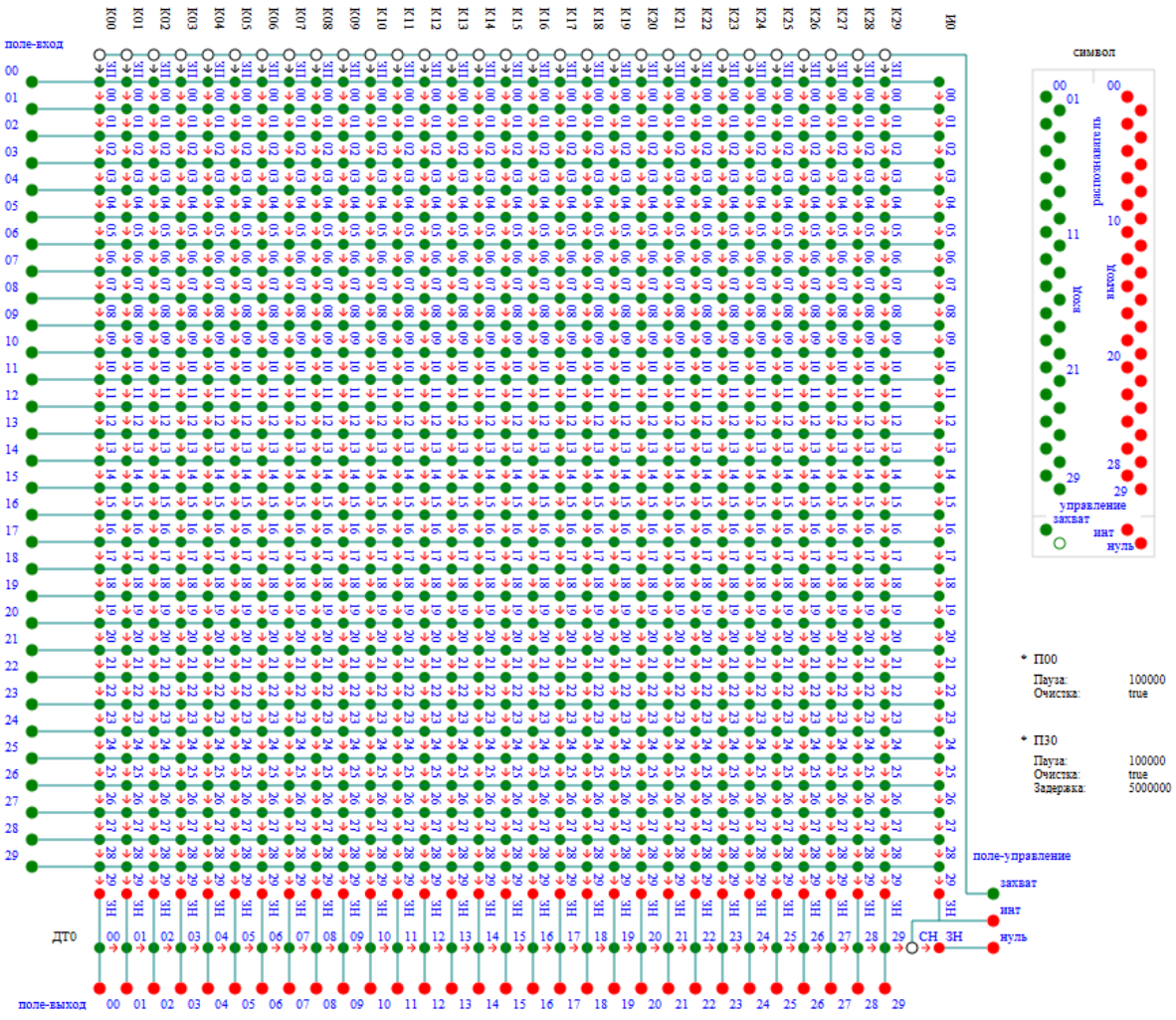
Thinker™

Functional Description

Version 1.0

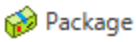


Introduction

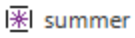
The Thinker™ is a math simulator instrument to perform visual examination of numerous computational processes running concurrently.

