MIMOS Tool

Release 1.0

User Manual

March 2, 2026

Contents

Chapter 1

Introduction

MIMOS-Tools is a software for designing, analyzing, simulating and transforming **MIMOS** models [5] to final implementation, enabling *model-based* development of embedded systems. Main target applications include safety-critical systems where correct functional and timing behaviors are vital. The tool provides a seamless path to safe system development, ensuring that verified functions maintain the same behavior in the final implementation. This is achieved through automated stages for static analysis, functional verification, simulation, scheduling and timing analysis, and code generation for target platforms.

1.1 An Overview

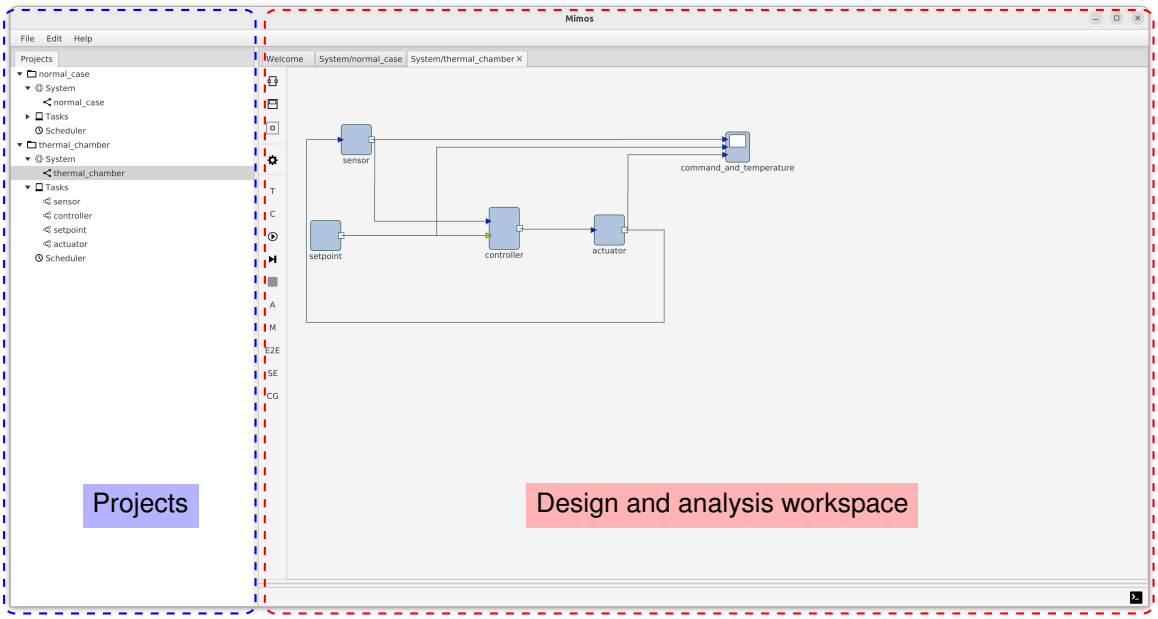


Figure 1.1: Tool's main window

As shown in Figure 1.1, the main window consists of two panels: the left lists currently open projects, and the right provides a graphical workspace for design and analysis.

The tool supports a layered design flow that separates functional correctness from timing concerns:

1. **Functional Design and Analysis**: Designers focus on system functionality by modeling the system as a network of tasks and channels, performing static analysis to detect

deadlocks and buffer overflows, and verifying functional correctness through simulation and symbolic execution. The deterministic semantics of **MIMOS** ensure that functional properties are preserved throughout the design flow.

2. **Timing Design and Analysis**: After functional verification, designers address real-time requirements by assigning timing parameters, performing schedulability analysis, and analyzing end-to-end latency. Changes to timing parameters do not affect validated functional behavior.

3. **Code Generation and Deployment**: The validated model is automatically transformed into executable code for target platforms (e.g., Real-Time Linux), preserving both functional and timing properties.

This separation allows designers to explore different timing configurations and scheduling policies without invalidating functional correctness, supporting efficient design-space exploration for safety-critical systems.

1.2 Project Structure and Management

1.2.1 Creating, Opening, and Saving Projects

To create a new project, select **File** → **New Project**. To open an existing project, choose **File** → **Open Project**, navigate to the project directory, and double-click the `project.ini` file (Figure 1.2). The tool does not support auto-save; to persist changes at any time, press **Ctrl+S** or use **File** → **Save Project**.

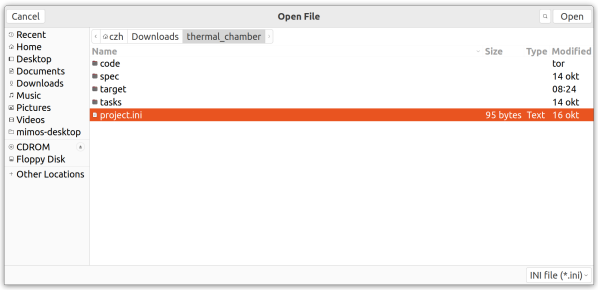


Figure 1.2: Opening a MIMOS project by selecting the `project.ini` file.

1.2.2 Project Directory Structure

Each MIMOS project follows a standard directory structure to organize different aspects of the system design.

- **code**/: Contains simulation code (described in Chapter 2).
- **spec**/: Contains JSON files describing the MIMOS network specification, including nodes, channels, and their configurations (described in Chapter 2).
- **tasks**/: Contains XML files describing real-time DAG task structures, including node parameters, dependencies, and timing constraints (described in Chapter 3).
- **target**/: Contains generated executable code for deployment on Real-Time Linux platforms (described in Chapter 4).

Chapter 2

Functional Design

At the functional design level, the system behavior is described and verified independently of timing concerns. The **MIMOS** tool provides comprehensive support for graphical modeling, static analysis, simulation, and formal verification of functional correctness.

To begin modeling, open a project, double-click on the **Graphical** icon in the left-hand side tree view to open the design workspace.

2.1 System Modeling

2.1.1 Graphical Interface

The graphical modeling environment enables users to construct **MIMOS** systems through an intuitive visual interface. As shown in Figure 2.1, systems are designed as networks of nodes (tasks) connected by channels.

