Workcraft: a static data flow structure editing, visualisation and analysis tool

Ivan Poliakov, Danil Sokolov, Andrey Mokhov, Alex Yakovlev

{ivan.poliakov, danil.sokolov, andrey.mokhov, alex.yakovlev}@ncl.ac.uk

Microelectronics System Design Group - University of Newcastle

http://www.ncl.ac.uk/

http://async.org.uk/

Abstract

Workcraft is a framework for the simulation, conversion and analysis of the SDFS models. The plug-in based architecture with embedded scripting language makes the framework an easily extensible and very flexible environment.

1. Introduction

Reliable high-level modelling constructs are crucial to the design of efficient asynchronous circuits. Concepts such as static data flow structures (SDFS) considerably facilitate the design process by separating the circuit structure and functionality from the lower-level implementation details.

Aside from providing a more abstract, higher level view, SDFS [3] allows efficient circuit analysis to be done by converting it to a Petri Net preserving behavioural equivalence. Once the equivalent Petri Net is obtained, existing theoretical and tool base can be applied to perform the model verification.

However, recent advances in SDFS design were largely theoretical. There are no practical software tools available which would allow working with different SDFS models in a consistent way and provide means for their analysis and comparison.

This work presents a tool which aims to provide a common, cross-platform environment to assist with these tasks. The tool offers a GUI-based framework for visual editing, real-time simulation, animation and extendable analysis features for different SDFS types. The models themselves, as well as the supporting tools, are implemented as plug-ins.

2. Software architecture

The tool is composed of a base framework and three principal plug-in types as shown in Figure 1.

The framework consists of three functionally separate modules: core plug-in management and scripting system, which is referred to as the server, the GUI-based editor, and the vector graphics visualisation system.

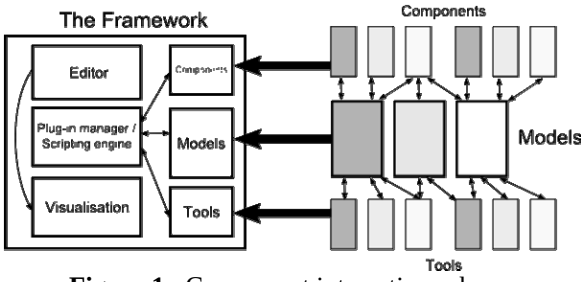

Figure 1 : Component interaction scheme

The *server* module is the fundamental part of the framework. It performs automatic plug-in management (loading external plug-in classes, and grouping them using the reported model identifiers). It also provides a scripting engine, based on Jython [1], which is used to provide further run-time customization flexibility.

The *editor* provides visual environment for efficient model design. Its functions include rendering a document view using the visualisation system; viewport scaling and panning; moving and connecting components; group move, delete and copy/paste operations. The editor also supports auxiliary features, such as "snap-to-grid" function (which restricts component coordinates to intersection points of grid lines), thus enabling the user to get desired alignment of the components with ease.

The vector graphics *visualisation system* is designed to provide two types of output: interactive visualisation, which is implemented using OpenGL hardware acceleration, and graphics export, which renders the document as an SVG [2] file. Both visualisation methods support a common set of drawing functions, which includes drawing of lines, poly-lines, Bezier curves, text and arbitrary shapes with customizable fill and outline styles. The two methods produce nearly identical result, so the developer only has to implement his or her component's drawing routine once for both interactive and external visualisation. The exported graphics provide measurements in real-world units, so that the effort of using them in printed material is minimal. The editor grid cells also have explicitly defined real-world size, thus document's final look on paper is always well-defined.

The concept of *model* is central to the software's plug-in architecture; the other plug-in types identify themselves as belonging to one or several models. The model plug-in manages simulation behaviour, handles component creation, deletion and connection operations, as well as performing document validation. Each model type defines a model identifier in the form of UUID (universally unique

identifier), which is used by other plug-in types to report supported models.