A math model is described as a hierarchy of math modules. Each function or group of functions inside a module can evaluate the result in its own computing thread. Modules exchange computed values in asynchronous manner. Each math value can be visualized at runtime as either a plain number or floating timeline graph.

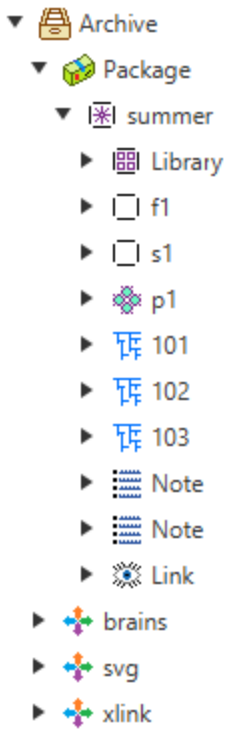
Math Model Definition



Math model (a synonym of project in the simulator) has external representation in XML format. Inside the simulator, every loaded XML file is represented as a separate package comprising one or more models.



Each model contains a set of parameterized computing blocks (a synonym of fragment in the simulator), data interconnections between fragments and processor definitions. A model can include a library of components that can be reused across the project. An example of a project, named “summer”, is shown on the picture to the right.

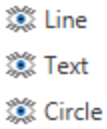


A network connection (a synonym of signal in the simulator) is simply a label, or a tag, that serves as a reference to a common sort of virtual media to propagate numeric values.

A fragment contains a set of so-called “pins”, or gates, for numeric values to flow in or out of the fragment. Each pin references one signal. This way many pins, having the same reference, are grouped together to form a separate network. A pin also declares a direction of data flow through it. In other words, a direction defines whether a particular pin is either a source or consumer of data floating through the pin.

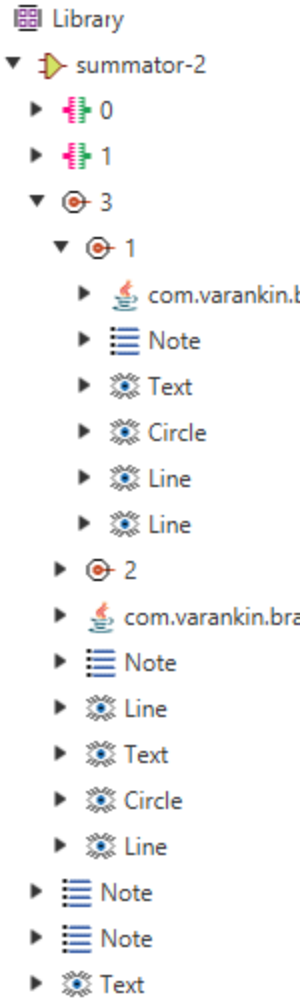
An arbitrary textual note can be added at any point of the package hierarchy,

to add comments to the data or for any other purpose. A note has a classification mark that helps to categorize messages.

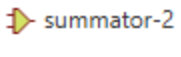


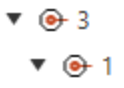
Also every node in the hierarchy can have supplementary information defined by a third-party. One sort of them, a standard of Scalable Vector Graphics (SVG for short), is supported directly by the Thinker. Many element types of SVG can be used to form an image of select nodes, including animations and HTTP hyperlinks. Combined together, they form a complete picture, for any node of the model hierarchy. An example for the “summer” model image is shown at the end of this paragraph.

A library can be treated as a local storage of component patterns (a synonym of module in the simulator). These patterns have no direct connections in between, they get connected when they are instantiated in a model via fragments. A pattern can be referenced by the fragment, provided with a processor to use and a set of so-called “external” pins, as was said above. An example of a computing module, named “summator-2” and stored in the library, is shown on the picture to the left.



There are few types of patterns. Most important is a computing module (on the right). It contains a special sort of hierarchy seen as a tree. Each node there evaluates a math function (“dot in circle” on the right). Arguments of these functions are taken either from enclosed nodes or external pins declared as data consumers. Top level node provides a computed value to a pin declared as a source of data. A particular formula of math function is declared in a form of Java class code (“Java cup logo” on the right), implementing a special interface. This way any complex math function can be computed.





A module having many input and output pins can be seen as a math operator over the matrix.