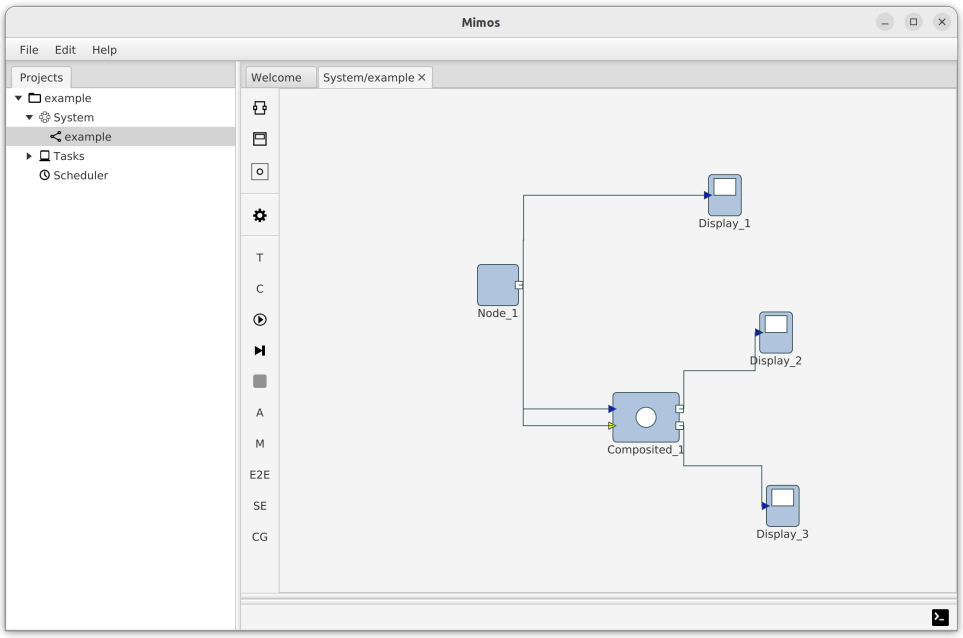


Figure 2.1: Graphical modeling interface showing a simple **MIMOS** network

The tool supports two types of nodes:

- **Computation Nodes** ( ): Periodic tasks that perform computations. These are the primary building blocks of a **MIMOS** model.

- **Display Nodes** ( ): Special nodes used for visualizing outputs. These nodes do not affect system behavior.

Nodes can be added by clicking the respective icon in the toolbar and placing them in the design area. Channels are created by clicking on a node's output port and dragging to another node's input port.

2.1.2 Node Properties

Node properties are configured through the Properties panel, accessible by double-clicking a node or right-clicking and selecting "Properties". Figure 2.2 shows the configuration interface.

Figure 2.2: Node properties configuration panel

Basic Parameters

- Name: Each node must have a unique name within the system. Names must conform to valid identifier syntax (alphanumeric characters and underscores, starting with a letter) as they are used in code generation.
- Period: The time interval between consecutive releases of the node.
- Deadline: The relative deadline by which the node's execution must complete. Outputs are written to output channels at the deadline instant.
- Offset: The time offset for the first release of the node.

Input Channels

Inputs are specified in a table with columns for *Name*, *Data Type*, *Port Type*, and *Initial Value*. You can add, delete, or modify ports directly in this table. To modify or delete an existing port, simply select the corresponding row with your mouse to edit or remove it. The *Port Type* determines whether the channel is a Register or FIFO channel. In the graphical interface, FIFO channels are indicated by blue triangles and register channels by yellow triangles. The channel semantics are determined by the *Port Type* and the *Data Type* syntax:

- **Register channels**: Sample the latest value without consuming it. The data type can be any primitive type (`int`, `double`, `boolean`, etc.). Reads are always enabled. Register channels must be initialized with exactly one value.

- **FIFO channels with blocking read**: Specified using a primitive type (e.g., `int`) or fixed-size array syntax (e.g., `int[5]`). The node requires exactly the specified number of tokens to execute (1 token for `int`, 5 tokens for `int[5]`). If insufficient tokens are available at release, the activation is skipped.
- **FIFO channels with up-to-N read**: Specified using bounded array syntax, e.g., `int[<=5]`. The node consumes up to 5 tokens if available, otherwise fewer or none. This non-blocking read is useful for handling rate mismatches.
- **FIFO channels with unbounded read**: Specified as `int[]`. The node consumes all available tokens from the FIFO. This is also a non-blocking read.

FIFO channels may be initialized with any number of tokens (including zero) using semicolon-separated values in the *Initial Value* field (e.g., `1;2;3`).

Output Channels

Outputs are specified with *Name* and *Data Type*. The data type syntax determines how many tokens are written:

- For FIFO channels: Array syntax defines the number of tokens written per execution (e.g., `int[3]` writes three tokens, `int` writes one token).
- For register channels: A single value overwrites the previous value.

Output channels may be connected to multiple consumers, effectively copying the data to all connected inputs.

2.1.3 Task Function Implementation

For each node, the tool generates a code template based on the declared inputs and outputs. The template includes:

- Class definition (for object-oriented languages)
- Input and output variable declarations
- Getter and setter methods for inputs
- A `run()` method containing the core computation

Accessing Code Generation:
There are two ways to access the code generation interface:

1. **Right-click menu**: Right-click on a node and select **Code Generation**, then choose:
   - **Java**: Generate Java code template
   - **C++**: Generate C++ code template
   - **Natural Language**: Open AI-assisted code generation dialog (described below)
2. **Properties panel**: In the node properties panel, click the code editor icon to view and edit the generated code

Figure 2.3 shows an example template generated for a node with one integer input and one integer output.

Users implement the node functionality by filling in the `run()` method. The tool currently supports two programming languages:

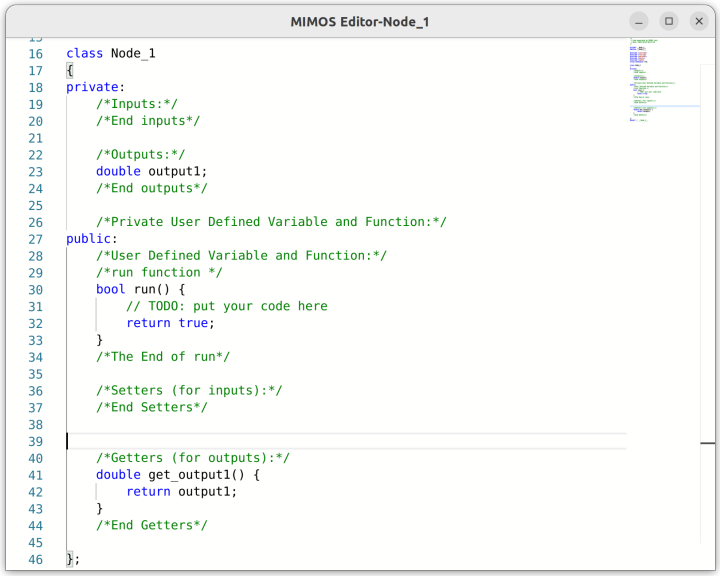


Figure 2.3: Generated code template for node implementation

- **C++**: For performance-critical components
- **Java**: For rapid prototyping and development

Different nodes within the same system can be implemented in different languages, with the tool handling cross-language integration during code generation.

