewer window.

When you build a new circuit in the Hades editor, default names are created for each new simulation component and signal, which consist of an initial string followed by a number to ensure that names are unique. For signals, the default names are *n0*, *n1*, *n2*, etc., and those names will also show up in the waveform viewer. It is therefore good practice to select meaningful names for all signals that you want to trace, before adding probes to those signals. Use the *name signal* operation described in section 4.7 on page 35 to change the signal names. If necessary, e.g. for documentation purposes, you can use scripting functions to directly access and change the name of a waveform.

settting signal names is important!

In order to provide the best simulator performance, the Hades framework will not automatically update the waveform viewer display during a running simulation. While Java graphics operations are pretty fast on most platforms today, a single repaint can still take several seconds, when large amounts of waveform data are involved. Naturally, the waveform viewer will repaint only those parts of the waveform traces that are visible at the current zoom factor, but it cannot (yet) optimize away drawing operations from the visible parts of the waveforms. For example, doing a zoom-fit on waveform traces with a million events will result in as many drawing operations, and the whole user-interface will be inactive during the repainting. Therefore, you have to request a waveform repaint explicitly via a call to the *WaveformViewer.redraw()* method or by selection the *Edit ▷ Update Waves* command from the menu. Naturally, all commands that scroll through the waveform data or change the current waveform zoom factor also result in a redraw.

no automatic repainting

As usual, the scrollbars on the waveform viewer window are used to navigate through the waveform data. The vertical scrollbar allows browsing through the list of active waveforms. The horizontal scrollbar changes the time range visible in the viewer. While the leftmost position corresponds to the constant time $t = 0.0$, the rightmost position of the scrollbars changes during a simulation to reflect the current simulation time. This maximum value, however, is only updated when a user input event occurs.

scrolling

Instead of dragging the scrollbars with the mouse, you can also navigate using the *cursor-keys*. This also allows to keep the mouse at a fixed position, in order to compare the value of multiple waveforms which are not visible on one screen.

cursor keys

Use the *zoom fit* button to update the current waveform viewer transformation so as to make the whole waveform data visible. This is the fourth button from the left on the waveform viewer control panel (the single button marked with the the circular arrows). Clicking the *zoom fit* button often is the best way to update the waveforms. However, once the amount of waveform data gets large, using the zoom fit command may become (very) slow, as the operation results in a redraw of *all* waveform data.

zoom in, zoom out   To change the zoom-factor, simply press one of the five *Zoom*-buttons on the waveform viewer control panel. From left to right, these buttons activate the *zoom out 1:4*, *zoom out 1:2*, *zoom in by 2:1*, *zoom in by 4:1*, and the *zoom to selection* functions. The first four of those buttons work by changing the zoom factor of the waveform viewer window, while the selected simulation time (in the center of the window) remains constant.

zoom region   To select the time interval for a zoom operation interactively, first move the mouse to the start time of the desired interval and than drag the mouse to the end time. The waveform viewer will highlight the selected interval and the status panel indicates the duration of the selected time range. Finally, click on the *zoom to selection* button (the rightmost control button) to execute the operation.

time scales   Note that an interactive simulation may use events at very different time scales. For example, the default gate delays of the basic gates from the *hades.models.gates* package are a few nanoseconds, while user-generated events occur in the range of seconds. The simulator has no problem with this, but the very different time scales imply that you may have to zoom into the waveforms by a very large factor before details of the circuit timing become visible. For example, you might have to click the *zoom in 2:1* buttons up to 30 times to change from the range of seconds to the gates nanosecond range (because $2^{30} \approx 1^9$).

zoom auto   To zoom into an interesting range of waveform data, you first move the crosshair cursor to the center of the corresponding time range, and then select the zoom-auto function (type the key *a*). This function automatically searches for the nearest events just before and just after the current cursor position, and recalculates the zoom-factor so that those events are a few pixels apart. When different timescales are involved, this function automatically streches the zoom so that subsequent events (the small timescale) are clearly separated. You can then use the zoom-in and zoom-out functions to fine-tune the zoom-factor.

6.4   Searching waveform data

Often, it is much more efficient to search for certain waveform events than to just browse through the data manually. For example, you may want to warp directly to the next or previous event on a certain signal, or to look at the next rising edge of a signal, without changing the zoom factor and without having to scroll tediously through idle periods of that signal.

Our initial user interface concept for those seearch operations was based on context-sensitive popup menus on the individual waveforms, with the search base time specified by the mouse position when activating the popup menu. However, most of the originally planned search functions were never fully implemented, because the popup-menu approach turned out to be less user-friendly then expected. For example, having to re-invoke a menu often is quite cumbersome when trying to browse a large data set.

The current version of the waveform viewer offers only a minimal set of functions via the main *Search* menu, and uses *bindkeys* (*accelerators*) for everything else. This means that you have to learn those bindkeys first, but most operations can be selected much quicker than via menus. A summary of the bindkeys is listed in section 6.6 on page 56.

search base time   All search operations start at the *search base time*, and every successful search operation also automatically updates the search base time. This makes it possible to quickly browse through the waveform data by just repeating a search operation. The search base time is initialized to $t_{\text{search}} = 0$. To chage its current value to a designated numerical value, first select *menu ▷ search ▷ set base time* from the menu, and then type in the corresponding value as a decimal number (without units) into the dialog window. Alternatively, scroll the waveform window and position the crosshair cursor at the designated time, then type the *S* (shift+s) bindkey to update the search base time.

To provide visual feedback of search operations, the search base time is highlighted by the *search marker* *search marker*, a red vertical line, after each (successful) search operation. Use the corresponding commands, *menu ▷ search ▷ show search marker* to enable the marker (which also autoscrolls to the corresponding time) and *hide search marker* or type *R* to disable the marker.

The global search operations can be used to step through subsequent events across all wave- *next event* forms, starting from the current search base time. Select *menu ▷ search ▷ next event* or *previ-* *previous event* *ous event* from the main menu, or type the bindkeys *N* and *P* respectively. Should multiple events occur at exactly the same time, duplicates are disregarded, and only one of those events is considered during searching.

You can also search for the next or previous events on a selected waveform, instead of searching across all waveforms. Just position the crosshair cursor on the designated waveform and then type the *n* or *p* keys to search the next or previous event on that waveform.

Searching specific values

To search for specific values in the waveform data, you first specify the designated search pat- *specify search value* tern as a string. Either select *menu ▷ search ▷ set search pattern* from the main menu, or type the *V* (shift+v) bindkey to bring up the search value dialog, and then enter the value. For example, enter the (lowercase) characters *u, x, 0, 1*, etc. to search for specific logical values on a *WaveStdLogic1164* waveform, type a decimal integer to search on a *WaveInteger*, or an arbitrary string to search a *WaveString*. The search pattern for *WaveStdLogicVector* supports the different number formats supported by the *parse* method of class *StdLogicVector*, including the standard decimal notation, and the prefixes 0b and 0x for binary and hexadecimal notation. You can also use the suffixes _b, _h, for example 0xcafe or 0000101011111110_b.

Please note that the search functions don't support substrings, wildcard characters, or regular expressions for the search pattern.

Once you have set the search pattern, position the crosshair cursor over the waveform and *searching:* type the *v* bindkey to initiate the search. The search proceeds in positive direction (next *cursor position* events) when the cursor is in the right half of the waveform window, and towards earlier *selects direction* times (previous events) when the cursor is in the left half of the waveform window. This trick was chosen to avoid wasting bindkeys or a separate search-direction control, and should work well enough with a little bit of practice. After a successful search, the search marker line will appear at the middle of the waveform window.

Searching for std_logic values

As most of the gate-level simulation models in Hades are based on the *std_logic_1164* logic model, the single-bit signals of class *SignalStdLogic1164* are the most common. To help searching for the different logic values, a few extra search functions are provided for the corresponding waveform type *WaveStdLogic1164*.

Just type the (lowercase) keys *u, x, 0, 1, z, w, l* and *h* to search for the corresponding logical value on the currently selected waveform. As with the search value functions explained in the previous section, the direction of the search is taken from the current crosshair cursor position. Move the mouse to the right half of the waveform window to search for the next events (forward in time), and to the left half of the window to search for earlier events.

For example, to search for the next rising edge on a *std_logic_1164* signal, just position the crosshair cursor over the right half of waveform, and type *0* followed by *1*.

6.5   Saving and loading waveform data

To save the current set of waveforms to a file, simply select the *File ▷ Save As. . .* menu item *saving* and select the output file. This will save the waveform names and the full waveform data, *waveform data* but no references to the Hades editor or simulator. This means that the waveform data can be restored for displaying, but not for simulation.

Select *File ▷ Load. . .* to load a previously saved set of waveform data. You can now zoom and search through the data same as in the realtime mode during a simulation.

To start the waveform viewer as a standalone application, include the *hades.jar* class archive file into your CLASSPATH and run the following command in a shell:

```
java hades.styx.WaveformViewer
```

For very large waveform data files, you might have to increase the memory limit of your Java runtime via the corresponding option, for example `-Xmx512m`. After the program comes up, use the *load* menu item to open a previously stored waveform file, and study the waveforms.

6.6 Bindkeys

When studying waveforms with more than a few signals and events, controlling the waveform viewer via the menus and scrollbars can become quite time-consuming. It is usually much quicker to just type one of the predefined *bindkeys* (*accelerators*) instead of selected some function over and over again from the menu. The combination of

The following table lists the available bindkeys:

Key	Function
left	scroll waveform window left (decrease time)
right	scroll waveform window right (increase time)
up	scroll waveform window up (index)
down	scroll waveform window down (index)
f	zoom fit (whole simulation time)
y	zoom out (50%)
Y	zoom in (200%)
s	zoom into selected area (mark with mouse-dragging)
S	set search time from cursor position
R	remove (hide) search marker line
N	search next event (all waveforms)
P	search previous event (all waveforms)
n	search next event on selected waveform
p	search previous event on selected waveform
	Note: search direction for the following is controlled by the mouse position — positive when cursor is in the right half of the waveform window.
V	specify search value/pattern
v	search next/previous occurence of selected value
u	search next/previous std_logic_1164 U value
x	search next/previous std_logic_1164 X value
0	search next/previous std_logic_1164 0 value
1	search next/previous std_logic_1164 1 value
z	search next/previous std_logic_1164 Z value
w	search next/previous std_logic_1164 W value
l	search next/previous std_logic_1164 L value
h	search next/previous std_logic_1164 H value

Figure 35: Waveform viewer bindkeys

6.7   Scripting

When using waveforms to debug or demonstrate a circuit, it is often necessary to select the
same set of probes for several runs of the simulator, which can become very cumbersome via
the GUI. Instead, you may want to automate this task by scripting. For details about scripting
the Hades editor and simulator with Java or Jython see section 8 on page 71. The following
code example shows the principle of selecting probes via a Jython script:

```
# Jython/Python script for waveform initialization
...

# create an editor and load a design into the editor
#
from hades.gui import Editor
editor = Editor()
editor.doOpenDesign( "designname.hds", 1 )

# create probes for selected signals
#
design = editor.getDesign()
signalNames = [ 'D', 'C', 'i1.y', 'i0.y', 'Q', 'NQ' ]
for i in range( len( signalNames )):
   signal = design.getSignal( signalNames[i] )
   if (signal != None):
      editor.addProbeToSignal( signalNames[i] )

# create and show the waveform viewer
#
editor.doShowWaves()
waveformViewer = editor.getWaveformViewer()
waveformViewer.setBounds( 450, 50, 500, 400 )
...

# update waveform viewer display
#
waveformViewer.updateTrafo()
waveformViewer.getWaveCanvas().zoomFit()
waveformViewer.redraw()
...
```

7   Model libraries

This chapter presents an overview of the simulation models currently available in Hades, including the most frequently used models for basic and complex logic gates, flipflops, and interactive switches. Additionally, a few of the RTLIB components for register-transfer-level modelling are also decribed here. Last but not least, the PIC16 and MIPS R3000 microprocessor cores for hardware/software cosimulation are presented.

A first overview of the model library organization is given in section 7.1, while section 7.2 explains how to access the simulation components from the editor via the popup-menu or the *create by name* command. Next, section 7.3 describes Colibri, our component and library browser and its configuration.

The remaining sections in this chapter present an overview of the available simulation components:

- label to annotate your schematics 7.4
- interactive switches and displays 7.5
- power and ground connectors 7.6
- basic and complex logic gates 7.7
- flipflops 7.8
- registers 7.9
- memories including RAM and ROM 7.10
- *RTLIB* components operate on *std_logic_vector* signals 7.11
- TTL-series components 7.12
- system-level components 7.13 like serial terminals or animated displays
- PIC16 processor 7.14
- MIPS R3000 microprocessor 7.15

The default installation of Hades includes many more simulation models, for example switch-level simulation components or image-processing filters, but these are not described in this tutorial.

7.1   Model library organization

SimObject   The Java class *hades.simulator.SimObject* is the common base class of all simulation components in Hades. A *SimObject* implements the interface *hades.simulator.Simulatable* which defines the basic operations — the *elaborate()* and *evaluate()* methods — for discrete event simulation. All remaining Java classes and resources used for simulation models are organized in a Java package hierarchy starting with *hades.models*.

Design   The second most important class is *hades.models.Design*, the central abstraction of a digital circuit. A *Design* is derived from *SimObject* and provides all the functionality to organize a set of *SimObjects* (including other *Design*s as nested sub-components) into a circuit. The Hades editor always operates on the current toplevel *Design*, but a *Design* can also be opened and used without the editor user interface, for example to perform a batch-mode simulation without GUI overhead.

Class hierarchy   All other simulation models are currently organized in subpackages of *hades.models*. For example, the basic gates like AND, OR, XOR are collected in package *hades.models.gates*, the complexgates in *hades.models.complexgates*, and several flipflops in *hades.models.flipflops*. The package *hades.models.io* contains the switches and display components used for interactive simulation. A separate hierarchy for register-transfer level components starts with package *hades.models.rtlib*. These models are introduced in section 7.11 on page 67.

For a list of available simulation models including documentation see the Hades class documentation or the interactive library browser *Colibri*, described in section 7.3 below. Naturally, you can also directly list the contents of the hades.jar archive file and look for the *hades.models* directory and its subdirectories to find the available simulation models.

The Java environment currently does not enforce the package names. (If it did, *hades.models* would probably have to be called *DE.uni-hamburg.informatik.tams.hades.models*, etc.) This means that you can use arbitrary class and package names when writing your own simulation models. While packages are not write protected, you should not add new components into existing packages, however.

7.2   Accessing simulation components

As explained above in section 4.3 on page 31, the editor popup-menu provides the easiest way to add simulation components to the current design. However, in most cases only a subset of all simulation models available in the Hades software archive (`hades.jar`) will also be listed in the popup-menu. For example, we restrict the popup-menu entries to only a few basic gates, flipflops, and switches for our own undergraduate courses. The reduced range of simulation components helps the students to quickly select components, and avoids the use of more advanced components that would bypass the intended answers.

popup ▷ create

However, you can also easily customize the entries in the popup-menu. The contents of the popup-menu are not hardcoded into the software, but are generated at runtime from a configuration file. When starting the Hades editor as a standalone application, the default configuration file is called */hades/gui/PopupMenu.txt*, while */hades/gui/ViewModePopupMenu.txt* is used in view-mode or for applets. Both files are included in the *hades.jar* class archive, and you are free to copy and modify those files (for example via the *jar -u* command).

configuring the popup-menu

Alternatively, you can set the property *hades.gui.Editor.PopupMenuResource* in your Hades startup configuration file, *$home/.hadesrc*. If this property is set, the editor will parse the specified file instead of the default files to build the popup menu. This allows you to customize the popup-menu and to add or remove entries.

However, you can still access and use all simulation components available on your system directly from the editor user-interface, even if those are not listed in the popup-menu. To this end, activate the popup menu and select the menu item *popup ▷ create ▷ create by name*. This command in turn opens a dialog window which prompts you for the full Java class name of the simulation model in question. Just enter the full class name, e.g. *hades.models.gates.Nand3*, to create a new instance of the corresponding class. As the Java classloader resolves class names at runtime and searches the whole CLASSPATH, this command also lets you access your own newly written classes or any third-party simulation models. Here are a few examples of possible class names:

create by name

```
hades.models.io.ClockGen         - clock generator
hades.models.gatter.And4         - AND4 gate, german DIN symbol
hades.models.gates.Xor2          - XOR2 gate, standard US symbol
hades.models.flipflops.Dffr      - D-type flipflop with reset
hades.models.meta.Label          - label for annotations
hades.models.FigObject           - embedded FIG drawing
hades.models.rtl.ROM_64Kx8       - ROM, 64K words a 8 bit
hades.models.rtlib.arith.Adder   - n-bit adder
hades.models.rtlib.memory.RAM    - configurable RAM
hades.models.pic.Pic16C84        - PIC16C84 microcontroller
...
```

The third way to access simulation components is via the Hades component and library browser, which is the topic of the next section.
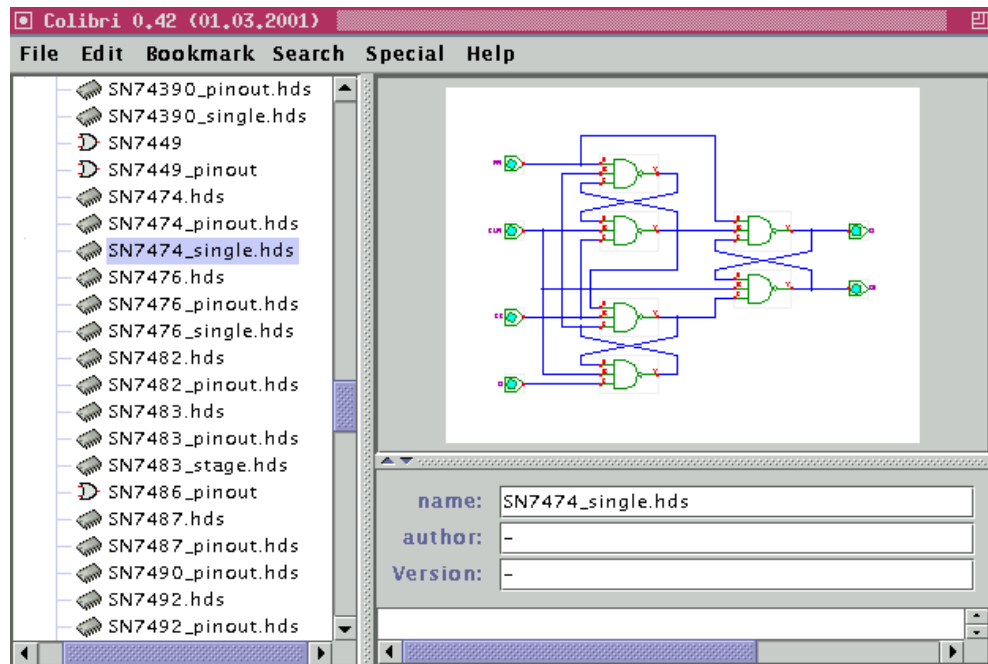
component browser

Figure 36: Screenshot of the Hades component and library browser, Colibri. The explorer-like control on the left displays all Hades simulation models, design files, and related design resources like memory data or assembler files. On the right part of its screen, the browser displays information about the selected objects.

7.3   Colibri Browser

Colibri   The *component and library browser* (or Colibri for short) allows browsing the Hades simulation model library and integrates functions for managing design files. While the standard popop-menu in the editor only includes the most frequently used simulation components, the browser gives access to the complete set of simulation models via an explorer-like tree-view of both the built-in model libraries and user-specified design directories. Figure 36 on page 60 shows a screenshot of the browser.

original concept   Originally, the first version of the browser was designed as a complete management system
and history   for standalone Java beans components, with support for beans versioning as a central function. The idea was to support smooth upgrading of simulation components, where custom classloaders were used to distinguish between different versions of the same class, and to use multiple versions concurrently in one simulation. Meta-information about simulation components was (and is) kept in separate text files, with filename extension *.clb*.

While the custom classloader concept worked fine when running the simulator as a standalone application, it also meant a lot of overhead and complexity. For example, extra care was required when comparing objects for equality, or when trying to serialize and de-serialize objects. Even worse, custom classloaders made it impossible to run the editor and simulator as an applet on older versions of Java. Even the most recent versions of the Java runtime still require extra permissions when using custom classloaders. But worst of all, it turned out we never really had different versions of the same simulation models, because it was better to just keep and use the most recent version.