Another type of module is a so-called field. Simply put, it is a one-to-one interlink for connections, to perform elementary

transitional operations over the data.

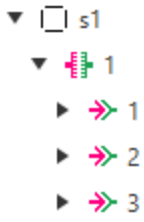
Modules can include other modules instantiated through their internal fragments, provided that embedded modules don't create a module reference loop. This way complex regular computing structures can be defined in a very compact form.

Each fragment, referencing the same module name, receives its own copy of a module, so all modules of the same type appear distinct to the model and so don't share the same computed data unless they are connected at fragment level.

As was said above, both fragments and modules have one or more sets of pins, named as joints for short (picture on the left).

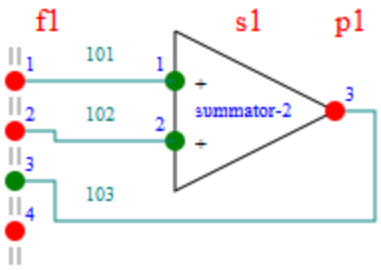
Every module, like a fragment, defines a set of pins (picture on the left) within a joint, called differently for the module as "internal" pins. These pins participate in connections declared within a module.

When a fragment instantiates a module, both "external" and "internal" joints get connected by means of enclosed pins. This feature allows data to flow in and out of the module. An example of internal pins is shown on the left. Another matched example for external pins - on the right.



A module can have more than one joint for the same set of connected signals. It is possible because a connection is rather a tag than a real "pipe" with data. Unconnected joins don't participate in runtime computation. All the border data flow performs through pin-to-pin declaration at fragment level. At a module level, every pin features a direction for data to flow across the border.

A main purpose to have more than one duplicate joint is a possibility to present a different visual representation of the module through the fragment. One method is to show the full contents of the module. Another variant is to display just a picture enclosed into a selected joint - the so-called "symbol" of the module (used on the picture to the right). There is a

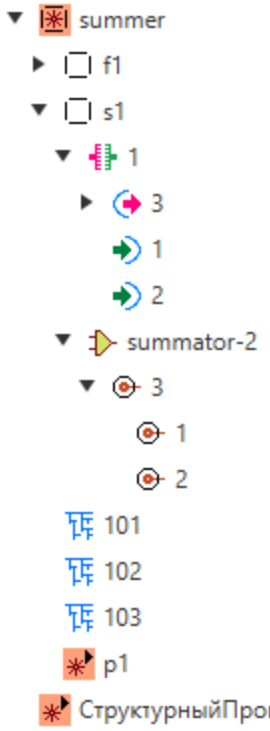


third variant too, having no graphics at all, but just a text string enlisting pins of the module. In a special case, no information will be shown at all. Last three variants effectively hide the contents of a module when a concise representation is needed. Selection of a participating joint is performed by name or through an “assembly” parameter.

Math model build-up and computation

Described definition of a math model uses a hierarchy approach to benefit low efforts to express what and how should be computed. In contrary to regular software, such a definition cannot be used directly during computing time, because there is no hardware to support these data structures directly. Very similar to integrated circuits production, this form of hierarchy description must be flattened.

The process of conversion is performed automatically and doesn’t require assistance from the researcher. All needed parameters and modes are already included in the math model. After the model is built, it appears in another browser (see picture on the right). It looks very similar to the original hierarchy and even keeps all labels, but at the background it is really flat. The runtime model is composed of a very limited number of element types. They are: processor, transmitter, receiver and math function. Other elements, despite the different picture in the tree, are simply virtual containers, used for the purpose of easy navigation through the model.



A transmitter (red, on the left) includes a set of receivers. Once a signal (data value) arrives at the transmitter, it is being replicated to every receiver. A receiver

(green, on the left), in turn, performs some elementary checks over the incoming data. Then, under the test result, it may schedule a computation request for the function.

This request arrives into the queue of the assigned processor. This processor (on the left) isn’t a real CPU core but a separate computing Java thread running concurrently with other such threads. Once ready for the next request, a processor

applies selected modes of computation to the queue. They can be divided into two groups: performance modes and accuracy modes.

A request at the front end of the queue triggers computation of the math function (on the left) at the assigned node. Once computed, this result flows either to the upper level computing node or to the transmitter, when the node is the top level node. This way a model computation process continues in the infinite dynamic sequence.