2.1.4 Composite Nodes

For complex functionality, the tool supports **composite nodes** that internally decompose a single task into a directed acyclic graph (DAG) of sub-functions. This composition enables:

- Explicit representation of internal data dependencies
- Exploitation of internal parallelism during scheduling and code generation

Importantly, composite nodes are *functionally transparent*: from the perspective of other tasks in the system, a composite node behaves identically to an atomic node. Its activation depends solely on the enabling conditions of its input channels at the release instant. The internal DAG structure does not affect when the node becomes enabled or when it produces outputs.

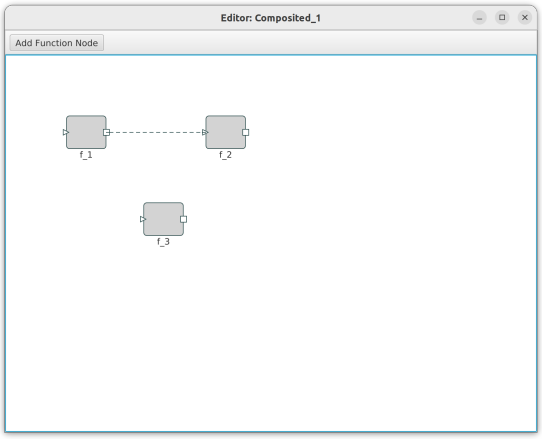
Double-clicking the composite node opens a DAG editor (Figure 2.4a) where sub-functions can be added using the "Add Function Node" button and connected to form a DAG.

The tool automatically generates a `run()` function that executes the DAG nodes in topologically sorted order, respecting their data dependencies (Figure 2.4b). Users only need to implement the individual DAG node functions (e.g., `f_1()`, `f_2()`, `f_3()`), each of which defines the computation for a single sub-function. The execution order of these functions is handled automatically by the generated `run()` method.

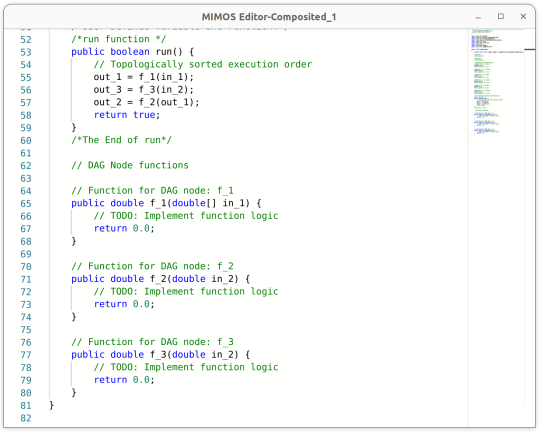
2.1.5 AI-Assisted Code Generation

Beyond manual implementation, the tool provides **natural language specification** of task functions using large language models (LLMs). Users can describe the desired functionality in plain language, and the tool automatically generates the implementation.

The AI-assisted generation is available for both:



(a) DAG editor showing internal structure

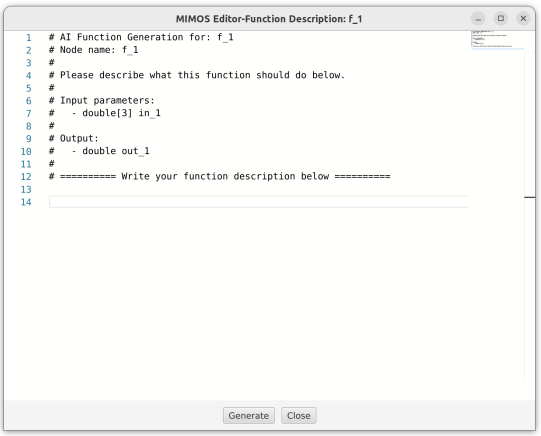


(b) Generated code with DAG node functions

Figure 2.4: Composite node design: (a) Internal DAG structure with function nodes f_1, f_2, and f_3, (b) Auto-generated code template where users implement individual DAG node functions

- **Regular nodes**: Generate the complete task function implementation
- **Composite node DAG functions**: Generate individual sub-function implementations (e.g., `f_1()`, `f_2()`, `f_3()`)

(a) AI generation for a regular node



(b) AI generation for a DAG function node

Figure 2.5: AI-assisted code generation interface: (a) Generating code for a regular task node, (b) Generating code for an individual DAG function within a composite node

To use AI-assisted generation:

1. Right-click on a node and select **Code Generation** → **Natural Language**
2. Describe the desired functionality in natural language in the provided text area
3. Optionally specify the target programming language (C++ or Java) in the description
4. Click the "Generate" button

The tool automatically recognizes the input and output parameters of the function and provides them in the description template, allowing users to focus on describing the computation logic.

2.1.6 Hierarchical Modeling

For complex systems containing many nodes, the tool supports **hierarchical modeling** through node grouping. This feature allows to organize related nodes into layers, simplifying the visual representation while maintaining all underlying connectivity and behavior.