User-interface overview

The current version of the browser is still based on the original ideas and even source code, but it no longer relies on custom classloaders, and it does not include the beans-versioning functions. The main user-interface consists of the browser window with the components *explorer* tree-control on the left, and the component *info panel* on the right. Just navigate the tree to browse the simulation library. See figure 36 for a screenshot of the browser.

There are three different ways to start the browser:

- in the Hades editor, select *menu ▷ edit ▷ open component browser*
- in the Hades editor, select *popup-menu ▷ create ▷ browse. . .*
- run *java hades.manager.Colibri* as the main application.

When you click on a simulation component in the tree, the browser will attempt to load and display help information about that component. If available, the browser will also show a thumbnail image of the selected simulation component or a thumbnail image of the selected subdesign. The browser also recognized a few other file types, including python scripts (*.py*) and assembly source files (*.asm*). If you select those items in the tree, the file contents will be shown in the browser info panel.

single-clicking

As noted above, the meta-information about simulation components and design files is kept in separate text-files with extension *.clb*. Therefore, you must also write the required *.clb* files when you write your own simulation components (*.class*), or when you want to access your own design files (*.hds*) as subdesigns/components from the browser.

Double-clicking an item in the tree will execute the default action bound to the corresponding type of tree node. As usual, double-clicking a directory node will expand or collapse the subtree corresponding to the directory. Double-clicking a simulation component will activate the Hades editor (or the most recently used editor window, if multiple editors are open), and initiate an *add component to current design* operation. Move the mouse to the target position of the new component and click the left mouse button to place the component, or click the right mouse button to cancel the operation. Similarly, double-clicking a Hades design file will open that design file in the current editor.

double-clicking

For any object in the browser tree, you can also activate a popup-menu whose contents will correspond to the type of the selected object. For example, the popup-menu will allow you to open a Hades design file (in the current editor, a new editor, or in view mode) or to include the design file as a subdesign in the current editor. For Python source files, you can view, edit, or execute the script, etc. Unfortunately, many menu items are still disabled, because we never managed to actually implement all the functions. Just drop us a note if you are interested in helping with this!

popup-menu

Browser setup

The current version of the Colibri browser looks for Hades simulation components and design files in several different places:

- the *built-in* simulation components included in the *hades.jar* class archive file
- optionally, additional JAR class archive files in your CLASSPATH
- optionally, all JAR class archive files in the *extension directory* of your Java virtual machine
- up to four user-specified directories on your local system

Select the *menu ▷ browser setup* menu item in the browser to bring up a dialog window that lets you select those options and the actual filenames of the four user directories. There is no use to enable the *include CLASSPATH JARs* or *include extension directory JARs* options unless you actually have custom JAR archives that contain Hades simulation models or design files.

In the dialog, you can also select the names of the four directories that should be scanned for simulation components. The contents of each directory (if it exists) are then shown as separate subtrees in the browser tree control. If one of the selected directories does not exist, the browser indicates this with a small red cross in the file-system icon. Similarly, empty directories are indiciated with a small orange cross.

Press the *apply* button in the dialog to rebuild the component tree with the currently selection options and directory names, or press the *apply and save* button to save the current options and filenames to your *$HOME/.hadesrc* configuration file and then rebuild the component tree. Finally, click the *close* button to close the browser setup window.

Note: scanning a directory for Hades design files can take a lot of time if the number of files is large or if the underlying device is slow (e.g. a network drive or a slow USB device). For ease of implementation, the updating is done in the main user-interface thread, which means that the editor and browser are blocked until the file system scan in complete. While selecting a root directory (like 'C:\' on Windows or '/' on Unix) will work, scanning and updating the file system can block the editor for minutes...

You can also customize the components that are included in the *built-in* part of the component tree. Just unpack the file *hades/.clblist.txt* from the *hades.jar* archive, edit it, and then save as file *$HOME/.hades/clblist.txt*, where *$HOME* stands for your personal home directory. On startup, the browser will check whether this special file exists, and use the file contents instead of the default component list. For each entry that you want to include in your component tree, add one line of text with the resource name of a simulation component (*.clb* files), Hades design file (*.hds* files), or other resource file (like *.py* files). You will need to restart Hades for the changes to take effect.

7.4   Label component

comments and
annotations

This passive simulation component allows to embed an arbitrary one-line text into a design schematic, for example as a comment. To create a new *Label* component, either select *popup ▷ create ▷ special ▷ label* from the popup-menu or use *create by name* and select the class name *hades.models.meta.Label*. At the moment, the Hades editor does not support a special handling of *Label* objects with direct text input. Instead, all new *Label* objects are created with default text and default text attributes. Use the *Label* property dialog to select the label text and the text attributes, including text font, font size, alignment, and color.

7.5   Interactive I/O

Ipin

The *Ipin* simulation model (class *hades.models.io.Ipin*) is an interactive switch for a single-bit signal. A normal mouse click in the center of the Ipin symbol will toggle the Ipin output value between the values 0 and 1. When the *SHIFT*-key is hold down while clicking the Ipin, the output value will step through the values 0, 1, Z (high impedance) and U (undefined) instead. The default output value of the Ipin at the start of a simulation ($t = 0$) is U; but this initial value can be changed via the property-dialog.

Note that each instance of Ipin also defines an external input for the design schematic, when this is later used as a subdesign in a hierarchical design. In this case, the name of the Ipin component (always shown on the Ipin symbol) is used as the name of the external input.
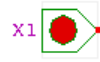
Opin

Use *Opin* instances to define an external output for a design schematic. As with Ipin, the name of the Opin component is used to specify the output port name.

The symbol of an Opin component includes a LED which uses the *glow-mode* colors to indicate the current input value.

ClockGen

This simulation model represents a clock generator with user-settable parameters for clock period (frequency), clock duty-cycle, and the initial delay. A mouse-click on the ClockGen symbol will pause or re-enable the clock generator.
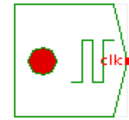
PowerOnReset

A simulation model that generates a single *active-low* reset impulse at simulation start. This is very useful to initialize flipflops and registers (like *DFFR* or *RegR*). The timing parameters (initial delay and pulse duration) can be specified in the property dialog. To generate an *acitve-high* impulse just use the combination of PowerOnReset and an inverter. A mouse-click on the PowerOnReset generator toggles the output value between 0 and 1, which allows to use the component as an interactive switch and to generate reset pulses during the simulation.

LED

This component models a simple light emitting diode for visualization. The color for the 1 input value can be selected via the property dialog, while the *glow-mode* color encoding is used for the remaining *std_logic* values.

HexSwitch

An interactive switch with 4-bit output. A mouse-click in one of the areas marked 0 to F. allows to directly select the corresponding hex output value. A mouse-click in the „–“ or „+“ areas will decrement or increment the current HexSwitch output value.
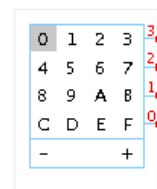
HexDisplay

A seven segment display with integrated hex decoder, to visualize 4-bit input values from 0 to F. For undefined inputs, the display will show a „8“ in the U glow-mode color.

7.6   VCC, GND, Pullup

While the interactive switches can be used to generate logic values, it is still useful to have sources for constant logic values, for example to create a constant 0 value for a carry input. This is possible via the components *VCC* (supply voltage, logic value 1) and *GND* (ground, logic value 0). The *Pullup*-resistor generates the output value H (weak 1), which can be used to model open-collector circuits or to create a default logic value for buses.

Because these simulation models are passive, they are only activated once at the beginning of the simulation. This implies that a connection of a signal to a *VCC*, *GND* or *Pullup* component during a running simulation will not update the signal value. To re-initialize the signals you will have to stop the current simulation via the *rewind*-button (<<) in the simulator control panel and the to restart the simulation via the *run*-button (>>).

7.7   Basic and complex logic gates

Naturally, Hades includes simulation models for all basic and several complex logic gates, most of which are available with 2 to 4 inputs. All of those simulation models internally use the nine-valued *std_logic* multilevel logic model [IEEE 93b]. Therefore, the gates can be used for buses and open-collector type logic, and allow to model and detect short-circuit or open-input conditions.

See the Java package *hades.models.gates* for the *basic gates* simulations models like AND, OR, NAND, NOR, XOR and a few multiplexers. The package *hades.models.complexgates* contains the simulation models for complex gates with up to three inputs (AOI21 . . . OAI33).

Additionally, the package *hades.models.gatter* contains a second set of basic gates, realized as direct subclasses of the corresponding classes in *hades.models.gates* with exactly the same behaviour. However, the classes from *hades.models.gatter* use the German DIN-style graphical symbols for the gates, which are often used in German textbooks.

To add a gate to your circuit, just select the corresponding entry from the popup-menu. For example, select *popup ▷ create ▷ gates ▷ Nand2* to add a two-input NAND-gate to your Hades design.
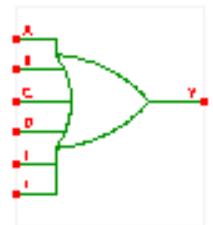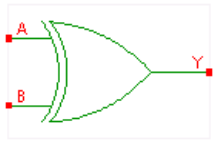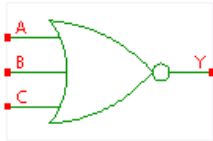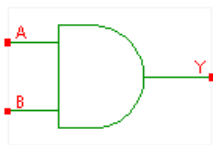
```
AND2              AND-gate, 2 inputs
AND3              AND-gate, 3 inputs
AND4              AND-gate, 4 inputs
AND4NEG2          AND-gate, 4 inputs, 2 of them inverted
...
XOR2              XOR-gate
XNOR2             XNOR-gate
...
Buffer            buffer
INV               inverter
Demux14           2-bit to 4-bit decoder
Mux21             2:1 multiplexer
Mux41             4:1 multiplexer
...
```

All gate simulation models allow to change the gate propagation delay. The default values are set for a typical 1.0μm CMOS library and range from 5. . . 10 ns. Non-inverting gates typically use the double propagation delay than the inverting gates. However, the gates do not distinguish between their different inputs. To visualize gate propagation delays, it is sometimes helpful to select propagation delays in the range of 0.5 . . . 1 sec. instead of nanoseconds. Just use *popup ▷ edit* to show the property dialog for the gate in question and enter the required value.
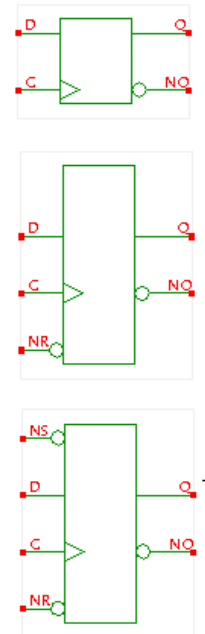
7.8   Flipflops

Apart from the simple and complex gates, Hades also includes simulation models for a variety of flipflops. These can be accessed from the *Popup ▷ Create ▷ Flipflops* menu. To enforce a proper design style, a few of the available flipflops might be disabled in the popup-menu. The symbols in the margin show (top down) the simple D-type flipflop *Dff*, the D-flipflop with asynchronous active-low reset *Dffr*, and the D-flipflop with asynchronous set and reset *Dffrs*. All of these flipflops are clocked with the *rising edge* of the clock input signal.

At the start of the simulation, all flipflops are initialized to the undefined value U. If your circuit contains feedback loops, this implies that an explicit reset or initialization sequence is required for the flipflops. Therefore, it is often better to use the *Dffr* flipflops with reset input together with a *PowerOnReset* generator instead of the simple *Dff* flipflops. See section 7.5 for a description of the *PowerOnReset*-component.

If you need a flipflop that is initialized to the output value 1, you can use the *Dffrs* flipflop and connect its *NR* reset input to a *VCC* component with a constant 1 value and the *NS* set input to a *PowerOnReset* generator.

The JK-type flipflop *Jkff* is modeled after the TTL-series 74107 type flipflop. Correspondingly, the *Jkff* flipflop is clocked with the *falling edge* of the clock signal.

7.9   Register

This simulation component models a standard n-bit register built from rising-edge triggered D-flipflops and active-low reset input. The register also works as an interactive switch; a mouse click inside a bit of the register's symbol toggles the corresponding bit through the values 0, 1, and X.

As the contents of the register are directly shown on the register's symbol, this component is useful for visualization, too. To this end, the registers bits are aligned horizontally with the MSB on the left and the LSB on the right. The data inputs are on the top and the data outputs on the bottom. The bit-width (number of bits) of the register can be changed via the register property dialog.

Shift-register

A simple variant of the register described above, which works as a shift register. On each rising edge of the clock input signal, the contents of the register are shifted one position to the left. Unlike the normal D-register, this simulation models has no parallel data inputs, but only the single *Sin* (shift-in) input. As with the D-register, the contents of each bit in the shift register can be toggled by mouse clicks into the corresponding bit position.

7.10   ROM

A *ROM* read-only memory component is often used to create custom logic functions. The Hades package *hades.models.rtl* includes simple ROM simulation components of different capacity, e.g. *ROM_64Kx8*, *ROM_1Kx8*, or *ROM_256x8*, where the first number indicates the number of words (64K, 1K, 256). Each of those simulation models uses 8-bits per word, which is the common size used by many real ROM chips.

Also, the ROM components have two additional inputs *nCS* (active low chip select) and *nOE* (active low output enable) which are required to cascade multiple chips in bus-systems. For circuits with single chips, simply tie those inputs to a constant 0 level to activate the ROM, e.g. by using a *GND* component. With both select inputs tied to ground, the ROM components behave exactly like combinatorial circuits and can be used to model the simple ROMs often used in textbooks.

The screenshot in figure 37 shows a small ROM with only 8 memory words (3 address bits) of 8 databits each. In the screenshot, the third memory word is addressed via the address
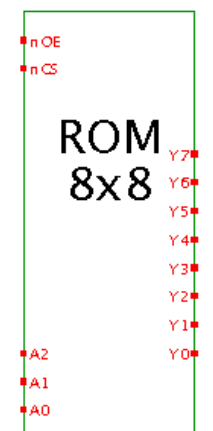
inputs `A2 A2 A0`, and the corresponding data value `0xF2` is output on the ROM data outputs `D7..D0`.

The property dialog for the ROM components includes an interactive editor which allows to save, load, merge, initialize and edit the memory contents. The central part of the editor window shows the memory contents in the usual *hex editor* form, with the address on the left and a line of data values on the right. Use the scrollbar to scroll the visible subset of the memory data. Subclasses of the editor may also implement additional functionality. For example, a ROM component used in a system built around a microcontroller might offer the option to display the memory contents in a disassembled view.

Use a mouse click or the *Cursor* keys to move the data entry cursor to a new memory address. Afterwards, you can enter a hex value at the current cursor position directly via the *0...F* keys, or increment (or decrement) the current value (including carry) using the *Shift+Cursor-Up* or *Shift+Cursor-Down* keys. Using the *Tab* (or *Shift+Tab*) key, you can move the cursor to the next (previous) memory address:

```
mouse click             - select active memory address
cursor                  - select active memory address
0 .. 9 a .. f           - data entry for a single hex value
Tab                     - move to next memory address
shift-Tab               - move to previous memory address
shfit-Cursor up/down    - increment / decrement data value
```

To save the current memory contents into a file, select the *File ▷ Save as* command from the memory editor menu bar and select a useful filename. Use *File ▷ Load* to load memory data from a file, or *File ▷ Merge* to merge the file data with the current memory contents. Because the memory data is stored in standard text files, you can also edit those files with your favorite text editor, or use a program to automatically generate memory data files.
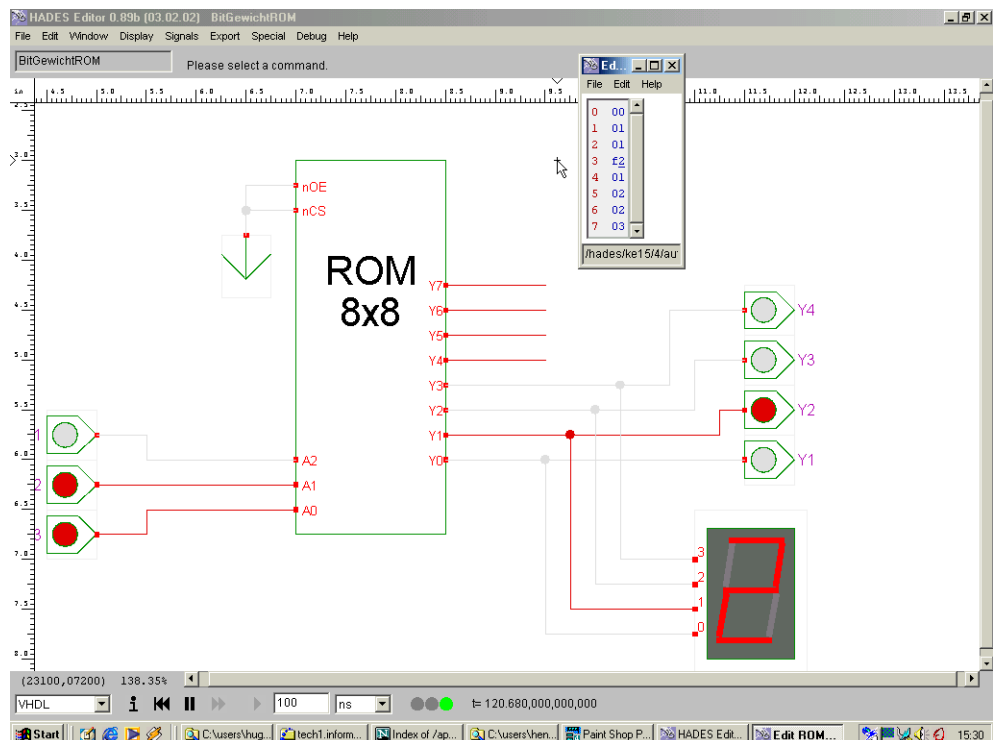

Figure 37: Example for the usage of the ROM components. The nCS and nOE inputs are tied to 0 level to active the ROM. The property dialog includes the memory editor which allows to change the memory contents.

7.11   RTLIB

The Hades framework also includes a library (called *RTLIB*) of *register-transfer level* com-   *buses*
ponents like arithmetic and logical operators, registers, register files, memories, and a
few interactive I/O components. All of these simulation models use a bus representation
(class *hades.models.StdLogicVector*) for their multi-bit inputs and outputs and the usual *Std-
Logic1164* data type for the single bit data.

The Hades graphics editor uses a special color (lilac instead of red) to indicate the bus ports
on RTLIB components. When connecting to such a port, the editor automatically creates a
bus of the corresponding bit-width. When connecting a bus to a bus port, the editor checks
that the bit-width is consistent. The bit-width of all RTLIB components can be changed via
the component's property editor in the range of 1 to 63 bits. (This limit is imposed by the
implementation in *StdLogicVector*, which uses a set of long integers to store the individual
bits). In order to avoid inconsistent circuits, the width of RTLIB-models can only be changed
as long as the component is not connected to signals.

In *glow-mode* a simple trick is used to animate and visualize the bus signals. Unfortunately, it   *glow-mode*
is obviously impractical to assign a separate color to each possible value on a bus. Therefore,
the editor calculates the integer value modulo 10 and uses this index into a fixed list of colors,
where the color is taken from the IEC-code for resistor labeling:

```
X    cyan
Z    orange

0    black
1    brown
2    red
3    orange
4    light green (instead of yellow)
5    green
6    blue
7    lilac
8    grey
9    blue-gray (instead of white)
```

Due to the large number of available RTLIB-components, only a few (if at all) are included
in the editor popup-menu. Use either the design browser of the *popup ▷ create ▷ create by
name* command from the popup-menu to create RTLIB simulation models. For example,
using *hades.models.rtlib.register.RegRE* represent a rising-edge triggered n-bit register with
asynchonous reset and data enable inputs.

Several RTLIB components use animated symbols in the graphics editor. For example, a   *rtlib animation*
register might indicate its current contents via updating a label on its symbol with the cor-
responding integer value. During simulation, all those labels and animated graphics objects
need to be updated, which requires a lot of processor cycles. Therefore, you may prefer to
disable the animation of RTLIB simulation models via *menu ▷ display ▷ rtlib animation* from
the editor menu bar. As the animation is stopped completely when switched off, the displayed
values may no longer coincide with the current values of the simulation itself.