Component plug-in type defines individual nodes, such as registers and combinational logic. Components define the model-related data elements, visualisation features, user-editable properties, and serialization mechanism.

Tool plug-ins define operations on the model. Their functionality is not limited in any way, so this plug-in class includes everything from external file format exporters to interfaces with stand-alone model-checking tools.

3. User Interface

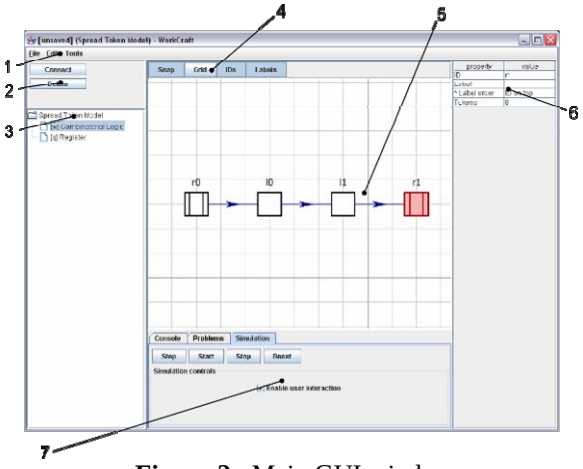

Figure 2 : Main GUI window

The framework's main window is shown on figure Figure 2. The main menu (1), besides standard file and editing operations, includes an automatically selected set of tools which support the current model. This selection occurs when a new document is created, or a document is open from a file. The editor commands (2) are duplicated both in the main menu and by hotkeys. The component list (3) presents set of all components supported by the current model, which are chosen in similar fashion to the tool set. A component can be added to the document either by dragging it from the component list, or by using a hotkey, which is optionally specified by the component. All components are further assigned numeric hotkeys from '1' to '9', corresponding to their order in component list. The editor options bar (4) contains toggle buttons to enable or disable certain auxiliary functions, such as display of labels, component IDs, editor grid and snap-to-grid editing mode. The document view pane (5) presents the document visualisation. It supports scaling (using the mouse wheel) and panning (holding right mouse button and dragging) to change the current viewport. It also supports moving of components, which is done by dragging them, and group selection done by holding the left mouse button and dragging the selection box over desired components. The selected components can then be moved together by dragging any of them, or deleted. The property editor (6) displays properties of currently selected component and allows editing them, such as, for example, changing the number of tokens in a Petri Net place. The utility area (7) holds three tabs: the console, which is used to display various information during normal execution of the program and also allows to execute script commands; the problems list which displays a list of errors which occurred during execution; and the simulation control panel which allows to start, stop and reset model simulation, as well as presenting additional, model-defined simulation controls.

4. Conclusion

The Workcraft tool presents a consistent framework for design, simulation and analysis of SDFS-based models. Its plug-in based architecture makes it easily extensible and very flexible environment, while inherent support for run-time scripting makes it even more powerful. Compact visualisation interface is very easy to use, and produces nearly identical results for both real-time visualisation and export to external graphics format without additional effort from the developer. Workcraft uses OpenGL hardware acceleration for real-time visualisation through JOGL API[4], which allows fluent, interactive animated simulations to be presented. Underlying Java technology provides robust cross-platform operation.

Currently developed Workcraft plug-ins support editing and simulation of Petri Nets, spread token, anti-token and counterflow SDFS models; conversion of SDFS models to behaviourally equivalent Petri Nets; Petri Net export in several formats for analysis using external tools.

5. References

[1] The Jython Project - http://www.jython.org/
[2] Scalable Vector Graphics - http://www.w3.org/Graphics/SVG/
[3] I. Poliakov, D.Sokolov and A.Yakovlev. “Spread token, antitoken and counterflow semantics for asynchronous datapath model”, *EECE Postgraduate Conference, Newcastle University*, January 2007.
[4] JOGL API project - https://jogl.dev.java.net/

Acknowledgement

This work was supported by EPSRC grant number EP/D053056/1.