Creating a Group

Grouping nodes into a hierarchical layer involves selecting multiple nodes and combining them into a composite structure. To create a group:

1. **Select multiple nodes** using one of these methods:
   - Hold **Ctrl** and left-click on each node want to group or
   - Click and drag to draw a selection box around multiple nodes (Figure 2.6)
2. **Create the group** by pressing **Ctrl+G**
3. A new composite node is created containing the selected nodes as an internal layer (Figure 2.7)

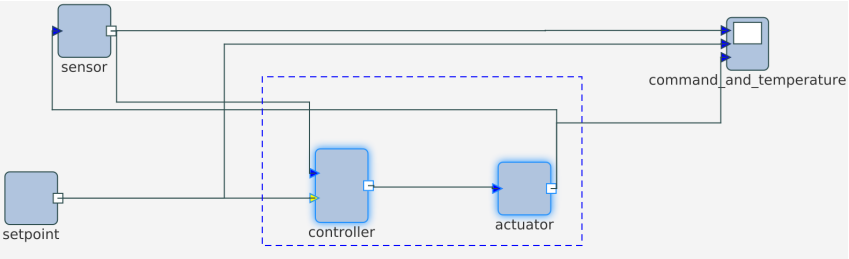


Figure 2.6: Selecting multiple nodes for grouping.

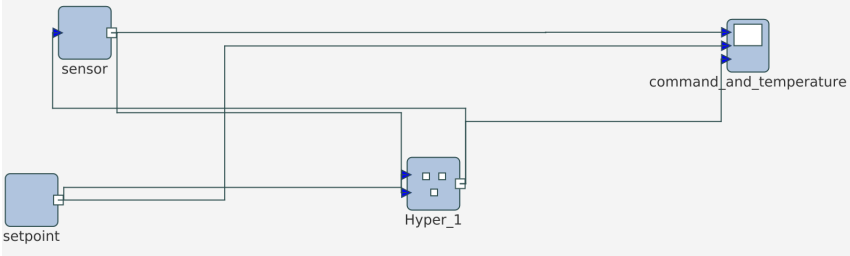


Figure 2.7: After grouping, the selected nodes are encapsulated into a single node.

The grouped nodes are represented as a single node in the parent view. Double click that node to view and edit its internal structure.

2.1.7 Code Libraries and Reusability

The MIMOS tool supports the use of external libraries in node implementations, enabling code reuse and integration with third-party functionality.

Managing External Libraries

To configure external libraries for your project, right-click on the project name in the left panel and select **Libraries Settings** from the context menu to open the Libraries Settings dialog (Figure 2.8, left).

For C++ libraries, set the Path to the directory containing the library files (which should contain an `include` subfolder). For Java libraries, set the Path to the absolute path of the JAR file.

Creating and Reusing User-Defined Libraries

The tool also supports creating reusable code libraries from node implementations. To save a node's implementation as a library component, right-click on any node with implemented functionality and select **Save in Libraries** from the context menu. The node's implementation will be saved as a reusable library component.

To reuse saved libraries in other nodes, open the **Library Browser** by selecting it from the file menu. The Available Libraries panel (Figure 2.8, right) displays all saved library components. User can then drag a library component from the browser onto the canvas to use it in the system.

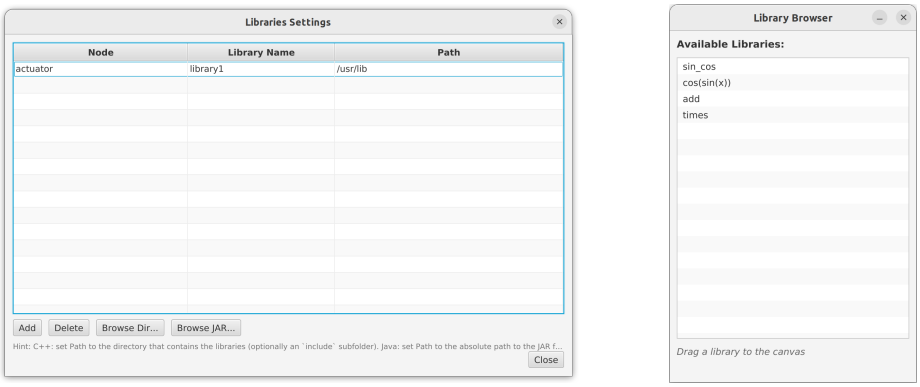


Figure 2.8: Libraries management: (left) Libraries Settings dialog for configuring external libraries, (right) Library Browser showing available user-defined libraries

2.2 Static Analysis

The tool performs static analysis to formally verify two fundamental data-flow properties: **deadlock-freedom** and **overflow-freedom**. To invoke the analysis before simulation or deployment, click the **Static Analysis** button in the toolbar. The analysis reasons about long-run token production and consumption using only the network structure, task periods and deadlines, channel read policies, and initial tokens.

If the analysis detects issues, the tool provides diagnostic information indicating:

- Which tasks are involved in deadlocks
- Which channels will overflow

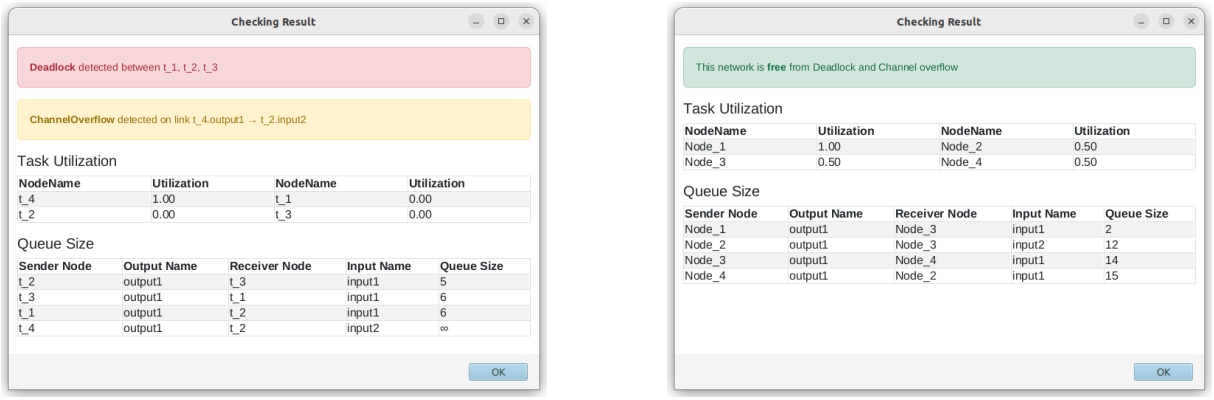
If no issues are found, the tool produces actionable design metrics:

- Task Utilization: For each task, the ratio of successful activations to total release instants over the long run.
- Queue Size Bounds: For each FIFO channel, the peak number of tokens that can accumulate during execution.