IpinVector und OpinVector

The Java package *hades.models.rtlib.io* collects the interactive I/O components of RTLIB.
The most important of those are *IpinVector* and *OpinVector*, which are used to define the
external (bus-) inputs and outputs of a circuit.

IpinVector is also used as the interactive switch for bus signals. There are two ways to change
the output value of the *IpinVector*. First, you can use the property dialog to specify any numer-
ical value in the corresponding text field. The current value from the text field is committed
every time the *Apply-* or *OK*-buttons are pressed. Note that you can specify values in binary,

decimal, or hex notation. For example, you could select binary format and enter a string like 0010UXZ0HHHH_b for a 12-bit output including U, X, Z and H bits, where the trailing _b tells the parser to use binary format.

Second, the output value of an *IpinVector* can be changed interactively via mouse clicks into the *IpinVector* symbol. Each simple mouse click increments the output value, while each *Shift*-click decrements it. The combination of *Cntl*-click (left mouse button click with *Control*-key hold down) steps throught the output values XXX, UUU, and ZZZ respectively, which means a bus with all bits set to X, U, or Z values.

Note that many frequently used output values can be selected with a few input events. For example, the input sequence *Cntl-click, click* sets the output value to 0000...0, *Cntl-click, click, click, click* results in the output value 2, and *Cntl-click, Shift-click* sets all bits (output value 1111...1).
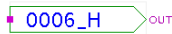
An *OpinVector* defines an external output of your circuit. The *OpinVector* symbol includes an interactive display that shows the current input value in either binary, decimal, or hex notation, as long as *menu ▷ display ▷ rtlib animation* is selected. The output format can be selected via the property dialog. Note that binary strings take a lot of screen space and may „overflow" beyond the border of the components. If necessary, select *menu ▷ edit ▷ redraw all* to redraw and update the editor canvas.

Register
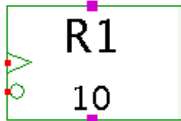
Several frequently used registers are available from the package *hades.models.rtlib.register*. For example, the *RegR* component shown in the margin models a rising-edge triggered D-type register with asynchronous active-low reset input. (Use a *PowerOnReset* component to generate a suitable reset pulse). On the *RegR* symbol, the data input is on the top and the output on the bottom, which gives the usual top-to-bottom dataflow often used for register-transfer level schematics in textbooks. If *rtlib animation* is on, the current state of the register is displayed numerically on the register symbol. Again, it is possible to select binary, decimal, or hex-format via the property dialog.

Other RTLIB registers include D-type registers with and without reset and enable, latches, a counter, a shift-register, and a linear-feedback shift-register.

Multiplexer
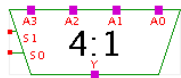
Standard bus multiplexers (2:1 and 4:1) are found in classes *hades.models.rtlib.muxes.Mux21* and *Mux41*. The control inputs *S1* and *S0* are on the left side of the mux symbol, while the data inputs *A3* to *A0* are on the top and the output on the bottom.

Arithmetic and logical operators
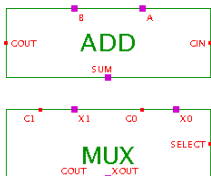
The Hades RTLIB also contains a variety of arithmetical and logical operators. Adders, incrementer and decrementer, ALUs, user-defined ALUs, and shifters are in the package *hades.models.rtlib.arith*, while *hades.models.rtlib.logic* hosts several logical operators, and *hades.models.rtlib.compare* includes the standard comparison operators.

The examples in the margin show an 8-bit adder and a 8-plus-1 bit multiplexer. An example screenshot of the user-defined ALU is shown in figure 8 on page 11.

7.12   TTL 74xx series models

The package *hades.models.ttl74* collects several components of the TTL 74xx series of MSI components. A few of the components are written as behavioural models in Java code, for example the 7449 seven-segment decoder. However, most of the models are implemented as Hades subdesigns, built from basic gates and flipflops. To add these components to a Hades schematic, select the *popup ▷ create ▷ create subdesign* command. Because the schematics can also be opened as designs (instead of being used as subdesigns), the 74xx series library offers a sizeable collection of Hades design examples.

For several components, both a functional and a pinout variant of the components symbol are available. The pinout symbols can be used whenever a direct correspondence with the real devices is desirable; e.g. when using Hades to design and simulate circuits that will afterwards be realized on a lab prototype.

A few of the basic gate pinout models, like the *SN7400_pinout* or *SN7432_pinout* components, also require that the ground and power pins are connected in order to get the component working.
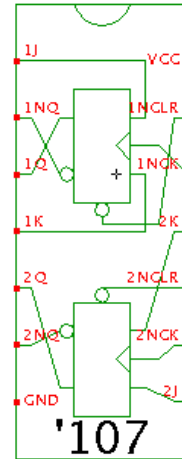
7.13   System-level components

This section lists a few *system-level* simulation models included with Hades.

- *hades.models.special.SerialTerminal*, a VT52-compatible text terminal with serial interface. The RS232 communication parameters are user-configurable (bits per word, baud rate, parity and stop bits).

- *hades.models.special.ParallelTerminal*, a VT52-terminal text terminal with parallel interface, consisting of 8 data lines and a strobe signal.

- *hades.models.special.TextLCD*, a standard ASCII liquid crystal display with up to four lines of text of 8 to 40 characters each.

- *hades.models.special.GraphicsLCD*, a graphical 128x64 pixel monochrome liquid crystal display modelled after the KS0108 controller chip.

- *hades.models.io.Iso8859Display*, a 5x7 dot-matrix LED display with integrated ASCII (ISO8859) decoder.

- *hades.models.special.Counter*. This component allows to detect and count events on *std_logic* type signals. For example, the counter can be configured to count all occurences of the X value or rising 0->1 edges.

- *hades.models.special.HazardTrigger*, which allows to detect selected hazards on a signal. For example, couple the *HazardTrigger* and a *Counter* to count the number of hazards of a given minimum duration on a clock signal.

- *hades.models.fsm.FsmWrapper*, a wrapper around a simple interactive state-machine editor. This allows to integrate state-machines into a Hades gate-level circuit.

- *hades.models.FigObject*, a wrapper that allows to integrate existing FIG-drawings into Hades schematics.

- *hades.models.dcf77.Dcf77Sender* and *Dcf77Clock*. A behavioural simulation model of the German DCF77 sender, which emits a digital signal that encodes the current local date and time. The simulated date and time values can be specified via the property dialog. The *Dcf77Clock* is the corresponding decoder.

- *hades.models.meta.AudioPlayer*, a simulation component that plays audio data from a file or URL. Note that support for audio depends on the Java virtual machine and operating system details.
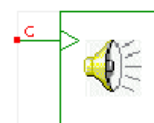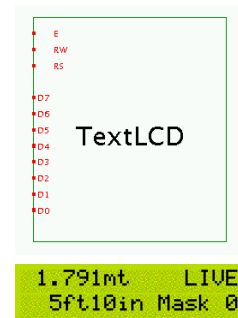
7.14   PIC 16C84 microcontroller

One of the goals of Hades was to provide efficient support for hardware/software-cosimulation of digital systems built around microprocessor cores. The *PIC16C84* component is a cycle-accurate simulation model of the PIC 16C84 microcontroller from *www.microchip.com*. This microcontroller is a complete system on a chip including the PIC16 processor, program EPROM, data RAM, data EEPROM, and programmable I/O. The package *hades.models.pic* collects all Java classes belonging to the PIC microcontrollers, including a simple batch-mode assembler and the GUI with interactive editors for program memory, data and EEPROM memory. It also hosts the description and software for a very low cost programmer for the PIC16C84 devices.

Currently, the package holds four variants of the basic microncontroller simulation model. The *Pic16C84* is the cycle-accurate simulation model which requires an external clock input, but supports the whole interrupt architecture including power-down and wakeup. The *Pic16C84Debug* model adds a few signals (not found on the real device) that allow to trace important internal register values via waveforms for better program debugging.

A faster simulation is possible with the *FastPic16C84* component, which uses an internally generated clock (and ignores events on the processor clock pin). As the processor requires two clock cycles per instruction cycle, and each clock cycle takes at least three events, this trick alone can give a speedup of up to factor 6 over the basic *Pic16C84* component.

Finally, the *SmartPic16c84* simulation model uses a cycle-based simulation algorithm and direct synchronization with the Hades simulation kernel for further speedup. This means that the processor will behave like a cycle-based full-speed simulator while still keeping in full synchronisation with the (event-driven) external circuits. A full system simulation should run at several million PIC instructions when using the *SmartPic16c84* component.

7.15   MIPS IDT R3051 core

This includes the *IDTR3051* simulation component that implements the R3051 MIPS R3000 series 32-bit microcontroller from IDT, *www.idt.com*. The model is believed to be cycle-accurate for most instructions (except multiplication and division) including I-cache and memory interface timing. The GUI for the processor also shows the five-stage processor pipeline and allows to watch the instruction execution on a typical scalar RISC-processor. The memory models included with the *IDTR3051* allow to parse and load *ELF*-format binaries into the simulation. A separate GUI is used to configure the memory ranges for the processor.

The Java classes for the *IDTR3051* processor simulation model are organized in package *hades.models.mips* and its sub-packages. For example, *hades.models.mips.instr* defines the MIPS R3000 instruction set.

8   Scripting and Stimuli

While the interactive simulation mode with its direct control and immediate feedback via the *automation*
user interface is one of the advantages of the Hades framework, the simulation and debugging
or more complex circuits can profit greatly from scripting. Also, scripting allows to execute
methods which are not accessible directly via the user interface. For example, the setup and
selection of probes to watch important signals with waveforms is a tiresome operation which
can easily be automated.

Of coure, every scripting language with a Java binding can be used to control and automate *languages*
the Hades framework, including JavaScript, EcmaScript, Tcl, or the BeanShell. However, the
following sections will concentrate on just two languages, namely Java and Jython. First, a
few code-examples explain how to control the Hades simulator and editor from user-supplied
Java classes. Second, a Jython-script is presented to show the excellent integration of Jython
code into the Java runtime environment.

8.1   Java-written scripts

The easiest way to control the Hades framework including editor and simulator is to write ad- *pure-Java*
ditional Java classes, one of which is then used as the main program. The obvious advantage
of this technique is that only one source language is used and there is no need for additional
runtime support, except possibly for the Java virtual machine CLASSPATH setup. Also, the
performance of Java code should be better than that of interpreted scripting languages. On the
down side, Java code is compiled and only very few runtime environments allow to change
Java code on-the-fly.

The following code example illustrates the basic concepts of using Java to script the Hades
simulator and editor. The new class, *RunHadesDemo*, is used to setup and create a Hades
editor and to run a simple simulation. The first code snippet presents the new class and its
main() method. As usual, a few *import* statements indicate what classes are used and help to
avoid typing the fully-qualified class names:

```
/* RunHadesDemo.java - run and control a Hades simulation
 *
 * set CLASSPATH to include "hades.jar" and "RunHadesDemo.class",
 * then start with "java RunHadesDemo"
 */
import jfig.utils.SetupManager;      // properties management
import hades.gui.*;                  // Editor and GUI stuff
import hades.models.Design;
import hades.models.Const1164;       // std_logic constants
import hades.models.io.Ipin;         // switches

public class RunHadesDemo {
  public static void main( String argv[] ) {
    ...
  }
}
```

The following few lines of code are used to initialize the simulator and editor setup from the *SetupManager*
global, user, and local properties resource-files. Naturally, properties can also be set from the
Java code itself:

```
  public static void main( String argv[] ) {
    SetupManager.loadGlobalProperties( "hades/.hadesrc" );
    SetupManager.loadUserProperties(  ".hadesrc" );
    SetupManager.loadLocalProperties( ".hadesrc" );
```

```
// we don't want the editor to start the simulator right away
SetupManager.setProperty(
  "Hades.Editor.AutoStartSimulation", "false" );
...
```

hades.gui.Editor Now we can create a new editor instance. As *hades.gui.Editor* is the toplevel GUI class in the Hades framework, this automatically creates the rest of the user-interface including the object canvas, and also the simulation kernel and a (so far empty) design. Unfortunately, the user-interface classes are managed by separate threads in the Java runtime environment and it is difficult to synchronize the GUI initialization. Therefore, the example uses a *sleep()* call to wait while the user interface is created, and then proceeds to set the editor window size and position:

```
...
// create an editor
Editor editor = new Editor();

// give the editor some time to initialize itself
try { Thread.currentThread().sleep( 2000 ); } // msec.
catch( Exception e ) {}

// specify window size and position
editor.getEditFrame().setBounds( 100, 100, 700, 500 );
...
```

opening a design The next step is to load a *Design* into the editor. The *doOpenDesign()* method accepts either a filename or a Java resource name as its first argument, while the second argument decides whether the editor should check for unsaved changes or load the new circuit unconditionally. The example code then uses the *addProbeToSignal()* method to initialize waveform probes for some signals which are identified by their names, shows the waveform window and specifies its position and size. The code also activates glow-mode for all top-level signals except for one:

```
...
// load a design
editor.doOpenDesign( "/hades/examples/simple/dlatch.hds", true );
Design design = editor.getDesign();

// add some probes and show the waveform viewer
String signalNames[] = { "D", "C", "Q", "NQ" };
for( int i=0; i < signalNames.length; i++ ) {
  editor.addProbeToSignal( signalNames[i] );
}
editor.doShowWaves();
editor.getWaveformViewer().setBounds(450,50,500,400); // x y w h

// set glow-mode globally, then disable glow-mode for one signal
editor.setGlowMode( true );
design.getSignal( "NQ" ).setGlowMode( false );
```

running the simulation At this point, we are ready to actually run the simulation. For interactive simulation it is best to just call *runForever()* to start the simulation engine. In this mode, the simulation continues forever until interrupted by the user. The example code also illustrates a basic (if cumbersome) technique to specify input stimuli. While *Ipin* components are normally used interactively or controlled implicitly from a higher-level design, a program can also create input events by calling *assign()*. This method takes a string input value and the simulation time (in seconds) and schedules the corresponding event with the simulator:

```
    // now start the simulator
    editor.getSimulator().runForever();

    // set some stimuli (or simply use the interactive switches)
    Ipin dataPin = (Ipin) design.getComponent( "D" );
    Ipin clkPin  = (Ipin) design.getComponent( "C" );

    dataPin.assign( "1", 1.0 ); // value, time
    clkPin.assign(  "0", 2.0 );
    clkPin.assign(  "1", 2.5 );
    dataPin.assign( "0", 3.5 );
    clkPin.assign(  "0", 4.0 );
    clkPin.assign(  "1", 4.000000001 );

    // finally, wait a little, then redraw the waveforms
    try { Thread.currentThread().sleep( 5000 ); }
    catch( Exception e2 ) { }

    editor.getWaveformViewer().updateTrafo();
    editor.getWaveformViewer().getWaveCanvas().zoomFit();
    editor.getWaveformViewer().redraw();

    // main thread stops here, editor runs until "Quit" selected
    ...
  } // end main()
```

Most Java compilers should be able to compile the above source file, but the required setup differs. Usually, you have to include the `hades.jar` archive as well as your working Java directory (for example, `"."`) in the Java CLASSPATH, so that the compiler can find the Hades classes. Most JVMs also allow to specify the classpath on a command line. Please check chapter 3 for details about JVM setup and classpath issues. After compilation, just call the Java virtual machine with the name of your newly created class, and the required parameters, if any. For example, on a default JDK 1.3 installation:

```
# compile and run the RunHadesDemo class,
# assuming an Unix/tcsh/JDK environment:
#
setenv CLASSPATH hades.jar:.
javac RunHadesDemo.java
java RunHadesDemo
```

8.2   Batch-mode simulation and circuit selftests

This section presents example Java code to run a Hades simulation in *batch-mode* without the user interface. To this end, the *Design* and *Simulator* classes are initialized explicitly without creating an *Editor*.

batch-mode

The example also presents the use of linear-feedback shift-registers (LFSRs) to create pseudorandom input patterns. A modified LFSR is used as a signature analysis register to collect the circuit responses to the pseudorandom input stimuli. This signature analysis technique is frequently used for automatic selftest in VLSI circuits, but is also very efficient for batch mode simulation. While the following example code does not create the user interface, figure 38 shows the typical setup with the LFSR-generator, the circuit under test, and the LFSR analysis registers.

LFSR selftest

Again, the code starts with a few *import* statements to indicate which Java classes are required by the program. Naturally, this also means that we can later write the shorter class names instead of the fully qualified names:
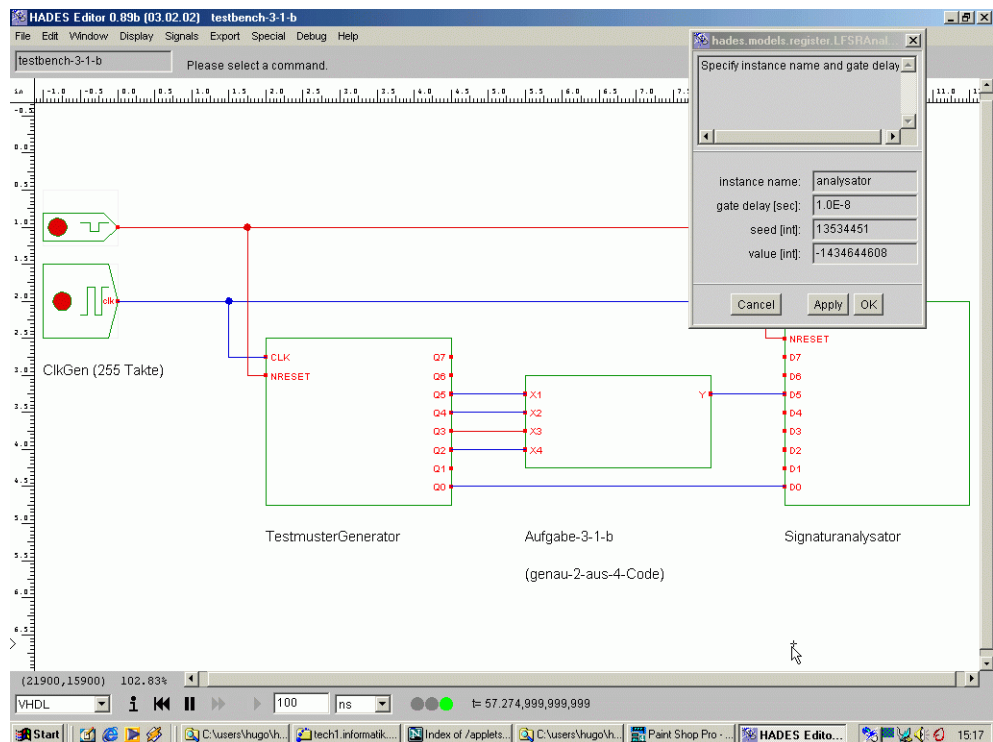
Figure 38: A testbench circuit for automatic LFSR-based signature analysis of a student's circuit. From left to right the clock generator and reset pulse generator, the LFSR generator used as test pattern generator, the students design, and the LFSR signature analysis register. The property dialog window for the analysis register displays the final signature calculated during the simulation.

```
...
import hades.models.Design;
import hades.models.register.LFSRGenerator;
import hades.models.register.LFSRAnalyzer;
import hades.utils.HexFormat;
...
```

DesignManager   The next few lines of code use the *DesignManager* class to initialize a new *Design* object from the specified Hades design file (here *testbench.hds*). Usually, the current Hades editor object is passed as the first argument to *DesignManager.getDesign()*. However, for a batch-mode simulation without GUI, it is also possible to just pass in a *null* argument. The third parameter of *getDesign()* is used to distinguish between the top-level and sub-designs. Here, *true* is used because the testbench is the top-level design. Next, a new simulation kernel is created and attached to the testbench *Design*:

```
/**
 * demonstrate batch-mode simulation and LFSR selftest.
 */
public class BatchModeSimulation {
 public static void main( String argv[] ) throws Exception {

  // load the testbench design, null=no editor, true=toplevel
  //
  String designname = "testbench.hds";
  Design design = DesignManager.
                      getDesignManager().
                       getDesign( null, designname, true );
  design.setVisible( false );
```

```
// create and setup the simulation kernel
//
SimKernel simulator = new VhdlBatchSimKernel();
design.setSimulator( simulator );
simulator.setDesign( design );
...
```

The following lines of code are used to initialize the seed values for the LFSR pattern gen-   *seed values*
erator and signature analysis registers. To obtain a reference to the simulation components,
the *getComponent()* method from class *Design* is called with the corresponding component
names ("generator", etc.). The (ugly) type casts are needed because of the strict Java typing
rules:

```
...
LFSRGenerator generator = null;
LFSRAnalyzer  analyzer  = null;

// type cast required: Design.getComponent() returns a SimObject
//
generator = (LFSRGenerator) design.getComponent( "generator" );
analyzer  = (LFSRAnalyzer) design.getComponent( "analysator" );

// set initial seed values:
//
generator.setSeed( 1234567 );
analyzer.setSeed(  9874323 );
...
```

After all the initialization steps, we are now ready to start the simulation. To start an interac-   *runFor()*
tive simulation one would usually call the simulation kernel's *run()* method, which runs for-
ever, even when no simulation events remain. Here, we call the simulation kernel's *runFor()*
method instead, to specify the end time (in seconds) for the simulation. As the simulation
kernel creates and uses its own Java thread, we need to synchronize in order to detect that the
simulation has finished. To keep the example code as short as possible, the code employs a
simple busy-wait loop. The loop just waits a few milliseconds (here 500), and then checks
again whether the simulation kernel has already reached its end time. If so, we just access the
LFSR analyzer register to retrieve the final signature from the circuit.

```
...
// setup is complete now, set the simulation end time,
// and start the simulator Thread
//
double endSimTime=  3.14; // in seconds
simulator.runFor( endSimTime );

// busy wait until the simulator is ready. This loop is simpler
// than full Thread synchronisation with the simulation thread.
//
while( simulator.getSimTime() < endSimTime ) {
  try {
    Thread.currentThread().sleep( 500 ); // 500 msec.
  }
  catch( InterruptedException e ) { /*ignore exception*/ }
}

// get final simulation data: here signature analysis
//
```

```
  long signature = analyzer.getValue(); // & 0x00000000ffffffffL;
  System.out.println("final signature is: " + signature);
  System.exit( 0 );
 } // main
}
```

8.3   Jython

scripting languages   Due to the greater flexibility and the option to develop code while running it, an interpreted environment often ideally complements the traditional compiled Java environment. At the moment, Java bindings and interactive interpreters exist for almost all well-known scripting languages. In this section, the Jython language [Jython] is chosen to demonstrate Hades scripting. As Jython, the Java-based version of the popular Python scripting language, also uses an object-oriented paradigm, and supports exceptions, it is particularly well matched to the Java environment. Also, Jython uses a very simple syntax and provides powerful libraries, including string processing and regular expression matching. The Jython runtime environment reads Jython bytecodes and compiles them to Java bytecode, resulting in good performance.