Some elements of the computing media can have data flow controls. They are: processors, computing blocks, models and the whole media. Their current state is indicated by the background of the icon in the browser. Controls allow to start, stop and pause computation for each such element individually. Of course, top level controls override settings of embedded controls.

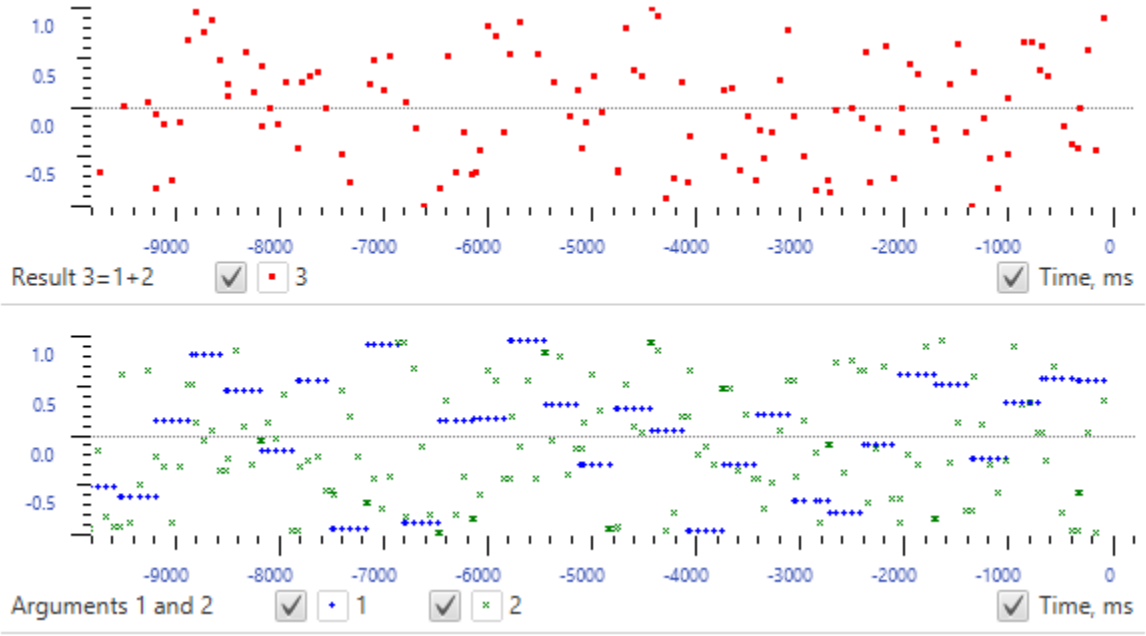
There are no limits on how many times a single model is built and run simultaneously. Each converted model appears as a separate tree in the browser. Unused models can be simply deleted to free resources to the application. While a model runs, a researcher has an opportunity to continue other work with the application.