2.3 Simulation and Monitoring

2.3.1 Simulation Execution

To run a functional simulation:



(a) Issues detected (b) Validation passed

Figure 2.9: Static analysis results showing (a) detected issues and (b) validated system

1. Ensure all nodes have implemented task functions
2. Add display nodes to observe outputs of interest
3. Click the simulation settings button in the toolbar to configure simulation parameters
4. In the configuration window (Figure 2.10), specify:
   - Simulation length: The duration of the simulation in time units
   - Simulation Mode: Sequential or Parallel execution
5. Click the simulate button in the toolbar to start the simulation
6. After simulation completes, double-click on any display node to view its visualization window

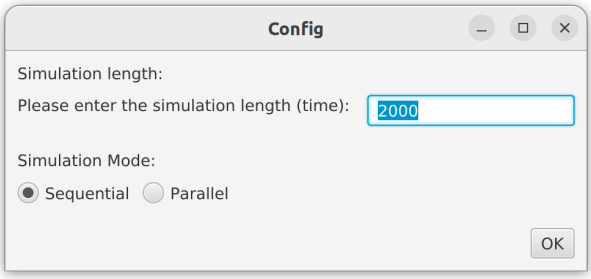


Figure 2.10: Simulation configuration window

Figure 2.11 shows the visualization window for a display node. The window consists of two panels:

- **Upper panel**: Displays a detailed view of the selected time region, showing precise waveform data
- **Lower panel**: Shows an overview of the entire simulation timeline. Users can select a specific region (highlighted in blue) by clicking and dragging to zoom into that time interval in the upper panel

The visualization window also provides interactive controls for stepping through the simulation, adjusting playback speed, and exporting results to CSV format.

The tool provides visual feedback on progress in the bottom-right corner. When the progress bar disappears, it means the simulation is complete.

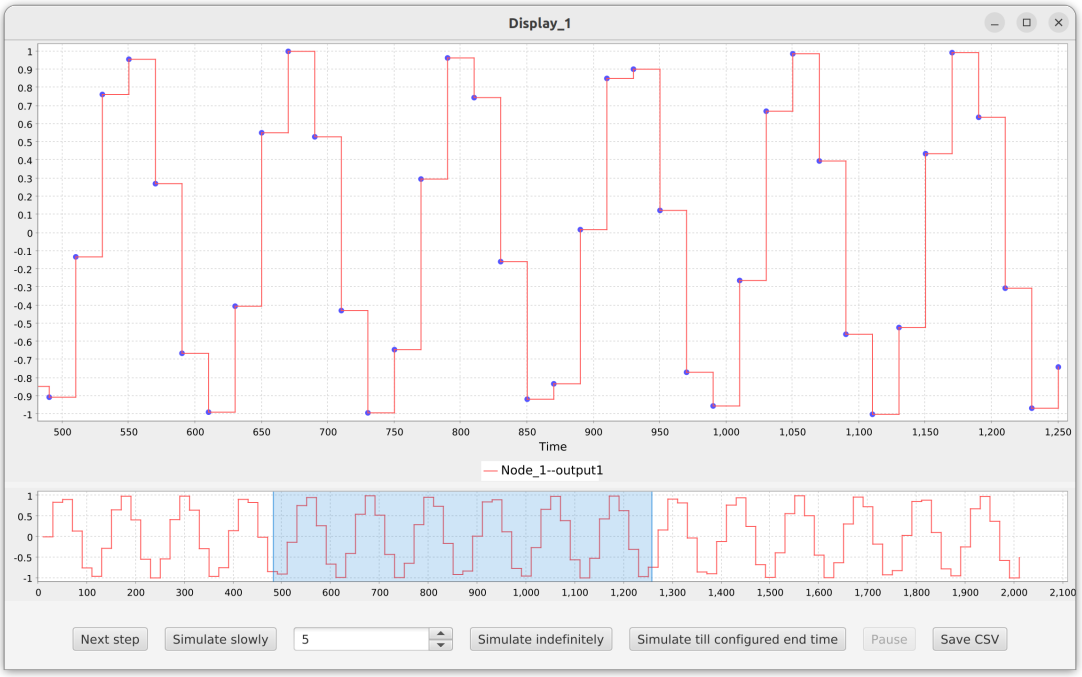


Figure 2.11: Functional simulation with waveform visualization.

2.3.2 Predicate Monitoring

During simulation, the execution trace can be evaluated against user-defined **predicates** (also called **assertions**) to verify functional correctness properties. A predicate is a boolean-valued expression that examines node inputs or outputs at each time instant.

Defining Predicates: Predicates are defined in the node properties panel under the **Assertion** section (Figure 2.12). To add a predicate:

1. Open the node properties window
2. In the **Assertion** section, specify:
   - **Name**: A descriptive identifier for the predicate (e.g., `P1`)
   - **Expression**: A boolean expression referencing node inputs or outputs (e.g., `output1 > 0.5`)
3. Click the **Add** button to create additional predicates

Figure 2.12: Defining predicates in the node properties panel

Viewing Monitor Results: After running a simulation, click the monitor button M in the toolbar to open the Predicate Monitor window. The monitor displays a timeline for each predicate, showing when it evaluates to true (high signal) or false (low signal) throughout the simulation (Figure 2.13).

The monitor window provides:

- **Time range**: The overall simulation duration
- **Current time**: A red vertical line indicating the current playback position
- **Timeline visualization**: For each node and predicate, showing boolean values over time
- **Playback controls**: Pause/resume and speed adjustment for stepping through the trace

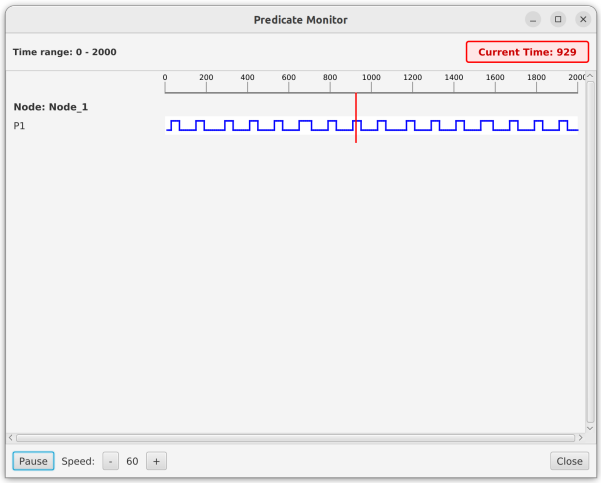


Figure 2.13: Predicate Monitor window showing the evaluation timeline for predicate .