The following code assumes that the Jython interpreter is used to load and control the Hades classes. However, it is also possible to first create the Hades editor and then to start a Jython interpreter shell from the Hades editor via the *menu ▷ special ▷ create Jython shell* command.

The following script is very similar to the Java-based script from section 8.1. However, variables can be used without previous declaration, there is implicit run-time type checking, and the values *1* and *0* are used as the boolean constants. Also note the *import* statements required to access Java classes:

```
  # optional, but useful to set the global/user/local defaults
  #
  from jfig.utils import SetupManager
  SetupManager.loadGlobalProperties( "hades/hades.cnf" )
  SetupManager.loadUserProperties( "hades.cnf" )
  SetupManager.loadLocalProperties( "hades.cnf" )

  # create the editor, set window size
  #
  from hades.gui import Editor
  from hades.models import StdLogic1164
  from hades.models import Const1164
  editor = Editor()

  # wait until the GUI is initialized
  #
  from java.lang import Thread
  Thread.currentThread().sleep( 2000 )
  window = editor.getEditFrame()
  window.setBounds( 50, 50, 600, 500 ) # xywh

  # load a design file, 1=true 0=false
  #
  editor.doOpenDesign( "/hades/examples/simple/dlatch.hds", 1 )
  editor.doZoomFit()
  ...
```

syntax and indentation   Similar to the Java-example presented above on page 72, we create waveform probes for selected signals. Note the indented formatting for the *for* and *if* constructs required by the Jython/Python syntax:

```
# create the waveform viewer, add some traces
# Note the formatting required by Python
#
signalNames = [ 'D', 'C', 'i1.y', 'i0.y', 'Q', 'NQ' ]
for i in range( len( signalNames )):
  signal = design.getSignal( signalNames[i] )
  if (signal != None):
    editor.addProbeToSignal( signalNames[i] )

editor.doShowWaves()
waveformViewer = editor.getWaveformViewer()
waveformViewer.setBounds( 450, 50, 500, 400 )

# or use:
# editor.addProbesToAllSignals()
# editor.removeProbesFromAllSignals()
```

There are two ways to acutally run the script. First, it is possible to start the *jython* interpreter   *running jython*
without any options and use the *execfile()* function to read and execute the script. Alterna-
tively, it is also possible to supply the script name on the command line. In the following shell
transcript on a Unix system, *prompt* indicates the Unix shell prompt and >>> is the Jython
interpreter prompt:

```
prompt> jython
Jython 2.0 on java1.3.0rc1 (JIT: null)
Type "copyright", "credits" or "license" for more information.
>>> execfile( 'jythondemo.py' )
>>> # interactive Jython commands here, for example
>>> # note that editor is declared inside the script
>>> d = editor.getDesign()
>>> d.printComponents()
...
>>> CNTL-D  # EOF/end-of-input marker

prompt> jython jythondemo.py
... Hades runs
```

8.4   Generating simulation stimuli

One use of scripting is to specify input stimuli for a circuit during a batch-mode simulation.   *input patterns*
While older simulators typically used a special but very simple language for stimuli descrip-
tion, modern languages like VHDL allow to write stimuli in the design language itself. In
Hades, there are several ways to specify simulation inputs, some of which were already shown
in previous examples:

- use of simulation models like *ClockGen* and *PowerOnReset* to generate commonly
  used signal patterns.

- using the *StimuliGenerator* and *StimuliParser* classes to read stimuli data (in VHDL
  style) from files; see section 8.5 below.

- employing the *hades.models.io.Stimulus* simulation model to generate a vector of
  std_logic values.

- generation of pseudo-random input patterns using LFSR-registers.

- writing additional Java or Jython *SimObject* classes to generate stimuli algorithmically.
  Two typical examples are the *hades.models.dcf77.DCF77Sender* and *DCF77Clock*
  simulation models which implement the encoder and decoder (clock) for the German
  DCF77 time broadcast signal.

- calling the *assign()* or *setValueAtTime()* methods to specify a new input value for *Ipin* switches.

- direct method calls to *scheduleEvent()* to register arbitrary input events with the simulator.

- using the Jython *jp_stl* module, which supports a very simple stimuli language similar to the (by now deprecated) STL language from Cadence [Cadence 97]. The module supports both stimuli specification and output value checks against user-defined master values.

- it is even possible to generate user-interface events directly via the *java.awt.Robot* class. Note that this class was introduced with JDK 1.2 and is not available when using either JDK 1.1 or the Microsoft VM.

The following example presents some of the Jython *jp_stl* module functions:

```
import jp_design_os        # file access
import jp_sim_control      # simulator control
from jp_stl import *       # STL language stuff
from VectorGen import *    # stimuli generation

# initialization
jp_design_os.openDesign( "/hades/examples/simple/dlatch.hds" )
jp_design_os.updateDesign()
jp_sim_control.updateSimulator()

# define inputs and outputs
pin_C = defPinByName( "C" )    # latch clock input
pin_D = defPinByName( "D" )    # latch data input
pin_Q = defPinByName( "Q" )    # latch data output

# define probes
addProbes( mode="ALL" )
probeQ, pNQ, pI1Y, pI0Y, pD, pC = defAllProbes()

# define stimuli timing and format
v = VectorGen()
v.defTiming( start=0.0, end=1.0E-8 )   # 10 nsec.
v.defFormat( [pin_C, pin_D] )

# stimuli data, one [pin_C, pin_D] data pair per line:
v.defStimuli( [
  1, 1,
  0, 1,
  0, 0,
  1, 0,
  1, 1,
  1, 0,
  1, 1,
] )

# run the simulation
v.start()
```
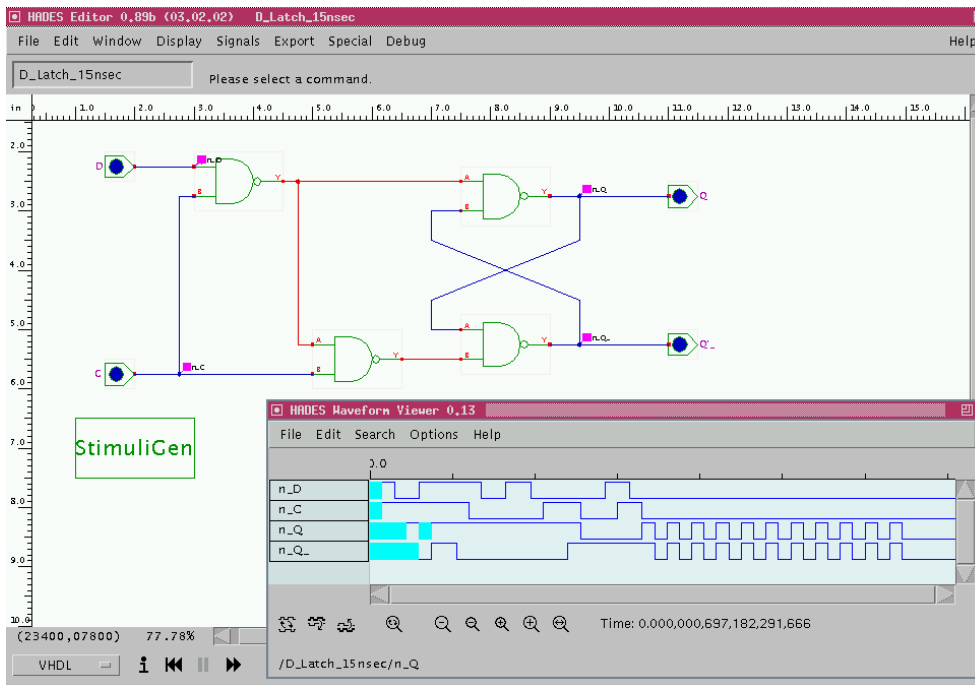
Figure 39: A demonstration of the *StimuliGenerator* simulation component, driving the data and clock inputs of the D-latch circuit. As the stimuli file references signals and components via their names, no explicit I/O ports are provided on the *StimuliGenerator*. The signals included in the waveform viewer are automatically selected via *trace* statements in the stimuli file.

8.5   Stimuli files and class StimuliParser

Despite all the advantages of the Jython scripting environment, there are still reasons to provide a Java-only way to specify simulation stimuli. To this end, Hades also includes the utility classes *StimuliGenerator* and *StimuliParser* in the *hades.models.stimuli* package. Together, those classes provide an easy way to write simulation stimuli, using concepts and syntax similar to a subset of VHDL.

The *StimuliParser* class is used to read and parse a stimuli specification, generating the corresponding simulation events. The parser is written using the JavaCC parser generator. The parser specification source code (*StimuliParser.jj*) is included in the standard *hades.zip* archive, allowing to modify or extend the stimuli file grammar when required.

StimuliParser

The *StimuliGenerator* class in turn is a simple *SimObject* that allows to embed and use a *StimuliParser* directly in a Hades design file. It does not provide any explicit I/O-ports in the schematics editor, because the underlying stimuli file accesses both simulation components (*SimObject*s) and signals directly via their name. The property sheet of *StimuliGenerator* allows to specify and edit the file name (or resource name) of the stimuli file. It also includes the option to enable or disable the *StimuliGenerator*, allowing to run a simulation with the specified stimuli (when enabled) or purely interactive (when disabled). However, the *Stimuli-Generator* cannot be switched off or on during a simulation run, because all events specified in the stimuli file are already scheduled at the elaboration phase.

StimuliGenerator

The stimuli file grammar tries to mimic a VHDL-like style for the stimuli specification, allowing for multiple parallel processes (process . . . end), using the '<=' operator for signal assignments and the wait statement to specify simulation times. To avoid a cumbersome syntax, strings are used as the values for the assignment statements and the *StimuliParser* converts the strings into values suitable for the assignment target. For example, strings like 'X' or '0' are used to specify std_logic values, and all of binary, decimal, and hex-formatted numbers like '0xcafe' are valid as values for std_logic_vector assignments. Additionally, signals can be traced (waveforms) and the simulator can be paused from the stimuli file.

VHDL style

Please check the parser specification file, *hades.models.stimuli.StimuliParser.jj* for the complete specification of the grammar.

Assignable Note that the target of an assignment can be either a signal or a simulation component that implements the *Assignable* interface. This tagging interface is currently implemented by the standard input components (*Ipin*, *PulseSwitch*, *HexSwitch* etc.) and some RTLIB components. This way, you can also control input switches from the stimuli file. Also, the graphical representation of the switches will be updated together with their output values.

The following example shows a simple stimuli file used to control the D-latch circuit shown in figure 39.

```
# example stimuli for a D-latch circuit,
# demonstrating the oscillation made possible with VHDL-style
# two-list simulation: the flipflop gates switch at the same time.
#
# a '#' hash sign starts a one-line comment
#


#
# select the signals to be traced (waveforms),
# names can be either local or fully-qualified
#
trace  n_D
trace  n_C
trace  n_Q
trace  /D_Latch_15nsec/n_Q_


process           # as many processes as you like
                  # everything initalized with U/X at t=0 (default)

  wait 15 ns      # wait until simulation time 15 nanoseconds
  D <= 1          # set D input switch to 1
  C <= 1          # set C input switch to 1

  wait 15 ns      # that is, simulation time 30 nanoseconds
  D <= 0          #

  wait 30 ns      # simulation time now 60 nanoseconds
  n_D <= 1        # assign a signal (n_D) instead of the switch


  ...             # more assignments here

  n_D <= X        # can use std_logic values: U X 0 1 Z W L H D
end

process              # a second, parallel process
  wait until 1.3 us  # run simulation for 1.3 microseconds,
  pause              # then pause the simulation
end
```

9 Writing Components

This chapter explains how to write new simulation models and how to integrate them into Hades. Unlike the other sections of this tutorial, this section assumes a solid background as a Java-programmer. After a short explanation of the simulation concepts, three examples are presented and discussed in detail. First, a simple gate is used to illustrate the basic methods required for a digital simulation component. Next, the simulation model for an edge-triggered D-flipflop shows how to detect input changes. Finally, a clock generator shows how to use *wakeup* events and a user-specified configuration dialog.

writing your own models

The following topics are covered in this chapter:

- Hades software architecture
- the *SimObject* and *Design* class hierarchy
- the discrete-event based simulation, including the *SimKernel* and *SimEvent* classes
- using the *DesignManager* to load and write files
- component configuration using *PropertySheet*
- graphical *Symbols* and animation

9.1 Overview and Architecture

An overview of the software architecture of the Hades framework is shown in figure 40. A simulation experiment consists of a set of simulation components, managed by the top-level *design* object. A hierarchical design style is possible, because each *design* object is also a simulation component and can therefore contain other design instances as subdesigns. The simulation algorithm is executed by a *simulation kernel* that interacts via *SimEvent* event objects with the simulation components. In the graphical editor, *signal* objects are used as intermediate components to manage the connections between simulation components.
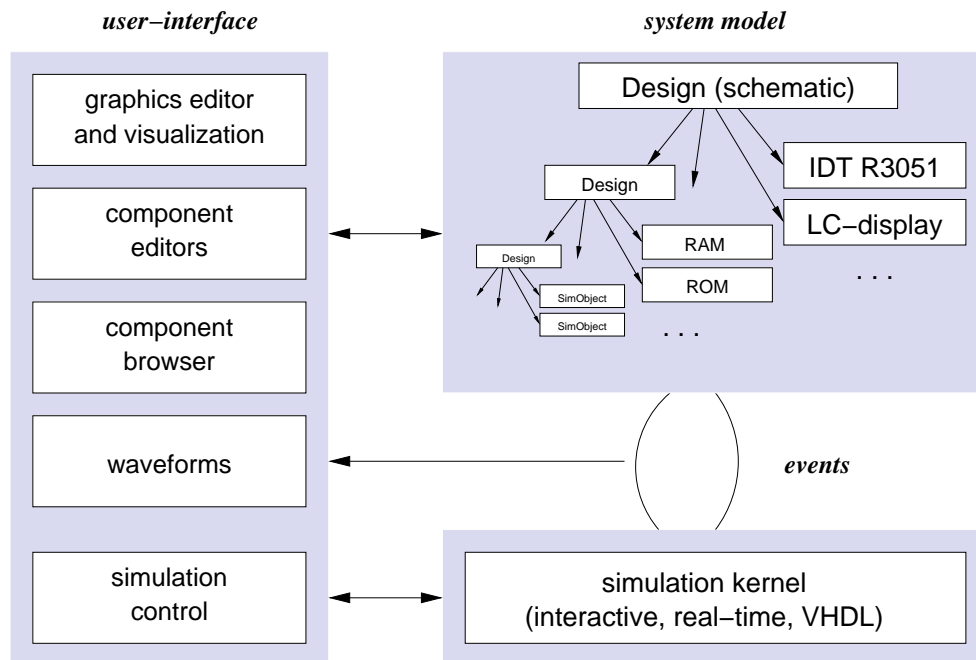
architecture

Figure 40: The software architecture of the Hades framework. A *design* represents a hierarchical collection of simulation models, which interact under control of the event-driven *simulation kernel*. The user interface consists of the main graphical *editor*, the component and library *browser*, simulation component *property dialogs* and editors, a *waveform viewer*, and the *simulation control*.

Several user-interface components are provided for interaction with the simulation experiment. The graphical *editor* is used to design a simulation setup (e.g. circuit schematics), but also for interactive control of a running simulation, and for graphical feedback via animation. All simulation components provide their own user interface, ranging from simple property dialogs for a few user-settable parameters to complex editors, e.g. hex memory editors. The waveform viewer allows recording and displaying signal waveforms.