Visual representation of computation results

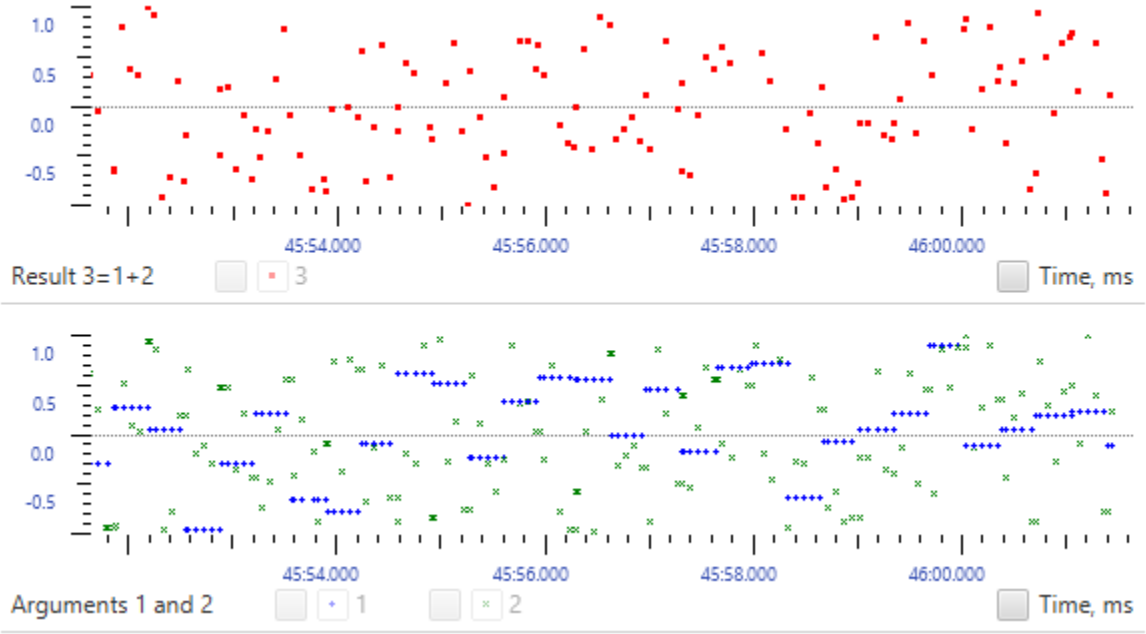
Generator of the model creates every node observable in a graph timeline tool. Any computing node or transmitter/receiver node can be applied to a graph by simple drag and drop operation. Displayed objects can be assigned presentation and conversion features. The graph itself also has separate presentation settings and value range settings for each axis. For the timeline axis, built-in control allows researchers to make a snapshot. An example of the “summer” model at runtime is shown below.

This is a dynamic flow graph. The simulator refreshes the graph after every specified interval. Values in millisecond range appear normal even for large screens having 4K resolution. Setting a small interval value creates a smooth motion picture. A big interval value may create flickerings but saves CPU/GPU resources and helps in total performance.

A moment of “now” is shown on the right side of the timeline axis and is labeled as “0”. All ticks to the left of it represent a state in the past, so they display labels as negative time offset.



A time flow can be stopped, to show a snapshot. For this case, timeline ticks display labels in current local time. A paused example of the same “summer” model is shown below.

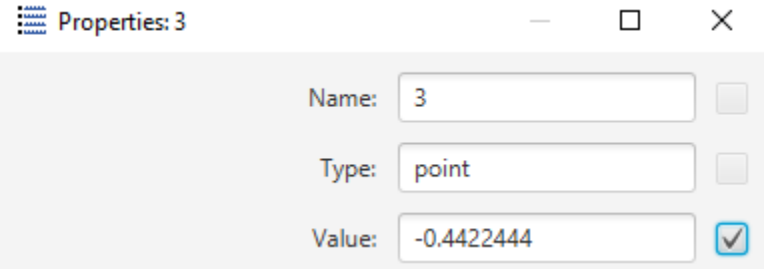


A pause feature is available for individual graphs too. Also, every observed value can be turned on and off, or simply deleted. For running values, it is possible to change

representation at every moment.

When a display of the value is paused, the model continues to compute it. To stop computing, a dedicated node, which has flow control features, should be used instead. When stopped, the graph will show no markings for this value on all graphs.

There are use cases when a graph is not needed. Researchers can have a look at node properties instead. This form of representation allows researchers to read an accurate digital value of the signal. When needed, a snapshot can be made as well. An example for a top level computing node is shown below.



Possible applications

This simulator is a convenient and easy tool to research, develop and debug complex processes described as math functions, where it appears very difficult to obtain an exact analytical formula for every part of a project. The Thinker allows researchers to define a compact math model for the entire project and simulate exact behavior instead, having a possibility to probe every model point.

As model data structure has a strong link to approaches used in integrated circuits design, it can be applied to simulation of complex digital and analog circuits having not trivial feedback loops. Final results can be mapped to a “silicon” design project with minimum efforts.

The Thinker has been successfully used in proprietary artificial general intelligence (AGI) research since around 2010.

Contacts for sale quotes and technical information inquiries

Scand Ltd.
Web: https://scand.com/
E-mail: info@scand.com
US: +1 (773) 831-4876
Poland: +48 (22) 219-98-19
Germany: +49 (212) 8807-9797
Belarus: +375 (44) 774-47-79