Violations can be investigated by pausing at specific time instants and examining the predicate values.

2.4 Symbolic Execution

While simulation explores a single execution trace, **symbolic execution** systematically explores *all possible execution traces* by reasoning about symbolic values. A symbolic state represents multiple concrete states simultaneously, enabling exhaustive exploration of system behavior.

To perform symbolic execution, click the Symbolic Execution button SE in the toolbar. Figure 2.14 shows the symbolic state graph visualization. Blue nodes represent symbolic states, edges represent task executions. The right panel displays detailed information about the selected state, including time, task states, and channel contents.

The exploration continues until either all reachable states are covered (completeness) or the specified state limit is reached.

Temporal Property Verification

The state graph can be used to verify **temporal logic properties**. The tool currently supports a subset of Computation Tree Logic (CTL), similar to UPPAAL's property specification language [2].

Property Syntax:

Properties combine path quantifiers with state predicates:

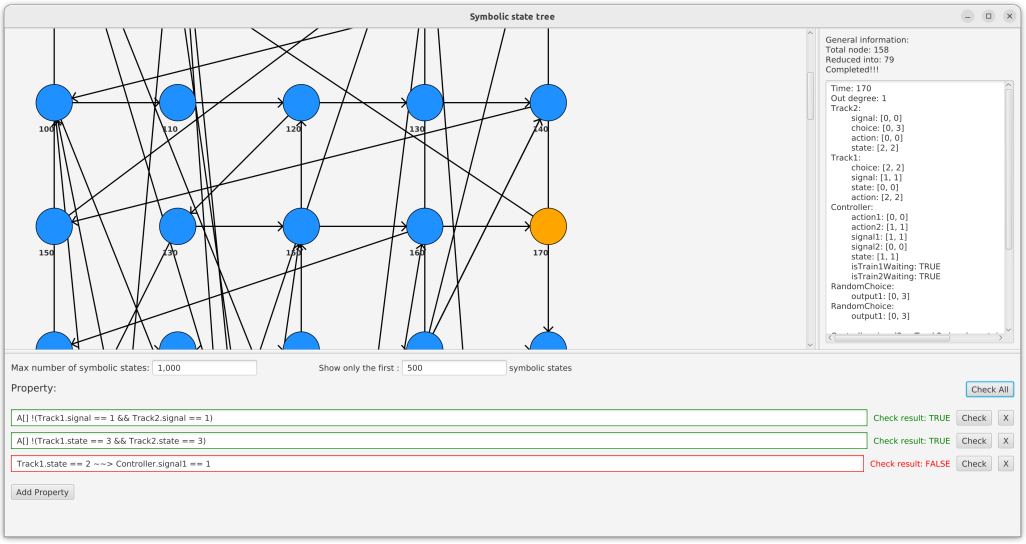


Figure 2.14: Symbolic state graph interface showing reachable states and state information panel

- `AG`(A[]): "Always Globally" – property holds in all states on all paths
- `EF`(E<>): "Eventually" – property is reachable on some path
- `AF`(A<>): "Always Eventually" – property eventually holds on all paths
- `EG`(E[]): "Exists Globally" – property holds in all states on some path
- `~~>`: "Leads to" – if first condition holds, second condition eventually follows

State predicates are boolean expressions over task states and channel contents.

Chapter 3

Timing Design

At the level of functional design, a **MIMOS** model is designed to implement a set of system functions. In general, each output channel corresponds to one function mapping a set of input streams to the output stream on the output channel. A system function may have to satisfy an end-to-end latency constraint, imposed by the application. The execution rates assigned to nodes should guarantee that the end-to-end latency constraints are met under the assumption that the nodes are all scheduled according to the execution rates and completed within the assigned deadlines, implying that the system is schedulable. **Time Design** is to design a scheduling policy for a system and a target platform such that the system is schedulable.

The **MIMOS** approach is to abstract the software of a system as a set of real-time tasks described using re-occurring (or periodic) DAG's and the target platform as either a set of parameters representing the number of processor cores and overheads for memory access, context switch and communication.

3.1 Real-Time DAG Tasks

The **MIMOS** tool provides the facility to design the structure of real-time tasks (as a DAG) and performing timing analysis.

The tasks in a project are listed under the `Tasks` item in the left-side tree view (see Figure 3.1). By default an empty task exists in the list. By **double-clicking** on the task, a tab is opened (right-hand side of Figure 3.1) to specify the task structure using the graphical interface, and configure its parameters.

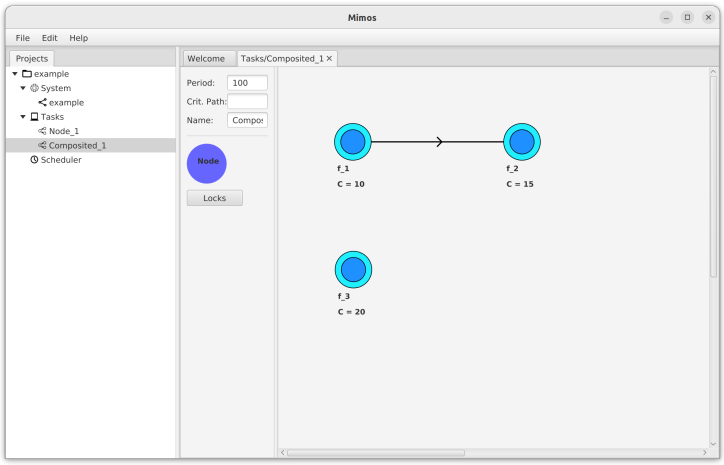


Figure 3.1: Task design.

As seen in Figure 3.1, the name and period of the task can be specified in the left-hand side bar. The filed *Critical Path* indicates the total WCET of the longest path in the task graph according to the user-specified parameters for the DAG. This parameter is calculated automatically.

In addition, in the graphical pane in the right-hand side, a DAG can be edited using the following functionalities.

Add a node: A new node (subtask) can be added to the task's DAG by dragging the blue circle in the tool bar (labeled with the text `Node`) and dropping it in the drawing pane.
Note, that in designing a DAG in **MIMOS**, the name of nodes in a DAG, as well as the name of the tasks in a project, must be unique.