SimObject Class Hierarchy

class SimObject   All simulation models in Hades are realized as subclasses of the generic base class *hades.simulator.SimObject*. This class fulfils a double role. First, it defines the basic interface required for the interaction between the simulation models and the event-driven simulation kernel. Second, it provides useful default implementations for many of these interface methods. For your own simulation models, you have to override some or all methods of *SimObject*. 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,   // simulation methods
                 Cloneable,          // allow copies
                 Serializable        // and serialization
{
  // object creation and initialization
  SimObject()             // the default constructor
  copy()                  // get a copy of this object
  initialize(String)      // read parameters from a String
  write(PrintWriter)      // write parameters as a String

  // simulation methods
  elaborate(Object)       // initialize the SimObject
  evaluate(Object)        // react to external events
  wakeup(Object)          // react to internal events

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

  ...                     // several accessor and utility methods
  ...                     // symbol and graphics stuff
  ...                     // configuration
}
```

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, and both the *elaborate()* and *evaluate()* methods. The next few paragraphs will explain the role of each group of these methods in more detail.

class hierarchy   The basic structure of the Hades simulation model class hierarchy is shown in figure 41. The common base class *hades.simulator.SimObject* is derived directly from *java.lang.Object*. The two interfaces *hades.simulator.Simulatable* and *hades.simulator.Wakeable* define the interaction between the simulation kernel and *SimObject*, based on the discrete-event simulation algorithm.

class Design   The most important subclass of *SimObject* is *hades.models.Design* which acts as a container for an arbitrary number of other *SimObjects* and represents one specific simulation system.
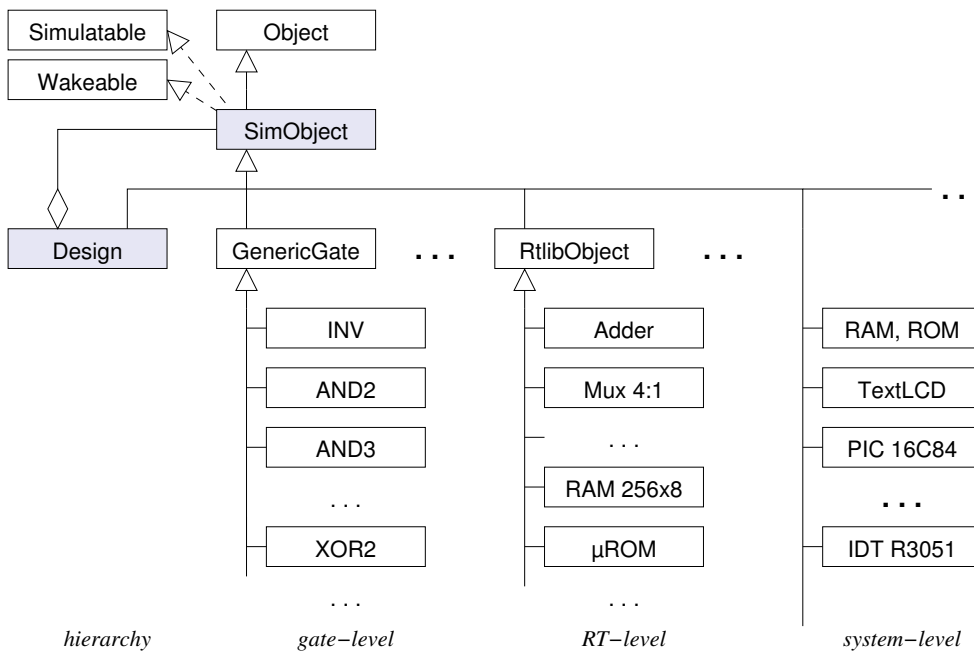
Figure 41: Overview of the Hades simulation model class hierarchy. All simulation components are derived from the *SimObject* class which implements the *Simulatable* and *Wakeable* interfaces for discrete-event simulation. The *Design* class represents a collection of simulation models and can be nested for hierarchical systems.

A *Design* does not define any behaviour itself, but is used to write and read the external representation of all contained simulation models via Hades `.hds` design files, including the graphical design schematic (if defined). A hierarchical system representation is possible, because a *Design* can also contain nested instances of other „sub-"*Designs* (the *composite design pattern*).

The individual simulation models like logic gates or flipflops are realized as direct or indirect subclasses of *SimObject*. Several utility subclasses are provided to ease the writing of new simulation models. For example, the *hades.models.gates.GenericGate* class provides a default implementation useful for simple logic gates based on the *StdLogic1164* logic system. Similarly, class *hades.models.rtlib.GenericRtlibObject* provides a default implementation for many register-transfer-level operations and data types.

9.2   Simulation Overview

During a simulation, the *SimObject* simulation components maintain their own state and exchange information via time-stamped event objects. The simulation kernel manages a time-sorted list (or similar data structure) and delivers event objects to the simulation components, which in turn may generate new event objects. This basic algorithm leads to a typical architecture and class hierarchy, which is sketched in figure 42.   *event-driven simulation*

All simulation components and signals implement the *Simulatable* interface that defines their interaction with the simulation kernel. Every simulation component is realized as a subclass of *hades.simulator.SimObject*, which implements the *Simulatable* interface and provides a basic set of utility methods and accessors but no behaviour.   *Simulatable*

The simulation engine is implemented by the class *hades.simulator.SimKernel* and its subclasses. For example, class *VhdlSimKernel* implements (most of) the two-list δ-delay algorithm specified in the VHDL reference manual [IEEE-93a]. The simulation kernel references its *EventList* object which manages the time-sorted list of simulation events. The sorting algorithm is implemented using instances of the inner class *EventList.EventListNode*. At the moment, a straightforward doubly-linked list structure is used to manage the nodes, which is simple and yet efficient for digital logic simulation (where most events can be appended to   *SimKernel*
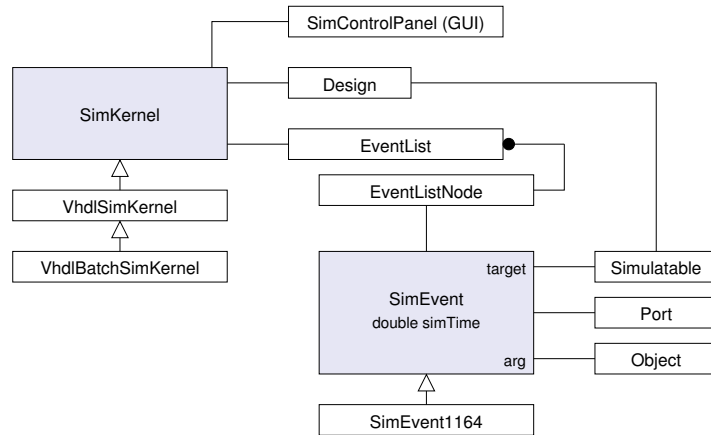
Figure 42: The simulation architecture used by Hades. Different simulation algorithms (like VHDL two-list simulation) are implemented by *SimKernel* and subclasses. Each kernel keeps a reference to the current *Design*, which in turn references the individual simulation components and signals, and uses class *EventList* to manage a time-sorted list of *EventListNodes* each of which references one atomic simulation event. A *SimEvent* is specified by the simulation time, the target object (a *Simulatable* class) modified by the event, and optional port and data value objects.

the list). To avoid the overhead of frequent Java object creation, class *EventList* maintains an internal buffer of event list nodes and uses its own memory management whenever possible. Naturally, you can also implement custom subclasses of *EventList* when more complex algorithms are called for.

SimEvent   All simulation events are realized as instances of class *hades.simulator.SimEvent* or its subclasses. A *SimEvent* contains five important data members:

```
class SimEvent {
  double  time;       // simulation time of this event
  Object  source;     // who generated this event
  Object  arg;        // (optional) payload object,
                      // e.g. a StdLogic1164 value
  Simulatable target; // who is to be notified
  Port    targetPort; // used to distinguish when an event
                      // is delivered to multiple ports
                      // on one target object object
  ...
}
```

As Hades simulation models are written as standard Java code, no special syntax is used for event creation and the simulation algorithm. Instead, the code relies on the *scheduleEvent()* method of class *SimKernel* to register an event object with the simulator eventlist. The typical code fragment for creating and registering a new event with the simulator looks like this:

```
...
SimObject    source = this;
Simulatable target = getOutputSignal();
StdLogic1164 value = getOutputValue();
double       time = simulator.getSimTime() + delay;

simulator.scheduleEvent(
  new SimEvent( target, time, value, source )
);
...
```
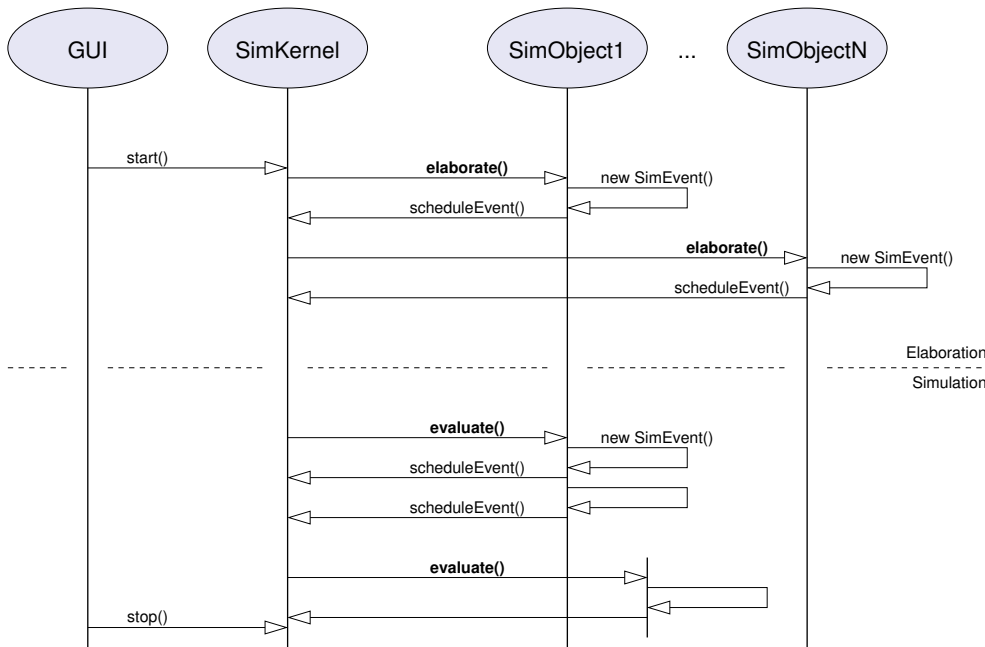
Figure 43: The simulation algorithm and event lifecycle. During the *elaboration* phase, the *SimKernel* calls the *elaborate()* method for all *SimObjects*. From those methods, an initial set of *SimEvent* objects is created and scheduled. After initialization, the *SimKernel* takes event objects from its event list and calls the *evaluate()* method of the target *SimObjects*, which in turn create and schedule new *SimEvents*.

The Hades framework also includes a few utility subclasses of *SimEvent*. For example, class *SimEvent1164* manages events for digital simulation based on the *StdLogic1164* logic system, and is both easier to use and more efficient than class *SimEvent*. Because most event objects are only used once and very shortlived, class *SimEvent* also includes support for its own memory management. While you can create simulation events by directly calling the Java constructor, it is often more efficient to call the static *createNewSimEvent()* method instead. This will look into a private buffer of previously *recycled() SimEvent* objects before calling the Java constructor.

The interaction of the simulation kernel and the simulation components relies on only three *elaborate()* very simple methods. At the start of the simulation, the simulator first calls the *elaborate()* *evaluate()* 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 calling them directly with arbitrary arguments.

A separate method *wakeup()* is provided to mark wakeup events used for periodic activi- *wakeup()* ties, 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. 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.

The remaining important part of the Hades class hierarchy are *signals* as implemented by *signals* class *hades.signals.Signal* and its subclasses. The software architecture of Hades does not restrict the program code in the *SimObject evaluate()* or *wakeup()* methods. Most importantly, a *SimObject* can always create new *SimEvent* objects with any other *SimObject* as the event target and arbitrary simulation time and payload objects. This allows for the direct interaction of simulation components. However, this direct connection of simulation components is not possible when using the graphical editor to create new circuits. Instead, the graphical editor uses *signals* as intermediate objects between source and target simulation components. To
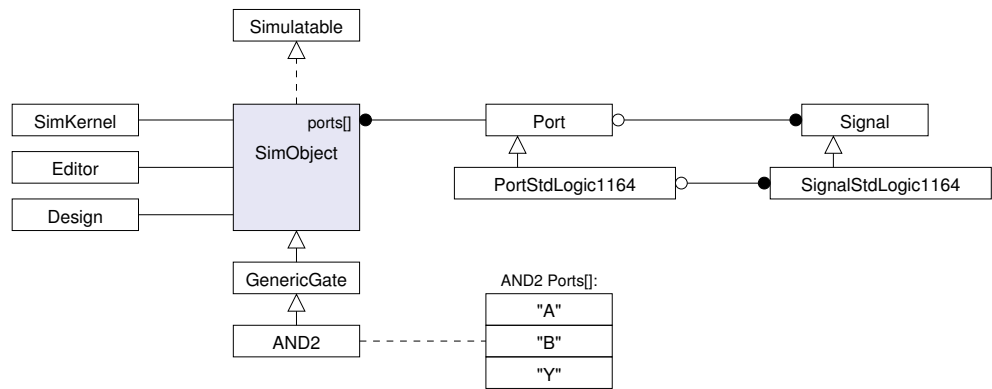
Figure 44: Every *SimObject* keeps references to its parent *Design*, the *Simulator* and the *Editor*. Signals are connected to the *SimObject* via *Port* object, which are accessed via the *ports[ ]* array. Additional subclasses can be used where appropriate. For example, classes *PortStdLogic1164* and *SignalStdLogic1164* provide utility functions for writing simulation components based on the *std_logic* simulation model.

this end, class *Signal* implements the *Simulatable* interface and can therefore also be used as the target object in a *SimEvent*.

The main role of a *Signal* is to distribute a single output event generate by a *SimObject* automatically to all other components connected to the *Signal*. The *SignalStdLogic1164* subclass additionally provides the *std_logic resolution function* required for buses driven by multiple gates. The second role of signals is the interface to the *Waveform* classes used to record and display waveform data. Finally, some signals subclasses support *glow-mode* and can be used for visualization.

9.3   Graphics: Static Symbols

symbols Besides the methods required for the simulation algorithms, 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 resource name of the symbol file, like this:

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

However, the default implementation in *SimObject* already does this (if slightly slower):

```
public String getSymbolResourceName() {
   String tmp = getClass().getName();
   return "/" + tmp.replace( '.', '/' ) + ".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 *Label*s. It is also possible to include FIG-file via *FigWrapper*. The *BboxRectangle* is special, because it is used to define the *bounding box* of the symbol in the editor.
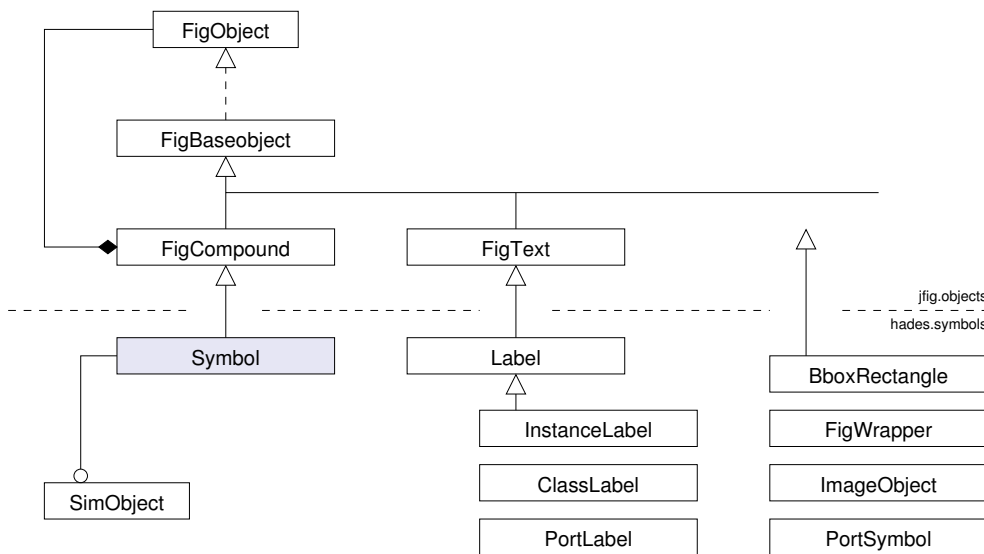
Figure 45: Class hierarchy overview of the *hades.symbols* package. The graphical *Symbol* for a simulation component is derived from the *FigCompound* class, which is used as a container for any number of *FigObjects*. Utility classes are provided for the polylines, circles, several variants of text labels, and the symbol bounding box. A *PortSymbol* defines a pin for connection to the simulation component; the schematics editor checks for mouse-clicks on *PortSymbol* positions to decide when to start *create wire* operations.

The following listing, taken from */hades/models/gates/Inv.sym*, shows the symbol file for a basic inverter. All coordinates are given in inch/2400 and relative to the top left corner:

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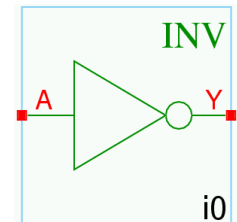
The file shows the use of the *BboxRectangle*, *ClassLabel* and *InstanceLabel* objects. Also, note the *PortSymbol* objects, used to define both the positions and the names of the object's ports. The *PortLabel* objects are just annotation without any other function. Many coordinates in the above example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

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* classes are used to embed GIF- and JPEG-images or FIG-format drawings [xfig 3.2] into Hades symbols. Because the planned symbol editor is not ready yet, you have to create the symbol files with a text editor. However, this is usually a simple task when compared to the complexity of writing the simulation functionality.

(To create a default symbol for a new simulation component, you can also create a dummy *design* schematic, add *Ipin* and *Opin* components, change their names to the names of your simulation component ports, and then execute the editor *Edit ▷ Create Symbol* function.)

9.4   A Simple Example: Basic AND2 Gate

An AND gate
with trivial timing
This section presents the full source code for a simple 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 delay parameter `t_delay` that specifies the output delay (in seconds) of the output after any input changes.

class template
As usual, 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 many models need to access classes from the *hades.simulator* and *hades.signals* packages, and digital logic gates based on the StdLogic1164 logic model often 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 2-input AND-gate with propagation delay.
 */
public class And2 extends GenericGate {
  ...
  public And2() {  ... }
  public void elaborate( Object arg ) { ... }
  public void evaluate( Object arg ) { ... }
  ...
}
```

GenericGate
As already indicated above, simulation models in Hades are written as standard Java program code, without the special syntax and support provided by dedicated simulation languages. However, it is often possible to write or re-use intermediate utility classes that contain default implementations for most of the functionality required for a new simulation model. In this example, the AND2 gate is derived from the *hades.models.gates.GenericGate* class, which provides several utility accessor methods and a default property sheet to specify the gate delay.

Accessor Methods

Most simulation models need not change the basic accessor and setter methods provided by *SimObject*. For example, the *getSimulator()* and *setSimulator()* methods are used to specify the simulation kernel responsible for the simulation model and its events, while *getDesign()* and *setDesign()* reference the design object the simulation model belongs to. The *getName()* and *setName()* return a set the *relative* name of the simulation component in its design object, while *getFullName()* returns the full hierarchical name including all subdesign names up to the top-level design (for example */adder32/adder8_1/sum_5/and2*).

The *getSymbol()* and *setSymbol()* methods reference the graphical symbol of the simulation model in the editor. Only simulation models with a *dynamic symbol*, used to visualize the current state of the object throughout the simulation, might have to override the *setSymbol()* method. See the example in section 9.6 for details.

The *GenericGate* class provides additional accessor methods to read and set its gate propagation delay member variable, *getDelay()* and *setDelay()*.

Constructor and instance initialization

To avoid some of the subtleties of the Java language concerning object initialization and constructors, Hades requires that all simulation objects provide a default constructor (without arguments). While a *SimObject* subclass may provide additional constructors, these are never used by the Hades framework.

constructor

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*, that is, the external connections to the simulation object. For simulation performance reasons, a simple array *ports[]* is used as the data structure to manage and reference the individual ports. This *ports[]* array is declared but not initialized in class *SimObject*. Therefore, simulation components are responsible to allocate a suitable array and to fill in the necessary number of *Port* objects. 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.

ports

Technically, only the *ports* array is required for simulation. However, because it is much more readable to write *port_Y* instead of *ports[2]* in the rest of the code, the AND class also declares three instance variables as aliases to the *ports* array members:

port aliases

```
public class AND2 ... {
  private PortStdLogic1164  port_A, port_B, port_Y;

  public And2() {
    super(); // -> GenericGate -> SimObject -> Object

    ports = new Port[3];
    ports[0] = new PortStdLogic1164(this, "A", Port.IN, null );
    ports[1] = new PortStdLogic1164(this, "B", Port.IN, null );
    ports[2] = new PortStdLogic1164(this, "Y", Port.OUT, null );

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

However, most simulation components require additional parameters, e.g. to specify the gate delay, file- or resource names, a clock period, or simply the default color. Therefore, directly after object creation with the default constructor a separate *initialize()* method is called, which takes a single *String* argument. The simulation component then parses this string to setup its parameters. When saving a Hades design from the editor, the corresponding *write()* method is called for all simulation components, allowing each component to write out its parameters in its own formatting. The resulting string is saved without further modification in the output design file, and used as the argument for *initialize()* when later loading the design. Therefore, each component is responsible for its own external representation, which may consist of simple binary-data or fully formatted readable text strings.

initialize()
write()

As an example, the following code shows the implementation of the *initialize()* and *write()* methods in class *GenericGate*. Both a version number and the gate delay are written and read back:

```
  ...
  protected double t_delay = 5.0E-9; // 5 nanoseconds

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

```
public boolean initialize( String s ) {
  StringTokenizer st = new StringTokenizer( s );
  try {
    versionId = Integer.parseInt( st.nextToken() );
    t_delay   = Double.parseDouble( st.nextToken());
  }
  catch( Exception e ) {
    ... // handle invalid String format
  }
  return true;
}
```

Elaborate and Evaluate Methods

elaborate()   It now remains to write the implementation for the *elaborate()* and *evaluate()* methods, that is, the actual simulation behaviour for the new simulation component. As explained above, the *elaborate()* method is called by the simulation kernel at the start of the simulation before any simulation events are processed. Because an AND2 gate needs no special initialization beyond the gate delay parameter setup already done in the constructor and the *initialize()* method, the *elaborate()* method is almost empty.

However, this is the right place to get and store a reference to the current 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:

```
/**
 *  And2.elaborate(): called to initialize a component.
 *  We save a reference to the current simulatin engine.
 */
public void elaborate( Object arg ) {
  simulator = parent.getSimulator();
}
```

evaluate()   As with all Hades simulation models, the *evaluate()* method defines the individual behavior of the AND gate. The following implementation relies heavily on the properties of the std_logic logic system and the methods provided by class *hades.models.StdLogic1164*. It first retrieves the current gate input values by calling the *getValueOrU()* method on the *PortStdLogic1164* ports. This method returns the current value of the signal connected to the port, or the undefined *U* value if no signal is connected. Next, the *StdLogic1164.and()* method is called to calculate the logical AND function of both input values. Finally, an utility function is called to create a new simulation event with the newly calculated output value:

```
/**
 * And2.evaluate(): calculate the logical AND of the gate
 * inputs after propagation delay "t_delay".
 */
public void evaluate( Object arg ) {
  StdLogic1164 value_A = port_A.getValueOrU();
  StdLogic1164 value_B = port_B.getValueOrU();

  StdLogic1164 next_Y  = StdLogic1164.and( value_A, value_B );
  scheduleOutputValueAfter( port_Y, next_Y, t_delay );
}
```

readability   The utility method *scheduleOuputValueAfter()* creates and registers a new event with payload value `next_Y` for the signal connected to `port_Y` (if any), occuring exactly `t_delay` seconds after the current simulation time. The method, implemented in class *GenericGate* also takes care of the various null-pointer checks required in an interactive simulation environment. The

method arguments are such that the code reads similar to the equivalent VHDL statement, `port_Y <= next_Y after t_delay;`

To demonstrate the improvement in readability gained by the *scheduleOutputValueAfter()* method, the following code example shows the actual implementation in class *GenericGate*. The method first checks whether a signal is connected to the output port of the gate, and returns if not. Next, the simulation time of the output event is calculated from the current simulation time and the gate propagation delay. Finally, a new simulation event is created with the corresponding parameters and scheduled:

```
/**
 * utility method to schedule a simulator event after
 * "delay" seconds from the current simulation time.
 */
public void
scheduleOutputValueAfter( Port port,
                          StdLogic1164 value,
                          double delay )
{
  Signal signal = port.getSignal();
  if (signal == null) return;

  double time = simulator.getSimTime() + delay;
  simulator.scheduleEvent(
    SimEvent1164.createNewSimEvent( signal, time, value, port )
  );
}
```

Runtime Checks

The runtime checks in the above code examples are necessary to support the interactive simulation mode. In a traditional compiled simulation environment, the whole structure of a simulation experiment including all component connections is known at compile time. Therefore, the compiler can check and verify that all inputs for a simulation component are connected, so that runtime checks are not necessary. However, the interactive environment provided by Hades allows the user to edit and change the simulation experiment during a running simulation. For example, the user can delete signals or components and afterwards add new components. This is turn requires that all simulation components detect and handle situations with open inputs and possibly illegal input combinations.

compiled vs. interactive environment

As the above examples show, the use of the *std_logic* logic system together with utility methods like *getValueOrU()* makes most such checks implicit, without additional source code. However, when using the basic classes like *hades.signals.Signal* and *hades.simulator.Port*, the resulting code will often require checks like the following:

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

9.5   A D-Flipflop

As a slightly more complex example, this section describes an edge-triggered D-flipflop with asynchronous reset input. To keep the example short, only the constructor and the *evaluate()* method are shown. Similar to the AND gate example, the constructor is only used to create the ports of the flipflop. As usual, *C* and *D* are the clock and data input, *NR* is the active-low asynchronous reset, and *Q* the data output:

```
public Dffr() {
  super(); // -> GenericFlipflop -> SimObject -> Object

  port_Q  = new PortStdLogic1164( this, "Q",  Port.OUT,  null );
  port_D  = new PortStdLogic1164( this, "D",  Port.IN,   null );
  port_C  = new PortStdLogic1164( this, "C",  Port.IN,   null );
  port_NR = new PortStdLogic1164( this, "NR", Port.IN,   null );

  ports = new Port[4];
  ports[0] = port_Q;
  ports[1] = port_D;
  ports[2] = port_C;
  ports[3] = port_NR;
}
```

In the *evaluate()* method, the input ports are queried for their current values. The check for the rising clock edge is similar to the typical VHDL idiom, CLK'event and CLK = '1'. When the flipflop detects undefined input values, it generates the *X* output value:

```
public void evaluate( Object arg ) {
  StdLogic1164  value_D   = port_D .getValueOrU();
  StdLogic1164  value_C   = port_C .getValueOrU();
  StdLogic1164  value_NR  = port_NR.getValueOrU();

  char tmp = 'u';

  if      (value_NR.is_0())   tmp = '0';  // reset active
  else if (value_NR.has_UXZ()) tmp = 'X';  // reset undefined
  else if (value_C.has_UXZ())  tmp = 'X';  // clock undefined
  else
      if (port_C.hasEvent() && value_C.is_1())
  {                                         // rising clock edge?
    if      (value_D.is_0())   tmp = '0';
    else if (value_D.is_1())   tmp = '1';
    else                       tmp = 'X';
  }
  else {                                   // others: store value
    return;
  }

  StdLogic1164 value_Q = new StdLogic1164( tmp );
  scheduleOutputValueAfter( port_Q, value_Q, t_delay );
}
```

The code also demonstrates several functions from *StdLogic1164*, including the constructor with a *char* argument. The boolean predicates *is_0()*, *is_1()*, ... predicates test for the corresponding logical value, while the *is_UXZ()* method returns true when the logical value is undefined ('U', 'X', 'Z', 'W', 'D'). The utility method *hasEvent()* in class *PortStdLogic1164* returns true when the signal connected to the port has an event at the current simulation time.

9.6   **Wakeup-Events: The Clock Generator**

This section presents a more complex example, a clock generator called *ClockGen*. It explains
the *WakeupEvents* used for periodic activities independent of input changes. The following
section will use the clock generator to also demonstrate the *configure()* method and *Proper-
tySheet* classes, and the interface to the Hades editor, including animation and the reaction to
mouse-events.

wakeup()
configure()
graphics

The clock generator model allows selecting 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 *ClockGen*s
symbol. Additionally, the variable *port_Y* is used as an alias to the output port of the *Clock-
Gen*, and *value˙U*, *value˙0*, and *value˙1* will hold the *StdLogic1164* values corresponding to
their name:

```
package hades.models.io;          // our package

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

import hades.symbols.*;           // graphics imports
import javafig.objects.FigAttribs;
import java.awt.*;

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

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;   // 1.0 sec period, i.e. 1Hz
  private double dutycycle = 0.5;   // 50% dutycycle, symmetric
  private double offset    = 0.0;   // no initial delay

  private boolean  running = true;  // start in active state
  private Circle   circleOnOff;     // on/off symbol

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

*a DOC-comment
explaining the
timing parameters*

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]*. Also, the constant output values used several times later on 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 9.4.

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 *schedule()* method is used to make to code more readable:

```
/**
 *  And2.elaborate(): called to initialize a component.
 *  We store a reference to the simulation engine.
 */
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, this ));
}
```

The *wakeup()* method is constructed similar to *elaborate()*. It first schedules all output events   *wakeup()*
for one clock cycle, namely the change from *1* to *0* at the current simulation time plus *(duty-
cycle*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() {
  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 );
}
```

Because the *ClockGen* model has no input, the simulator should not normally call its *evalu-*   *evaluate()*
ate() 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 function-
ality of the clock-generator model, including the initialization, its periodical wakeup activi-
ties, and the output event scheduling.

The next two methods are required to save and restore *ClockGen* objects with their current   *I/O*
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 *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 );
  }
}
```

The *write()* method is called on saving a design, and it should return a *String* with a textual
representation of the component's parameters:

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

9.7   Dynamic Symbols and Animation

graphics   The next part of this section explains how to write simulation models that access the graphics functions 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 27 on page 40. 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:

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

manipulate
attributes
```
  FigAttribs 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 );
}
```

FigAttribs   Each graphical object in Hades contains a reference to a *FigAttribs* 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.FigAttribs* for a complete description of all attributes

and their possible values. The above example shows the typical use of the *FigAttribs* 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 *FigAttribs* 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.

The methods *setSymbol()*, *initDisplay()*, and *showState()* are required to visualize the current   *mousePressed()*
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   *inside vs. border*
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*.

The algorithm to implement *mousePressed()* might seem trivial for this example. On each   *deleting events*
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 schedules two *SimEvents* and one fu-
ture *WakeupEvent* with the simulator when its *wakeup()* method is called. Therefore, in order
to really stop the clock generator, it is not sufficient just to reset the *running* variable. Instead,
it is necessary to also remove all future events already 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();
}
```

9.8   PropertySheet and SimObject User Interfaces

SimObjects as Beans   The remainder of this section explains how to interface simulation models with the *Property-Sheet* class which provides an user interface to access and set model parameters. In principle, all simulation models in Hades are written to be Java-Beans [Sun 96] compliant. So far, however, most simulation models use only the two basic requirements of the Beans specification. First, all simulation models adhere to the standard design pattern for all user-settable properties, that is, they provide *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 basic getter and setter methods are required for these three variables, while the base class *SimObject* already provides the suitable methods *getName()* and *setName()* to edit the instance name:

```
// Java Beans compatible getter/setter methods to access
// the parameters of the clock generator:
//

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;
  }
}
```

Together, these methods are sufficient to interface the simulation component to standard Java Beans property editors. Unfortunately, such property editors are not always available. An alternative is to use the *hades.gui.PropertySheet* class, which automatically constructs a (very basic) user interface window from a set of Beans variable names. As class *PropertySheet* expects setter methods with a *String* argument, we provide these additional 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 the base class *SimObject* only supports changing the instance name of the component.

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. While the appearance of Java AWT windows is platform dependent, the *PropertySheet* for the *ClockGen* component should look similar to the example shown in figure 22 on page 33.

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
    = 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 *Property-Sheet*s text fields. Clicking either *Cancel* or *OK* will also close the *PropertySheet* window.

While *PropertySheet* is a simple and convenient way to provide an user interface for simple simulation components, more complex simulation model often require their own specialized user interface. To this end, you can rely on the classes in the *hades.gui* package, or write your own user interface from scratch. For example, the following code example was taken from the *hades.models.rtl.RAM_256x8* memory component. It creates an instance of the *MemoryEditorFrame* class to provide a fully interactive hex-editor for the memory data:

custom dialogs

```
// hades.models.rtl.RAM_256x8 user interface: Hex memory editor
...
protected hades.gui.MemoryEditorFrame  MEF = null;


public void configure() {
  if (MEF == null) { // create on first call, 8 bytes per row
    MEF = new hades.gui.MemoryEditorFrame( this, 32, 8, "Edit RAM");
    this.addMemoryListener( MEF );
  }
  MEF.pack();
  MEF.setVisible( true );
}
...
```

9.9   Assignable

Some simulation models, for example switches like *hades.models.io.Ipin* or *PowerOnReset*, are meant to be controlled via their user interface during interactive simulation. Unfortunately, relying on the user interface only makes it rather difficult to control such components from scripts or to generate (synthetic non-interactive) input during batch-mode simulation.

To this end, the tagging interface *hades.simulator.Assignable* provides a way to mark simulation components whose internal state can be meaningfully set asynchronously. In order to provide a simple and consistent API, the new value for the simulation component is always encoded as a string. The *Assignable* component is responsible for parsing the string and to generate meaningful error messages for malformatted values. Therefore, the interface just defines a single method that takes a string to specify the new value and a double argument for the simulation time:

```
public interface Assignable {
  public void assign( String value, double simTime );
}
```

The following code-snippet shows a prototype implementation in class *PowerOnReset*:

```
public void assign( String value, double time ) {
  StdLogic1164 tmp = new StdLogic1164( value.charAt(0) );
  scheduleOutputValue( port_Y, tmp, time );
  if (visible) simulator.scheduleWakeup( this, time, tmp );
}
```

9.10   DesignManager

external resources   Many Hades simulation components need to access external resource data. For example, memory components will usually read their memory data from an external file (or URL) instead of storing the data directly into the Hades design file. The use of external resources is always recommended when the amount of external data is large or other programs besides the Hades editor should access the data. For example, the *FigWrapper* class uses the external resource to read a graphical symbol from a FIG file, created by the xfig or jfig graphical editors. Similarly, a microprocessor simulation model might use external files to read processor configuration data, program memory contents, and debug information created by an external assembler or compiler.

DesignManager   The *DesignManager* class, together with a few utility classes in the *hades.manager* package, provides a set of methods to manage access to such external resources. At the moment, *DesignManager* supports the following use-cases and scenarios:

- loading and saving existing design files,
- loading and saving external resource data,
- loading *SimObjects*
- loading *Image* files or thumbnails,

    all of the above from local files, JAR or ZIP archive files, URLs, and using either absolute or relative path names,

- creating thumbnails from design files or *SimObject* symbols,
- selecting a file or URL name, either for reading or writing, interactively via the graphical user interface. Depending on whether the *DesignBrowser* (colibri) is used, this will open the browser or display a standard AWT dialog window.
- several utility methods to manage file names.

The heart of *DesignManager* is the *getInputStream()* method, which takes two arguments.   *getInputStream()*
The first argument takes an *parent* object reference, used to pass in context information. For
example, when loading external resources for simulation components, you would usually pass
in the simulation component as the parent argument. The second, string, argument specifies
the resource, file, or URL name:

```
DesignManager   DM = DesignManager.getDesignManager();

String    className = "hades.models.rtl.ROM_1Kx8";
String     initName = "zip://demo.zip?/welcome.rom";
SimObject    eprom = DM.getSimObject( className );
InputStream stream = DM.getInputStream( eprom, initName );
... // parse data from stream
```

The algorithm used by *getInputStream()* is split into several phases.

First, if the *pathname* argument begins with an explicit protocol specification, this proto-
col is used to locate the resource. Valid protocols include `file://`, `file:`, `resource://`,
`zip://` (Hades specific, see below), and all protocols supported by the Java VM, includ-
ing `http://` or `ftp://`. Examples of such pathnames are *file://c:/temp/welcome.rom* or
ftp://ftp.uni-hamburg.de/pub/temp/adder32.hds.

Next, the algorithm tries to load a Java resource called *pathname* from the current classpath.
If necessary, a slash char is prepended to the name. If the resource can not be found, the
third phase of the algorithm checks for a file. Finally, the fourth phase of the algorithm
dispatches to the helper method *lookForInputStream()* which actually uses the *parent* object
for context-sensitive search.

The `zip://` protocol is used to search for resource files in JAR or ZIP-format archive files,
which have to be registered by calling *registerZipFile()*. For example,

```
DesignManager DM = DesignManager.getDesignManager();
DM.registerZipFile( "dsp-examples.zip" );
DM.registerZipFile( "mips-examples.zip" );

InputStream tmp = DM.getInputStream(
   null,
   "zip://dsp-examples.zip?/chapter1/exercise2.hds" );
```

If your simulation component provides its own graphical user interface, it will also be neces-   *selectFileOrURLName*
sary to add a user interface control that allows the user to select the corresponding external
file or resource. To this end, *DesignManager* provides the *selectFileOrURLName()* method
which display a file selection dialog and returns the string with the file or URL name (or null
if canceled by the user). You can specify the dialog title, the default file name, the default
extension, and the mode (load or save) via the method parameters:

```
...
DesignManager DM = DesignManager.getDesignManager();
String default = "c:/temp/hades/examples/simple/dlatch.hds";
DM.setFileDialogDirectoryAndFilename( default );

String filename = DM.selectFileOrURLName(
                     "Select a new design... (*.hds)",
                     "",
                     ".hds",
                     java.awt.FileDialog.LOAD );
// modal dialog: wait for user response

Design tmp     = DM.getDesign( this, filename, true /*toplevel*/ );
tmp.setVisible( true );
...
```

9.11   DesignHierarchyNavigator

Sometimes, it is necessary to reference other simulation components or signals via their name. Naturally, this is possible via the *getComponent()*, *getSignal()*, and *getParent()* methods in class *Design*. However, especially for hierarchical designs, it may be easier to use the utility class *DesignHierarchyNavigator*. Its *findSignal()* and *findSimObject()* methods supports both relative and fully qualified hierarchical names, and they cache results for efficient lookup:

```
import hades.utils.DesignHierarchyNavigator;

...
DesignHierachyNavigator navigator
              = new DesignHierarchyNavigator( design );

Signal signal = navigator.findSignal( "/uut/adder8/cla4/cout" );
SimObject obj = navigator.findSimObject( "clockgen" );
...
```

9.12   Logging messages

Sometimes it is useful to log the information, warning, and error messages from the Hades editor into a file. While the logging can be enabled via the GUI in the *Console* window, it might be better to enable message logging under program control. Using the *hades.gui.Console* class, here is how:

```
Console console = Console.getConsole(); // singleton
console.setLogFileName( "test.log" );
console.setLogFileEnable( true );
console.openLogStream();
...
console.println( "-#- your message here" );
...
console.closeFlushLogStream();
```

10   FAQ, tips and tricks

This chapter collects the answers to frequently asked questions, a few tricks for common design tasks, and a list of major known bugs and features in Hades.

Naturally, for those of the following answers which contain example commands, you should substitute the program, directory, and file names corresponding to your system.

10.1   Frequently asked questions

10.1.1   The documentation is wrong?

Unfortunately, that can be true. While the documentation is believed to match the Hades software at the time of writing, it is very hard to keep both in sync — which is true for most software projects. However, most changes to the software are made as improvements, so you should often be able to guess how the software should behave. Please report errors to the documentation, and include the proposed changes or a corrected text, if possible.

10.1.2   The editor hangs?

This is unlikely, unless you ran out of memory or the Java virtual machine crashed. 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, which should indicate what action the editor expects. Then supply the missing parameters or points, or cancel the command.

10.1.3   The popup menu is dead

This usually indicates that the editor is busy or expects some (mouse) input. For example, the popup menu is deactivated during a *move component* or *delete component* operation, because the right-mouse button is used to cancel the operation.

Check the status message panel at the top of the editor window, which should indicate what action the editor expects. To re-active the menu, just press the ESC-key to cancel the ongoing editor command.

10.1.4   How do I cancel a command?

To cancel an editor operation like *delete component*, just press the ESC (escape) key, or select the *Editor ▷ Edit ▷ Cancel* menu item. The status message panel should then indicate that the operation has been canceled, and that the editor is ready for new commands.

10.1.5   I can't get it running

Please read section 3 carefully again and check that your system meets all requirements. Then try the following steps:

1. check that your Java virtual machine is correctly installed. A good test is to run other Java programs. If this fails, try to re-install the JVM or another JVM.

2. check that your `hades.jar` archive is ok. See tip 10.1.6 on how to do this.

3. try double-clicking the `hades.jar` if your operating system and JVM allow this. If this works, but your startup script or double-clicking a ".hds" file does not, check the file type and file association settings (see 3.9).

4. make sure that you are specifying the correct name for the Hades editor main class. Use the following command:
   `java hades.gui.Editor`
   to avoid the following common mistakes:
   `java Editor` (missing package names)
   `java hades.gui.Editor.class` (don't specify .class)
   `java hades\gui\Editor` (backslashes instead of dots)

10.1.6   How to check whether my hades.jar archive is broken?

To check whether the `hades.jar` (or `hades.zip`) archive file was downloaded correctly, just try to list its contents with your favorite packer application, e.g. WinZip, PowerArchiver, or the `jar` utility program included with the JDK. In any case, do not unpack the archive, just list its contents. For example, try the following command, which should print several hundred entries, `jar tf c:\temp\hades.jar`.

If the above command fails, or if a packer like WinZIP fails to list the archive contents, the file might have been truncated or damaged during download. In both cases, try downloading again, if possible from a different machine or using a different browser.

10.1.7   I get a ClassNotFoundError

If your Java virtual machine prints a `ClassNotFoundError` message, it could not find or load the Java class mentioned in the error message. This is most often due to installation and setup problems, but it may also indicate a programming error or versioning problems. For example, the JVM may pick up conflicting versions for the given class, or an incompatible change to one the superclasses of the problematic class was detected.

First, see the above FAQ entry and check whether your `hades.jar` archive is ok. Try to list the `hades.jar` archive contents, to see if the problematic class is actually there. Also, re-read the installation section for information about JVM installation and possible CLASSPATH issues. If necessary, ask your local Java guru for help.

The following code examples show a minimal CLASSPATH setup for different environments. First Unix or Linux and the `bash` shell:

```
bash$ export CLASSPATH=/home/joe/hades/hades.jar:.
bash$ java -Xmx256M hades.gui.Editor
```

Unix or Linux running the `tcsh` shell:

```
tcsh> setenv CLASSPATH /home/joe/hades/hades.jar:.
tcsh> java -Xmx256M hades.gui.Editor
```

Windows running the `cmd.exe` shell:

```
c:> set CLASSPATH=c:\home\joe\hades\hades.jar;.
c:> java -Xmx256M hades.gui.Editor
```

10.1.8   The editor starts, but I cannot load design files

If the editor starts, but fails to load the design examples or your own design files, you should check your directory installation as described in 3.5 on 19. If possible, avoid special chars (and spaces) in the directory names. One common mistake is to use unpack the design example archive without the directory names.

On multi-user operating systems like Linux or Solaris, you should also check that the owner and permission flags are set up correctly for all design directories and files. Finally, when using older versions of Hades (0.88 and before), or JDK 1.1.x, you might have to check your CLASSPATH.

10.1.9   The Java virtual machine crashes

Overall, the stability of Java virtual machines has improved greatly during the last two years. Therefore, if your JVM crashes directly, this is almost always due to an installation problem. A typical problem is having the binaries for one JVM but the libraries for another version on your search path. When started, the binary will load the wrong libraries, which often crashes the JVM instantly, but it may also lead to subtle errors.

On Windows, you might try to deinstall and then re-install the Java virtual machine. On other systems, you may have to ask your system administrator for help.

The most basic test for the Java installation is to run the JVM without any arguments, which usually will print out a help message and/or a version number. Try running `java -version` for the JDK or `jview /help` for the Microsoft VM. If you experience frequent JVM crashes, you might also try to disable the JIT (just in time) compiler used by your JVM. See the documentation and release notes for your Java virtual machine for details.

10.1.10   The editor crashes

If the Hades editor crashes shortly after program start, you should first check the command line and parameters used to start the editor.

One common mistake is to provide the wrong parameters to the editor. This is sometimes hard to detect, because the editor will just print a message and then exit. For example, just calling `java hades.gui.Editor demo.hds` will not work, because the editor expects the `-file` option in this case, `java hades.gui.Editor -file demo.hds`. Note that the error message printed by the editor is not even visible when running with the `javaw.exe` program of the JDK on Windows. If you suspect such problem, use `java.exe` instead of `javaw.exe`.

If the command line parameters appear correct, there may be a problem with your Java virtual machine. In such cases, it is worth trying a Java virtual machine from a different vendor, if available. For example, JDK 1.3 compatible virtual machines for Linux are currently available from Sun, IBM, and the Blackdown team. It could also help to disable the JIT compiler fo your JVM.

Unfortunately, you may also experience crashes during editing or simulation. Most often these are related to AWT problems during screen repainting. Therefore, it might help to disable glow-mode and to disable rtlib-animation or to reduce the animation frame-rate via the corresponding *Editor* ▷ *Display* and *Editor* ▷ *Window* ▷ *Select repaint frequence* menu items.

Please report all combinations of operating system, virtual machine, Hades version, and possibly graphics drivers, which prove incompatible with running Hades.

10.1.11   I cannot double-click the hades.jar archive

This may happen for several reasons, some of which are platform dependent. However, while double-clicking the `hades.jar` is simple, using a small script to start the editor may even be better.

First, see tip 10.1.6 on how to check that that your `hades.jar` archive is ok. Next, check that your Java virtual machine is able to run Jar-files directly, and that the virtual machine is still registered for the JAR file type. For example, the default installation of many archive tools will change the file association to the archiver instead of the Java virtual machine. In that case, you will either have to re-install the Java VM or to repair the file associations manually.

Finally, try to run the command corresponding to double-clicking in a command shell, because this way you will see any error messages printed by the Java virtual machine. The exact command will depend on your operating system, JVM version, and file locations. An example command should look similar to:

```
c:\jdk131\bin\java.exe -jar "c:\my files\hades\hades.jar"
```

10.1.12   I got an OutOfMemoryError

This can easily happen when working with large circuits or large simulation traces, because some Java virtual machines start with very low memory limits, e.g. 32 MBytes. Note that a simulation trace (waveform) has to store both the time and the data value for each event on the signal, which amounts to about 10 bytes per event. At 100.000 or more events per seconds, the simulator may allocate more than 1 MByte of memory per second...

To avoid this problem, start your Java virtual machine with a higher memory limit. For example, to set a memory limit of 512 MByte for JDK 1.3, you would start Hades with the following command:

```
java -Xmx512M hades.gui.Editor
```

10.1.13   What are those editor messages?

For most messages that are printed to the console window or the stdout/stderr output streams, Hades uses a simple encoding scheme to indicate the nature of the message:

```
-I- this is an information message
-W- a warning message, should be read and checked
-E- an error
-F- fatal error, usually requires to restart the application
-#- debugging messages
```

10.1.14   Missing components after loading a design

Sometimes, some components may be missing after loading a design file into the editor. This is usally accompanied by dozens or hundreds of error messages from the editor, because it also reports errors for each signal to a missing component.

The most common problem is that the editor cannot find a subdesign referenced by the design file, which usually means that the subdesign `.hds` file is either missing or in the wrong directory. Check the first error message to see where Hades was looking for the missing file, and ignore all To solve the problem, you can either move the subdesign files to the position expected by Hades, or use a texteditor to change the subdesign reference in the top-level design file.

The second reason is that a SimObject is either missing from the `hades.jar` archive, or that it could not be instantiated. Please check the editor messages in the editor console window and the command shell for Java exception traces, which could help to locate the error. If your design references third-party simulation models, check that those models are included in the Java virtual machine CLASSPATH settings.

10.1.15   Editor prints hundreds of messages while loading

This will usually happen, when the editor cannot find a subdesign or a SimObject while parsing a `.hds` design file. In the first phase of parsing, it will print useful error messages about each missing component. However, during the second phase of parsing, it will also report another error message for each signal which should be connected to the missing component. As the number of signals connected to a single simulation model can be quite high, this often results in a very large number of error messages.

10.1.16   Something strange happened right now

You may have found an internal error in Hades. Try to cancel the current editor operation, then try to stop and restart the simulator. Often, this will restore things to a working condition. If not, try to save your current design (to a new file to avoid overwriting the last working version), then reload the design. If everything fails, or the simulation is very slow, quit the editor and restart it.

10.1.17   ghost components, ghost signals

For example, the editor may periodically display some animated simulation components, even after they have been deleted. This is due to a known bug in the simulator, but otherwise harmless. Just stop the simulator and restart it, and the ghost components and signals should be gone.

10.1.18   How can I disable the tooltips?

Depending on your operating system and window manager settings, the tooltips used by Hades may be useful or, indeed, annoying. This is especially true on Windows systems, where the toplevel window always gets the keyboard focus automatically. Unfortunately, there is no simple fix for this, because Hades uses the AWT (abstract window toolkit) library, whose support for popup windows is largely broken.

As a workaround, just disable the tooltip windows during editing via the *Editor ▷ Display ▷ enable tool-tips* menu item. The default (startup) value of the menu item can be set via the boolean `Hades.Editor.EnableToolTips` SetupManager property in your `.hadesrc` configuration files. Set `Hades.Editor.EnableToolTips false` to disable the tooltips.

10.1.19   Why is this object off-grid? Why won't the cursor snap to the object?

By default Hades starts with a magnetic grid of 1/4th inch, but some objects have ports on a 1/8th inch grid. See the explanation on page 34 on how to change or deactivate the magnetic grid setting.

10.1.20   Why can't I connect a wire to this port?

You are probably trying to connect a wiresegment to a port which is already connected to another signal. To connect the wiresegment use `<shift>`+click instead of a normal mouse click.

10.1.21   Hades won't let me delete an object

For *delete*, you must click exactly on the objects corner. Make sure that the magnetic grid is set correctly. Also, see the answer to the previous question about problems with off-grid objects. Workaround: try clicking on another corner of the object to delete.

10.1.22   Why don't the bindkeys work?

Sometimes, the editor won't react to the bindkeys described in section 5.2 on page 46. Due to "features" in the Java 1.1 keyboard focus policy on Windows systems, 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 should accept the bindkeys.

10.1.23   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 4.19 on page 42 for a detailed explanation.

10.1.24   Why won't the editor 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.

10.1.25   Why doesn't the cursor represent the editor state?

Yes, it would be good if the editor could change its cursor to reflect the current editing operation. Unfortunately, there is no way to change the cursor for Java 1.1 applications, and we felt that Java 1.1 (e.g. Microsoft VM) compatibility is still important for Hades. While the newer Java 2 specification allows applications to change the cursor, getting a decent interaction with the magnetic grid in the Hades editor is still impossible.

10.1.26   Operation X is slow

Given a modern PC or workstation and a current Java virtual machine, Hades should not feel slow overall. However, in many cases straightforwared but inefficient algorithms are used for some (infrequent) operations in Hades. For example, wire creation should be acceptably fast, while wire deletion is significantly slower. Also, a typical circuit schematic may easily contain several thousand graphical objects (hundreds of components and wires with a dozen of graphical subobjects each).

10.1.27   Remote X11-Display is very slow

The X11 window system allows to run the application code for an application on a remote computer, using the local system (the X-server) only for the redrawing, which works well for many X applications.

Unfortunately, this is not true for Java applications with the JDK up to and including version JDK 1.3.1. The problem is that the JDK allocates the buffers required for double-buffering on the X client application side and not on the X server with the display. Therefore, every

redraw operation has to transmit the whole buffered image over the network, resulting in very high network load, and crippling performance.

As Hades might request dozens of repaints per second, running it with a remote X11 display will saturate your network, but still without getting acceptable redraw performance. Often, running Hades on a slow machine locally will result in much better interactive performance than running it remote on a big compute server.

Sun has promised to fix the problem for JDK 1.4. However, due to the overhead of network access and the limited bandwidth, using a remote display will still be much slower than running Hades on your local machine.

10.1.28   The simulation is suddenly very slow

If the performance of the simulator suddenly drops down, there are a number of possible problems:

- Check whether your circuit is *oscillating*, probably due to direct feedback loops or feedback loops through transparent latches.

  This effect is very noticable in *real-time* simulation mode, when the simulator suddenly cannot keep up with real-time any more.

- Perhaps some problem occured during the simulation, and the simulator spends most of its time printing error messages. For example, a timing violation might have been detected, which in turn invalidated other signals. Another example is a microcontroller running a program which begins to execute illegal instructions.

  Check for error or warning messages via *Editor ▷ Special ▷ Show Messages* and the command shell (if visible).

- If you are using an older Java virtual machine, you may simply experience a slowdown due to inefficient garbage collection algorithms. For example, our profiling with JDK 1.1.6 indicated that the Hades simulator would spend up to 80% of total CPU time running the garbage collector, if the heap size would be larger than about 64 MBytes.

- The Microsoft VM seems to have a performance problem with buffering for the Java AWT TextArea component, which is used for the Hades message console. Whenever more than a few hundred lines are added to the console, performance drops rapidly. As a workaround, open the console with *Editor ▷ Special ▷ Show Message* and then press the *Clear* button from time to time.

- Due to caching of simulation model symbols, the editor can run out of memory, when a lot of design files are opened. As the maximum amount of memory for the JVM is usually fixed, less memory is left available for the simulation for each newly opened file. Selecting the *menu ▷ special ▷ flush symbol cache* function before opening a new design file can help to reclaim some memory at the expense of slower loading. If necessary, restart Hades from time to time.

10.1.29   GND, VCC, and Pullup components do not work

Due to the event-driven algorithm used by Hades, it is difficult to integrate *passive* simulation models into a running simulation. Therefore, you will have to stop and restart the simulation after adding GND, VCC, or Pullup components to your circuits, in order to re-initialize the simulation.

10.1.30   The simulator reports undefined values

If your circuit has signals that stay undefined or otherwise does not work correctly, check the following:

- If you have edited the circuit, check that all signals are initialized. For example, after integrating a VCC or GND component, stop and restart the simulator to (re-) initialize your circuit.

- Check for unconnected inputs and feedback circuits without proper reset.
- Check for hazards and setup- or hold-time violation on flipflops and memories.
- Check for undefined or short-circuit conditions on buses.
- Make sure that registers and memories are properly initialized via loading of initial memory contents, or via explicit initialization sequences.
- See section 10.2.4 for some tips on debugging.

10.1.31 How can I automatically restore editor settings?

First of all, try to set the corresponding values for the SetupManager properties in the `.hadesrc` configuration file. See section A for details.

If you want to customize other settings, you could use either a scripting language or write a small initialization class in Java.

10.1.32 My waveforms get overwritten?

Currently, all data for waveforms is stored in memory. As the simulator may easily execute 100.000 events per second, and the raw data for each event consists of a time value (8 bytes) and a data value (1 to 30 bytes), using waveforms may require more than one 1 MByte of memory per second.

While fresh memory is available, the waveforms will store all events. Once the available memory is exhausted, the waveform classes will periodically overwrite the oldest data in order to make room for the newer data. Due to the multithreaded nature of Java, even this algorithm can't always prevent premature OutOfMemoryErrors.

If you want to preserve the older waveform data, you should run the simulation only for some time, pause the simulation, and the save the waveform data to a file, before continuing with the simulation.

10.1.33 How can I edit a SimObject symbol?

As Hades does not have a graphical symbol editor, you will have to write the symbol file for your own simulation models with a text editor. See section 9.3 on page 86 for an explanation and an example.

10.2 Tips and tricks

10.2.1 What other programs are in hades.jar? How to run them?

One of the advantages of object-oriented programming and the Java object model is that every class can be used as an application, if only it provides its own `main()` method. Not surprisingly, many classes in Hades have their own `main()` and can be run as the main class. However, all classes are packed into the `hades.jar` archive file, which only provides a single main-class (namely, `hades.gui.Editor`).

To access and run the other classes inside `hades.jar`, you have to use a command shell to call the Java virtual machine with the corresponding parameters. When called without further parameters, most of the classes will print a short usage message.

Some of the useful classes inside `hades.jar`:

- `jfig.gui.Editor`   the JFIG graphics editor.
- `hades.models.pic.PicAssembler`   a command line assembler for the PIC 16 family of microcontrollers which creates raw `.rom` or Intel HEX `.hex` files. The assembler accepts many MPASM assembler files directly of with only minor changes.
- `hades.utils.NameMangler`   The tool used by Hades to encode Unicode characters for use in the `.hds` and `.sym` design and symbol files. Use the command line version to encode or decode strings manually.

- `hades.utils.vhdl.VHDLWriter`   A command line tool to convert Hades design files into a VHDL netlist, preserving the design hierarchy. Note that the converter cannot generate VHDL simulation models for most of the complex Hades simulation models. Currently, it only understands several gate-level models, including basic gates, complex gates, flipflops, and clock generator.

10.2.2   User-settings in .hadesrc

See the description in section 3.8 on how to customize the Hades editor and tools via the `.hadesrc` configuration files. Appendix A lists all configuration properties together with their default values.

10.2.3   How to enable or disable glow-mode for individual signals?

This is possible via the `h` bindkey in the editor. Move the cursor to a vertex of the signal in question, then press the `h` key to toggle glow-mode for that signal.

Naturally, you can also enable or disable glow-mode for each signal via explicit method calls to `signal.setGlowMode()`. See the examples in section 8 on page 71.

10.2.4   What can I do to debug my circuits?

- Activate glow-mode and interactive simulation to play with your circuit. Watch out for undefined signals (cyan and magenta colors).
- Add probes to all relevant signals and watch the signal traces in the waveform viewer. Check all signals which carry `U` or `X` values carefully. Also note that floating signals (`Z` value) are not a good idea. Consider adding pullup components to buses.
- Check the editor message console for all warning and error messages like flipflop timing violations, missing memory initialization data files, or even simulator internal errors.
- Select *Editor ▷ Special ▷ Check design* menu item and read the messages in the console window. The editor will report all unconnected (open) inputs, short-circuit outputs, etc.
- Use *design for test* methods, like including extra reset inputs, extra status-outputs, or built-in-selftest logic.
- Add *measurement equipment* to strategic signals in your circuit. For example, try a `hades.models.special.HazardTrigger` or a `hades.models.special.Counter` to detect hazards or to count events on a signal.
- As a last resort, try to single-step the simulator through your circuit, and periodically use the debug menu items like *Editor ▷ Debug ▷ Print eventlist* to print the internal simulator state. This will not usually be necessary, except when debugging your own Java-written simulation models.

10.2.5   I need a two-phase clock

To create a two-phase clock, just create two *ClockGen* components with the same clock period, 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.

10.2.6   How can I print my circuit schematics?

Just try the *Editor ▷ File ▷ Print* menu item. This will use the Java printing API to access the default printer(s) on your system. It will also change the zoom-factor of your current schematic to A4 landscape paper and request a full redraw. If you select landscape paper in the printer options dialog, your circuit should fit on the paper. Due to limitations in the Java printing API, however, the quality of the printing is not optimal. While texts and straight lines may be ok, arcs and spline objects are very jagged.

For high quality prints, you may prefer to export your schematics to FIG format, and then use the fig2dev converter to produce high-quality Postscript, see below.

10.2.7   Printing problems

Currently, Hades uses the Java 1.1 API to access a printer, which results in suboptimal print quality and is not very reliable. Also, there is a bug in the Hades graphics code, which prevents printing on some JDK 1.3 virtual machine, due to a conflict between Java2D enabled screen rendering and non-Java2D printer rendering. If you get exceptions while printing on a JDK 1.3 virtual machine, you might try to disable Java2D support via the `jfig.allowJava2D` SetupManager property in your `.hadesrc` file.

10.2.8   How can I export my circuit schematics via fig2dev?

As the Hades graphics editor is based on the JFIG graphics code, it is possible to export circuit schematics to the FIG file format, which can then also be edited with the `xfig` program available on many Unix/Linux systems, or the `jfig.gui.Editor` class included in the `hades.jar` archive.

Among the many programs that support the FIG format is the `fig2dev` format converter, which allows to convert FIG files into several image formats, HP GL, LATEX drawing commands, or high-quality Postscript. If `fig2dev` is not installed on your system, you can download the sources from `ftp.x.org`. Contact your system vendor for pre-built binaries for Unix and Linux systems. A binary for Windows is available from the JFIG home page, `tech-www .informatik.uni-hamburg.de /applets/jfig/`.

In the editor, select the *Editor ▷ Export ▷ Settings. . .* menu item, which opens the export options dialog. First, select whether you want color or black-and-white export (which might be better for printing on b/w printers). Next, fill in the remaining options and select *Export Now*.

10.2.9   I cannot initialize my circuit

For well designed circuits with proper reset logic, there should be no problem. However, if you try to simulate legacy circuits you may experience difficulties in getting your circuit to work. This is due to the simulation semantics and the industry standard nine-valued `std_logic` logic system used by Hades for gate-level simulations. All gates and flipflops will start and stay in the special U state, until all inputs are well-defined. Use switches or the `PowerOnReset` component to generate valid reset impulses after starting the simulation.

10.2.10   Simulation does not appear deterministic

This depends on your simulator settings and the simulation models used in your circuits.

When using the VHDL-based simulation kernel with its two-list algorithm (class `hades .simulator.VhdlSimKernel`), the simulation should be deterministic, as multiple instantaneous events are deferred until the next delta-cycle. Events occuring in one delta-cycle are executed in random order, but this should not influence the final simulation state.

On the other hand, when using the simpler one-list simulation kernels, the simulation state does depend on the (random) ordering of events. For example, a basic NOR-flipflop will never oscillate with this algorithm, because one gate will be evaluated first.

Note that some simulation models intentionally include random behaviour. For example, the metastable DFF flipflop will not generate U or X output values, when undefined input signals or a timing violation are detected. Instead, the metastable flipflop will enter a random 0 or 1 state after a random delay.

10.2.11   I took a schematic from a book, but the circuit does not work

Unfortunately, this can happen, and there are several reasons for it. First of all, it is very difficult to write a 100% error-free book, and it is even more difficult to get all the graphics and figures correct. Therefore, the schematic you copied may simply have a few small errors.

Second, figures usually serve to illustrate a certain idea. This in turn leads to a compromise between a nice and easily understandable figure, or a very complex figure with all the gory details. Many authors, especially in textbooks, prefer the readable variant.

Third, the function of a circuit often depends on technological constraints. For example, many legacy TTL circuits don't include complex reset logic, because real TTL devices will behave as expected by the designer. The same circuit in CMOS technology, however, may enter undefined states or even a latch-up.

Fourth, while the simulation models are written to accurately model the behaviour of real devices, there are certain differences. For example, the U and X states of the `std_logic` logic model are a simple means to model undefined logic states. However, circuits without proper reset logic will not be useable with `std_logic`, unless explicit manual initialization sequences are used.

While this is not recommended, you might use the special *metastable* flipflops models from the *hades.models.flipflops* package to model legacy circuits or other circuits with broken reset logic. These simulation components, *DffMetastable* and *LatchMetastable* behave differently from the usual *Dff* or *Latch* models. They never generate the undefined *X* or *U* output values, even when some inputs are undefined or a timing violation is detected. Instead, these flipflops use a random number to generate a random 0 or 1 output in such conditions, and another random number is used to generate a random gate-delay (with may be up to 50 times longer than the default gate-delay).

10.2.12   VHDL export

Netlist export of Hades designs to VHDL format is possible. This currently supports hierarchical designs and includes a name-mangling scheme which converts Hades signal and component names to unique but still readable VHDL names. Try running the `hades.utils`
`.vhdl.VhdlWriter` from the `hades.jar` archive and read the help message for the command syntax and options.

The `VhdlWriter` will write all `entity` declarations and default `architecture` blocks for all Hades schematics. It also includes support to write default `architectures` for gate-level components like simple and complex gates, several flipflops, clock generator and power-on-reset. Naturally, it can not translate arbitrary Java code to corresponding VHDL architectures.

So far, the export functions have only be tested with a few designs, and only with the Synopsys VSS simulator. Support for other simulators may require some changes in the name mangling.

10.3   Known bugs and features

10.3.1   How should I report bugs?

To report a bug send an email to `hendrich@informatik.uni-hamburg.de`.

However, you might first check that the bug is actually a Hades bug instead of a known problem of your Java virtual machine. For example, several redraw problems are known for Java applications when running with JDK 1.2.x or JDK 1.3.x. See the release documentation for your Java virtual machine for details.

Your mail should include your operating system and version, Java virtual machine and version, Hades version, CLASSPATH settings, .hadesrc property values, and the detailed error message, including stack traces. Use `java -version` to print out your JDK version information. (For example, a useful message might include the following information: Windows XP home, JDK 1.3.1_02, Hades 0.89b, no CLASSPATH set, default .hadesrc).

For multiple related errors, only report the first one. If possible, include all required `.hds` design files and detailed instructions on how to reproduce the bug or (even better) a script file which demonstrates it.

10.3.2   Spurious objects displayed

When deleting a component like the `hades.models.dcf77.Dcf77Clock` generator, which repeatedly updates itself, the redraw thread may keep a reference to the delete component, which will the re-appear on the editor canvas.

To avoid this problem, stop and restart the simulator after deleting such component(s).

10.3.3 Repaint algorithm

The redraw algorithm used by Hades relies on a separate Thread for asynchronous repaints. However, due to the overhead of object synchronisation in Java, the editor does not fully `synchronize` the simulation and the redrawing.

This means that the simulation continues to run even while a repaint is in progress. This may lead to inconsistent display for those objects, whose state changed during a repaint operation.

To get a static and fully consistent display, pause the simulation and wait for the next repaint operation. It is also possible to request a full repaint via the *Editor ▷ Edit ▷ Repaint all* menu item.

10.3.4 Repaint bugs, DirectDraw

To improve performance, Hades uses a complex buffering scheme for screen redrawing. Depending on the number of objects that require redrawing, either the whole offscreen buffer, parts of the offscreen buffer, or only the corresponding objects are drawn.

Unfortunately, this algorithm can trigger a known bug in JDK 1.3.x on Windows, where the JDK mixes Windows GDI and DirectDraw calls for redrawing. If you experience frequent redraw problems on a Windows system with JDK 1.3.x, try to call the Java virtual machine with the `java -Dsun.java2d.noddraw hades.gui.Editor` option. This will disable DirectDraw acceleration and should improve stability at the cost of performance.

On Linux and Solaris systems, the Java AWT toolkit is known for a lot of windowmanager related problems. For example, windows may appear in the wrong size, with zero size, or even outside the visible screen. The only known workaround in such cases is to try a different combination of Java virtual machine and window manager.

To repaint the whole editor user interface, select the *Editor ▷ Edit ▷ Repaint all* menu item, or press the CNTL-A bindkey on the main editor canvas.

10.3.5 How to get rid of an unconnected signal?

When opening a design file, the editor will check the design and detect several kinds of errors, including signals which are not connected to any simulation components. Unfortunately, such signals also don't usually have wire segments, and can therefore not be selected or deleted via the graphical editor. Currently, the only way to get rid of such unconnected signal is to edit the `.hds` design file with a text editor.

10.3.6 The 'run for' simulator command may deadlock

Due to issues with the Java Thread scheduling, the `hades.simulator.VhdlSimKernel` simulation kernel may have problems when using the *run for* mode. While the simulation will be paused after the specified simulation end time has been reached, it may be impossible to continue the simulation afterwards. This bug should be fixed in Hades version 0.9 or greater.

Workaround: please upgrade to the newest version of Hades. If the problem persists, try another Java VM for your system. Use the *run* simulation command instead of *run for*.

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

[IEEE 93b] *IEEE Standard Multivalue Logic System for VHDL Model Interoperability*, Standard IEEE 1164-1993, 1993

[Jython] The Jython language homepage, *www.jython.org*

[Hendrich 97] N. Hendrich, *HADES — A Java-based visual simulation environment*, 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

[Hennessy & Patterson] J.L. Hennessay and D.A. Patterson, Computer architecture, the hardware/software-interface, Morgan-Kaufmann, 199X

A   SetupManager properties

This section lists the user-settable properties for both the Hades editor and simulator and the JFIG graphics editor and library, together with useful default values.

A.1   Hades properties

For educational usage, you should probably activate glow-mode, RTLIB animation, and the simulator autostart properties. The Java2D antialising option is not generally useful during simulation, but can improve screenshots. Set the default window size and position corresponding to match the screen size of your computer:

```
# Hades Editor default properties
#
Hades.Console.SaveLines             600
Hades.Console.LogEnable     false
Hades.Console.LogfileName    /tmp/hades.log
Hades.Console.WindowInit            300 300 600 400
Hades.Console.ConsoleFontName       Courier
Hades.Console.ConsoleFontSize       12
Hades.Console.ButtonFontName        SansSerif
Hades.Console.ButtonFontSize        12
#
# use one of "VHDL", "Real time", "Batch mode", "Interactive":
Hades.Editor.DefaultSimKernel       VHDL
Hades.Editor.AutoStartSimulation    true
Hades.Editor.AutoZoomFit            true
Hades.Editor.DumpSystemProperties   false
Hades.Editor.EnableDebugMenu        false
Hades.Editor.EnableSelectionMenu    true
Hades.Editor.EnableTipOfTheDay      false
Hades.Editor.EnableToolTips         true
Hades.Editor.EnableVHDLMenu         false
#Hades.Editor.StartupFilename       /hades/examples/simple/jkff.hds
Hades.Editor.ToolTipsDelay          2000
Hades.Editor.PopupMenuResource      /hades/gui/PopupMenu.txt
#
# Java2D rendering hints
#
Hades.Editor.AntiAlias              false
Hades.Editor.RenderQuality          true
#
# the following items are disabled for foolproof operation;
# initialize signals works recursively and may usually lead to
# inconsistent data structures: use with care.
#
Hades.Editor.EnableInitializeSignalsMenu false
Hades.Editor.EnableStandardLatch false
Hades.Editor.EnableStandardDFF    false
#
# false is good for Windows, you may want true for Unix/X11:
Hades.Editor.FocusOnMouseEnter     false
#
Hades.Editor.GlowMode               true
Hades.Editor.InverseCanvas          false
Hades.Editor.RedrawMode             Buffered
Hades.Editor.RedrawDelayMillis      20
```

```
Hades.Editor.WindowWidth          700
Hades.Editor.WindowHeight         500
Hades.Editor.WindowY              80
Hades.Editor.WindowX              50
#
Hades.Editor.MenuFontSize         12
Hades.Editor.MenuFontName         Helvetica
Hades.Editor.HighlightColor       0x00FF5500
#
Hades.FileDialog.FontName         SansSerif
Hades.FileDialog.FontSize         12
#
# default redraw frequency
#
Hades.SyncRedrawTimer.UpdateInterval     50
#
Hades.Exporter.FontName           SansSerif
Hades.Exporter.FontSize           12
Hades.Exporter.Fig2devOptions     "-p portrait -m 1.0"
Hades.Exporter.EPSCommand         /usr/X11/bin/fig2dev -L ps -P
Hades.Exporter.PSCommand          /usr/X11/bin/fig2dev -L ps -P
Hades.Exporter.PrintCommand       lpr
Hades.Exporter.PrintFitToA4 true
#
# a "black on green" LCD:
Hades.GraphicsLCD.repaintMillis  100
Hades.GraphicsLCDCanvas.lcdBackground     0x00bfdf00
Hades.GraphicsLCDCanvas.lcdInactivePixel 0x00b5d200
Hades.GraphicsLCDCanvas.lcdActivePixel   0x00000000
#
Hades.TextLCDCanvas.lcdBackground     0x00bfdf00
Hades.TextLCDCanvas.lcdInactivePixel 0x00b5d200
Hades.TextLCDCanvas.lcdActivePixel   0x00000000


Hades.LayerTable.DisplayInstanceBorder    true
Hades.LayerTable.DisplayInstanceLabels    false
Hades.LayerTable.DisplayClassLabels       false
Hades.LayerTable.DisplayPortSymbols       true
Hades.LayerTable.DisplayPortLabels        true
Hades.LayerTable.DisplayBusPortSymbols    true
Hades.LayerTable.RtlibAnimation           true
#
# UseSelectDialog true=Netscape/MSIE style
#                 false=traditional FileDialog
#
Hades.DesignManager.Debug false
Hades.DesignManager.VerboseMessages true
Hades.DesignManager.UseSelectDialog false
Hades.DesignManager.AutoAddExtension     true
#
#
Hades.PropertySheet.backgroundColor      0x00d0d0d0
#
# enable EventNode recycling to help out the garbage-collector?
#
Hades.Simulator.EventList.EnableRecycling  true
Hades.Simulator.EventList.RecycleCapacity  1001
Hades.Simulator.SleepMillis      100
```

```
Hades.TipOfTheDay.ButtonFontSize            12
Hades.TipOfTheDay.ButtonFontName            SansSerif
Hades.TipOfTheDay.TextAreaFontSize          12
Hades.TipOfTheDay.TextAreaFontName          SansSerif
#
# default glow-mode colors for StdLogic1164 values
#
Hades.StdLogic1164.Color._U            0x0000ffff
Hades.StdLogic1164.Color._X            0x00ff00ff
Hades.StdLogic1164.Color._0            0x00dcdcdc
Hades.StdLogic1164.Color._1            0x00e00000
Hades.StdLogic1164.Color._Z            0x00ffBf00
Hades.StdLogic1164.Color._W            0x00ff7f00
Hades.StdLogic1164.Color._L            0x00404040
Hades.StdLogic1164.Color._H            0x00c00000
Hades.StdLogic1164.Color._D            0x007f7f7f
#
Hades.HexTextField.FontSize 16
Hades.HexTextField.FontStyle 1
Hades.HexTextField.BackgroundColor  0x00afafaf
Hades.HexTextField.TextColor        0x007f0000
#
Hades.MemoryHexEditorField.RepaintMillis 100
Hades.MemoryHexEditorField.FontName     Courier
Hades.MemoryHexEditorField.FontSize 12
Hades.MemoryHexEditorField.FontStyle 0
Hades.MemoryHexEditorField.BackgroundColor      0x00efefef
Hades.MemoryHexEditorField.DataColor            0x000000af
Hades.MemoryHexEditorField.ActiveDataColor      0x00ef0000
Hades.MemoryHexEditorField.PassiveDataColor     0x00afafaf
Hades.MemoryHexEditorField.AddrColor            0x00af0000
Hades.MemoryHexEditorField.EditColor            0x00ff8000
Hades.MemoryHexEditorField.ReadHighlightColor   0x0000bf00
Hades.MemoryHexEditorField.WriteHighlightColor  0x00ef00cf
Hades.MemoryHexEditorField.EnableTooltips false
#
# specify VT52 font and color
# amber:
hades.VT52Canvas.color        0x00f0a000
# perhaps you prefer green:
#hades.VT52Canvas.color       0x0000e000
hades.VT52Canvas.fontName     Courier
hades.VT52Canvas.fontSize     10
#
Hades.LED.Color.red               0x00ff0000
Hades.LED.Color.green             0x0000ff00
Hades.LED.Color.yellow            0x00ffff00
Hades.LED.Color.blue              0x000000ff
#
Hades.WaveformViewer.DefaultWidth                   800
Hades.WaveformViewer.DefaultHeight                  700
Hades.WaveformViewer.NameCanvas.BackgroundColor     0x00d0e0e0
Hades.WaveformViewer.WaveCanvas.BackgroundColor     0x00e0f0f0
Hades.WaveformViewer.WaveCanvas.XORColor            0x00ffffff
#
```

A.2   jfig default properties

The JFIG graphics libraries are used by Hades for the schematics editor with the SimObject symbols and wires. Most of the following properties are not directly useful when running Hades. However, you might want to modify these properties in case you plan to use the JFIG editor from the `hades.jar` archive. To run the JFIG editor, `jfig.gui.Editor` as the main class with a command line similar to

`java -classpath c:\temp\hades.jar jfig.gui.Editor`

If you experience printing problems with Hades on JDK 1.3, you may want to try the `jfig.allowJava2D false` setting.

```
#jfig default properties
jfig.FIG.Version                        32
jfig.Java2D.AntiAlias                   false
jfig.Java2D.RenderingQuality            true
jfig.allowJava2D                        true
jfig.cursorSnapping                     1/4 grid
jfig.enableXSplines                     true
jfig.grid                               coarse grid
jfig.gui.Canvas.RedrawMessages          false
jfig.gui.Canvas.RequestFocusOnMouseEnter  false
jfig.gui.Console.ButtonFontName         SansSerif
jfig.gui.Console.ButtonFontSize         12
jfig.gui.Console.ConsoleFontName        MonoSpaced
jfig.gui.Console.ConsoleFontSize        12
jfig.gui.CreateImageDialog.LayerCorrection  true
jfig.gui.EditDialog.FontName            SansSerif
jfig.gui.EditDialog.FontSize            12
jfig.gui.Editor.AttribsScrollPaneEnable  false
jfig.gui.Editor.DebugMenuItems          false
jfig.gui.Editor.DebugRedrawMessages     false
jfig.gui.Editor.Icon                    /jfig/images/icon.gif
jfig.gui.Editor.MenuFontName            SansSerif
jfig.gui.Editor.MenuFontSize            12
jfig.gui.Editor.WindowHeight            700
jfig.gui.Editor.WindowWidth             900
jfig.gui.ExportOptionsDialog.Autosave   true
jfig.gui.ExportOptionsDialog.Debug      false
jfig.gui.ExportOptionsDialog.EnableGIF  false
jfig.gui.ExportOptionsDialog.FontName   Times
jfig.gui.ExportOptionsDialog.FontSize   6
jfig.gui.ExportOptionsDialog.Print      lpr
jfig.gui.ExportOptionsDialog.WaitExec   true
jfig.gui.ExportOptionsDialog.fig2devEPS  -L ps
jfig.gui.ExportOptionsDialog.fig2devGIF  -L gif
jfig.gui.ExportOptionsDialog.fig2devIBMGL  -L ibmgl
jfig.gui.ExportOptionsDialog.fig2devJPG  -L jpeg
jfig.gui.ExportOptionsDialog.fig2devLATEX  -L latex
jfig.gui.ExportOptionsDialog.fig2devMagnification  1.0
jfig.gui.ExportOptionsDialog.fig2devPICTEX  -L pictex
jfig.gui.ExportOptionsDialog.fig2devPNG  -L png
jfig.gui.ExportOptionsDialog.fig2devPPM  -L ppm
jfig.gui.ExportOptionsDialog.fig2devPS  -L ps -P
jfig.gui.ExportOptionsDialog.fig2devPSCentered  true
jfig.gui.ExportOptionsDialog.fig2devPSOrientation  true
jfig.gui.ExportOptionsDialog.fig2devPath  /usr/X11/bin/fig2dev
jfig.gui.KeyHandler.BreakCompound       G
jfig.gui.KeyHandler.CopyObject          C
jfig.gui.KeyHandler.CreateArc           r
```

```
jfig.gui.KeyHandler.CreateCircle        c
jfig.gui.KeyHandler.CreateClosedBezier  I
jfig.gui.KeyHandler.CreateClosedSpline  S
jfig.gui.KeyHandler.CreateCompound      g
jfig.gui.KeyHandler.CreateEllipse       e
jfig.gui.KeyHandler.CreateImage         J
jfig.gui.KeyHandler.CreateOpenBezier    i
jfig.gui.KeyHandler.CreateOpenSpline    s
jfig.gui.KeyHandler.CreatePolygon       p
jfig.gui.KeyHandler.CreatePolyline      l
jfig.gui.KeyHandler.CreateRectangle     b
jfig.gui.KeyHandler.CreateText          t
jfig.gui.KeyHandler.CutPoint            D
jfig.gui.KeyHandler.DeleteObject        d
jfig.gui.KeyHandler.EditObject          E
jfig.gui.KeyHandler.InsertPoint         a
jfig.gui.KeyHandler.MirrorX             F
jfig.gui.KeyHandler.MirrorY             f
jfig.gui.KeyHandler.MoveObject          m
jfig.gui.KeyHandler.MovePoint           M
jfig.gui.KeyHandler.NextCachedAttributes  n
jfig.gui.KeyHandler.OpenCompound        o
jfig.gui.KeyHandler.SaveAttributesToCache  N
jfig.gui.KeyHandler.ScaleObject         $
jfig.gui.KeyHandler.ToggleShowGrid      W
jfig.gui.KeyHandler.UpdateObject        u
jfig.gui.KeyHandler.ZoomIn              Z
jfig.gui.KeyHandler.ZoomOut             z
jfig.gui.PresentationViewer.Icon        /jfig/images/icon.gif
jfig.gui.PresentationViewer.WindowHeight  600
jfig.gui.PresentationViewer.WindowWidth  800
jfig.gui.PresentationViewer.debug       false
jfig.gui.PresentationViewer.enablePageNames  false
jfig.gui.PresentationViewer.enablePageNumbers  true
jfig.gui.SelectFromLibraryDialog.BaseDir  /tmp/jfig/libraries
# default on X11: /usr/X11R6/lib/X11/xfig/Libraries
jfig.gui.Viewer.Icon                    /jfig/images/icon.gif
jfig.gui.Viewer.WindowHeight            600
jfig.gui.Viewer.WindowWidth             800
jfig.gui.Viewer.debug                   false
jfig.objects.Exporter.DumpStatus        false
jfig.objects.FigImage.verboseImageUpdate  false
jfig.objects.FigParser.UseFastMessages  false
jfig.objects.FigParser.debug            false
jfig.objects.FigParser.enableMessages   false
jfig.objects.FigParser.enableNonASCII   true
jfig.pageJustification                  Center
jfig.pageOrientation                    Portrait
jfig.paperSize                          A4
jfig.units                              Metric
jfig.utils.MouseMapper.Remap            false
jfig.utils.MouseMapper.ShiftClickIsMiddleClick  true
```

B Index