Add an edge: An edge between two nodes can be added by left-clicking on the border line of a node and dragging towards another node.

Remove an element: A node/edge can be removed by right-clicking on it and selecting the *Delete* item.

Set node parameters: The parameters of a node can be modified by right-clicking on it and choosing the *Properties* item. Upon this, a dialogue is opened.

Locks The access of two nodes to a shared resource is modeled using a lock. This can be done by clicking on the *Locks* button, which opens a dialogue seen in Figure 3.2.

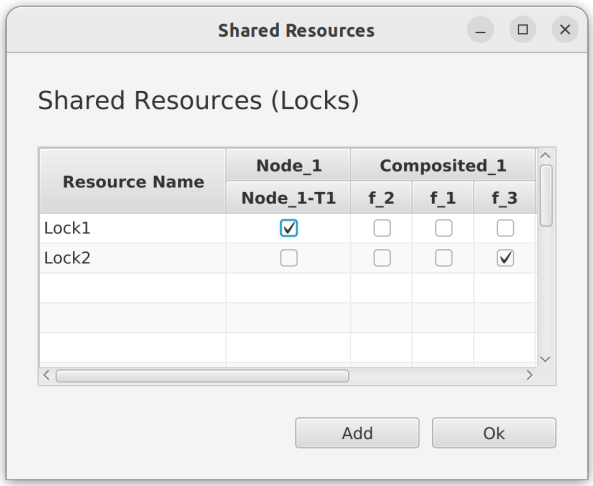


Figure 3.2: Specifying inter-node locks.

A lock is a shared resource (semaphor) to which two nodes (of the same task, or of two different task) access. From the timing analysis perspective, when one of the two nodes accessing a lock is running the other one cannot be executed even if it is running on a different processor core. This means that the node will experience an additional blocking/waiting time, influencing its response time. The tool allows only one lock per node of a task.

3.2 Real-Time Scheduling

Real-time scheduling for multiprocessors (in particular heterogeneous platforms) providing performance- and real-time guarantees is probably one of the most difficult challenges in embedded systems design. There are various scheduling algorithms proposed in the literature for single processors. These algorithms may be directly adopted to the multiprocessor setting. Unfortunately these algorithms perform often badly with low resource utilization; many of them

suffer also from anomalies due to the uncertainty introduced by complex hardware features and software components deployed.

MIMOS provides timing analysis of real-time DAG tasks running on multiprocessors. This is available by **double-clicking** on the *Scheduler* item in the tree view on the left-hand side of the main window. By this, the corresponding tab is opened as seen in Figure 3.3.

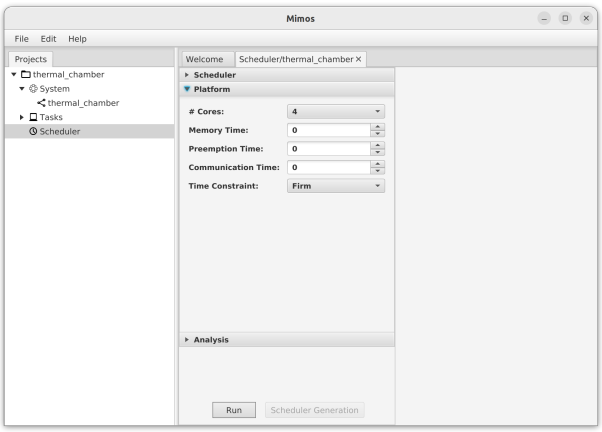


Figure 3.3: The Scheduler tab.

For timing analysis, we first need to specify the scheduling parameters and the platform, and then determine the analysis method, which can be analytical or simulation.

3.2.1 Scheduling Parameters

Scheduling parameters include scheduling policy, preemption policy, and the scheduling tick, detailed in the following.

Scheduling Policies

At run time, whenever there is a processor core available, eligible nodes will be selected according to their priorities. The priority of a node can be assigned statically or dynamically at node- or task-level according to static task parameters or parameters representing the current status of task executions.

Selecting and prioritizing the nodes for execution is accomplished by the *scheduler*. The scheduler considers *eligible* nodes to dispatch. Formally, at a time instant t, a node is *eligible* if by t: (1) all of its predecessors have been finished, and (2) the node itself is not finished.

There are a number of basic scheduling approaches that are used (directly or a modified version) in the tool.

EDF: Earliest-Deadline First assigns the highest priority to nodes with the earliest (absolute) deadline.

RM: Rate-Monotonic assigns the highest priority to nodes of the tasks with the shorter period.

LLF: Least-Laxity First (LLF) prioritizes eligible nodes according to their laxity. Laxity of a node in a DAG is the difference between the deadline and the longest path from that node to any sink node [4].

FIFO: In this approach, nodes are prioritized based on the arrival time (of the corresponding task instance). Earlier arrival means higher priority.

Federated: In the Federated scheduling approach [3], tasks are partitioned based on the minimum number of required cores (e.g., m). For tasks with $m > 1$, exactly m cores are exclusively dedicated. Other tasks (i.e., those with $m \leq 1$) can share cores to execute.

Partitioned: The nodes are accommodated to the cores so that each node is always run on the same core. Assigning the nodes to the cores can be done according to different criteria. The work in [1] presents some heuristics for that.

Preemption Strategies

To fulfill timing or QoS requirements, running tasks or nodes may be preempted by newly released instances of other nodes. Preemption may be (dis-)allowed at node- or task-level. Here only node-level preemption shall be considered.

Non-preemptive scheduling: A running node should run until it is completed. One may consider the case of task-level non-preemption: when a node is completed, only nodes from the same task can be executed.

Preemptive scheduling: A running node can be preempted at any time by any eligible node. One may consider the case of task-level preemption: a node can be preempted by only nodes from the same task.

Restricted preemptive scheduling: It is preemptive scheduling, but preemption is allowed to take place only at predefined time points. A special case is tick-based scheduling where the time line is divided into ticks or time slots. Preemption is allowed to take place only by the end of a time slot. In the tool, two variants of this policy are available:

- Ticked-Preemptive: This is similar to Non-Preemptive, but jobs can be preempted at ticks.
- NW-Ticked-Preemptive: This is a *Non-Work-Conserving* version of Ticked-Preemptive. The scheduler is invoked, and allowed to preempt the jobs, only at ticks, where ticks are equally-distant time points (specified by the user). If a job is finished before a tick, the core is idled until the next tick.

Platform Specification

Platform parameters needed for timing analysis can be specified in a panel. There is an option to use parameters from “Platform Tab”, or directly from this panel. In the former case, the number of cores, as well as “subtask to core” assignment (in the case of partitioned scheduling) will be obtained from the Platform tab.

- Number of cores can be set via a drop-down box, an integer number between 1 to 64.
- Memory Time: each task is assumed to have an initialization overhead to load the required data from memory.
- Preemption Time: in the case of preemptive scheduling, whenever a preempted entity wants to resume, a certain amount of time is spent on context-switch.
- Communication Time: this overhead is associated to nodes whose predecessor(s) has been executed on a different core.

Also, type of deadlines can be specified, which is either firm or soft. In a system with soft deadlines, a job may continue execution even after its deadline. All methods are implemented for the firm deadline, but soft deadline is enabled only for a subset of parameter settings.

Timing analysis can be done in two ways: analytical and simulation.

3.2.2 Timing Analysis

Timing analysis methods try to compute (usually an upper bound on) the worst-case response time (WCRT) of each task. In this way, a safe (but probably pessimistic) method for schedulability analysis is provided.

Figure 3.4 shows the result of timing analysis of a sample task set of four tasks. The second column lists the computed WCRT. As seen, the second task has a WCRT larger than its relative deadline. Therefore, the task set is not confirmed to be schedulable.

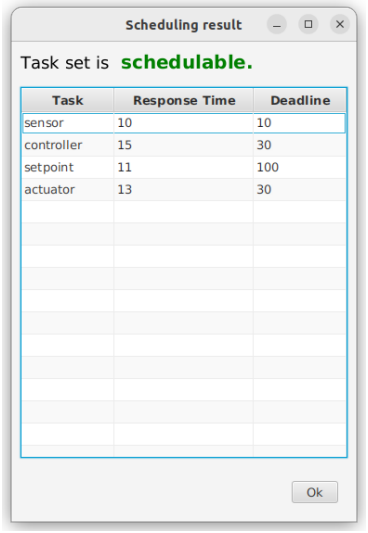


Figure 3.4: Timing analysis of a sample system with four tasks.

Deriving effective WCRT is not always affordable. For instance, for task sets with complicated restrictions (for example, DAG tasks with inter-task dependence) computing WCRT with an acceptable level of pessimism can be too difficult. In such cases, using simulation-based methods can be considered as an alternative. As it is practically non possible to simulate all situations, the result of simulation doe snot provide a guarantee. However, it can provide a statistical information about response time of the tasks.

Figure 3.5 shows the result of simulation for an example task set.

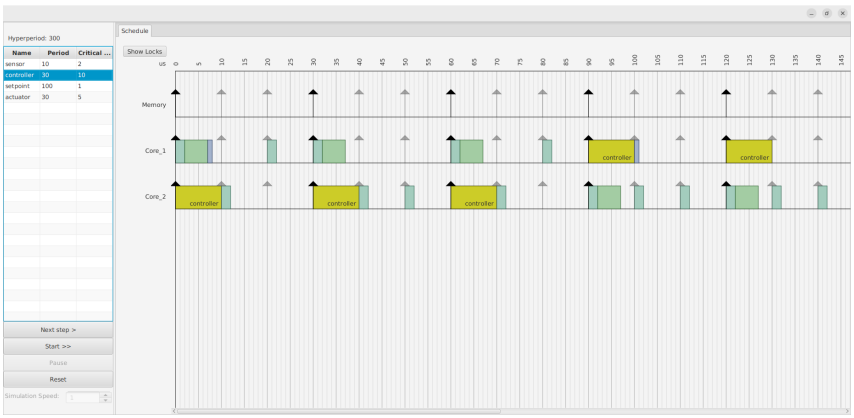


Figure 3.5: Gantt chart for the scheduling

Chapter 4

Code Generation

After designing and verifying the functional behavior of a MIMOS model and performing timing analysis, the final step is to generate executable code for deployment on target platforms. The MIMOS tool provides automatic code generation that preserves both the functional semantics and timing properties verified in earlier stages.

4.1 Code Generation Overview

The code generator transforms the validated MIMOS model into C++ code. The final executable is structured into three distinct layers, which separate user-defined logic from generated system code:

1. **Node Functionality**: The user-provided implementation of each task's computation logic (from Section 2.1.3). This is the same code used for simulation.
2. **Coordination**: *Generated* code that manages inter-task communication and synchronization through channels, ensuring correct read/write semantics.
3. **Scheduling**: *Generated* code that enforces the schedule produced by the Scheduler module (e.g., priorities, core affinities, and release times) to orchestrate task execution across processor cores.

The generated code targets multi-core platforms running Real-Time Linux with POSIX thread support.

4.2 Generate Code

To generate executable code:

1. Click the **CG** (Code Generation) button in the toolbar.
2. The tool will automatically generate the **Coordination code**. This generated layer is then integrated with your existing **Functional code** (i.e., the task functions you provided for simulation).

Note: The **Scheduling** policy code is generated separately through the Scheduler module (see Section 3.2). To generate the complete deployable system including scheduling, you must also use the **Scheduler Generation** button in the Scheduler tab after configuring scheduling parameters.

Bibliography

[1] Daniel Casini, Alessandro Biondi, Geoffrey Nelissen, and Giorgio Buttazzo. Partitioned fixed-priority scheduling of parallel tasks without preemptions. In *IEEE Real-Time Systems Symposium (RTSS)*, pages 421–433, 2018.

[2] Kim G Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell. *International journal on software tools for technology transfer*, 1(1):134–152, 1997.

[3] Jing Li, Jian Jia Chen, Kunal Agrawal, Chenyang Lu, Chris Gill, and Abusayeed Saifullah. Analysis of federated and global scheduling for parallel real-time tasks. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 85–96. IEEE, 2014.

[4] Roberto Medina, Etienne Borde, and Laurent Pautet. Scheduling multi-periodic mixed-criticality dags on multi-core architectures. In *2018 IEEE Real-Time Systems Symposium (RTSS)*, pages 254–264. IEEE, 2018.

[5] Wang Yi, Morteza Mohaqeqi, and Susanne Graf. MIMOS: A deterministic model for the design and update of real-time systems. In *International Conference on Coordination Languages and Models*, pages 17–34. Springer, 2022.