μSystems Research Group

School of Electrical and Electronic Engineering

ArchOn: Architecture-open Resource-driven Cross-layer Modelling Framework

A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, A. Yakovlev

Technical Report Series

NCL-EEE-MICRO-TR-2014-184

January 2014

Contact: ashur.rafiev@ncl.ac.uk, alexei.iliasov@ncl.ac.uk, alexander.romanovsky@ncl.ac.uk, andrey.mokhov@ncl.ac.uk, fei.xia@ncl.ac.uk, alex.yakovlev@ncl.ac.uk

Supported by EPSRC grant EP/K034448/1

NCL-EEE-MICRO-TR-2014-184

μSystems Research Group
School of Electrical and Electronic Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

ArchOn: Architecture-open Resource-driven Cross-layer Modelling Framework

A. Rafiev, A. Iliasov, A. Romanovsky, A. Mokhov, F. Xia, A. Yakovlev

January 2014

Abstract

This paper describes the first steps towards the development of a modelling method for large complex computing systems focusing on many-core types and concentrating on the cross-layer aspects. The models resulting from this method will help system designers reason about, analyse, and ultimately design such systems across all conventional computing and communication layers, from application, operating system, down to the finest hardware details. The main points of concern are energy and power and the physical parameters related to them, such as supply voltages and temperature, among other things, and how these impact on and relate to system "performance" metrics, including speed, throughput, and crucially, reliability.

In this paper, we will first establish our outlook for the general modelling method, and then develop an initial system simulator based on this methodological outlook. The simulator will then be demonstrated with an example case study.

1 Introduction

The notion of "green" computing has been at the top of the list for a lot of research. Understanding the trade-off across power consumption, performance and reliability in cyber-physical systems has taken the crucial part in recent hardware designs [4, 5]. Software developers, on the other hand, in many cases have been ignoring the problem, but this can't go on forever [2]. In our research project, we are facing the challenge to develop a general adaptable power management system applicable to various platforms, which would take into consideration all levels: from hardware to the operating system, and eventually to a running application software.

This task, however, requires appropriate means to reason about this vaguely defined exploration space. The modelling and simulation methods are required to be open-ended with the capability of easily extending the simulated platform's functionality. We can't rely on some fixed architecture if we want to develop a universal solution. Such open-ended simulators exist; initially we considered extending gem5 [1] to facilitate our requirements. This simulator is a popular tool and is actively developed to support various platforms, including ARM. However, we soon faced a few difficulties, most notably, we often needed to simulate simplified or even hypothetical platforms. Extending gem5 to support each of them was not as easy as we wanted.

In addition, since the interest of our research includes reliability analysis, our models require formal specification and verification, which can't be done by the simulation alone. Another important requirement for the framework is the ability to operate at different layers of abstraction. Unlike conventional ideas of separating

concerns cutting between the layers, our goal is to observe the system as an intricate conglomeration of elements of different nature.

The ultimate goal of modelling is to provide convenient ways of studying complex systems. Graph-based models have been very popular in the human endeavour to reason about the world around us, because most people respond well to graphically represented concepts. This popularity in turn caused a rich set of mathematical techniques to develop within and around graph theory, which has become a very powerful methodology for rigorous reasoning [7, 8]. For example, Petri Nets [9] and similar formalisms support modelling for design and analysis with extensive tool support. However, complex systems with many abstraction levels tend to present difficulties to these existing methods, especially when cross-layer behaviours need to be investigated. Some researchers use intermediate frameworks, so that the process of modelling becomes clear and insightful whilst still keeping a formal background [10].

This paper describes the first steps towards the development of ArchOn: a cross-layer modelling method for complex hardware-software computing systems.

This paper is organised as follows: Section 2 outlines our approach to designing the model. Section 3 presents the first attempts to formalise it. Section 4 gives a use-case example and outlines the plans for future work. As this is on-going research, the results presented here are subject to future revisions and developments.

2 ArchOn envisioned

Due to the strict requirements and unusually vast design space, our research demands a special thinking in designing the model. We approached the problem in three stages, as described below.

Hardware vision approach The traditional method of creating extendible software is based on plug-in modules. In this case, the design of module interfaces is crucial and generally defines how difficult it is to create a program extension.

In order to understand the interaction between software modules, we have taken an unusual approach: “think hardware – make software”. We imagined that we are to make a flexible hardware architecture. What would the challenges be, and how should we overcome them? Just as FPGA allows customisation at the scale of logic gates, our hypothetical hardware must allow customisation at the scale of hardware modules, e.g. ALUs, register banks, memory units. Although in our research we do not have an actual task to deliver such a platform, this non-traditional thinking immediately paid back with a number of original design decisions, giving a better insight to the simulation software.

Figure 1 shows a communication-based hardware architecture that could potentially emulate most cyber-physical systems. This type of architecture is called transport-triggered architecture [6]. It hasn’t become popular in general purpose microprocessors, but it appeared attractive for our purposes. Assuming that the target system has an instruction set, its software can be recompiled into the connectivity fabric routing commands. The process of executing such software would have alternating phases of configuring the connectivity fabric and executing modules.

Of course, not everything can be envisaged in terms of hardware logic modules. At some point we had to introduce other types of elements like, for example, limited energy pool or time in order to explore the system capabilities. This is where our model diverged from the purely hardware view.

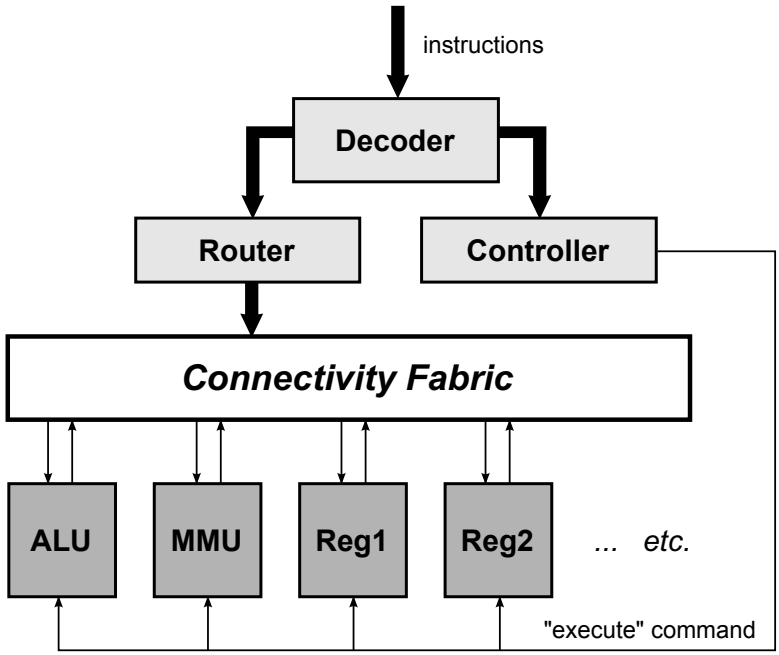

Figure 1: What a flexible architecture would look like in silicon? Making software design decisions while thinking in hardware terms.

Resource dependency approach The central subject of our method is the study of a computational platform comprising a number of diverse resources and the way resources may be handled in order to realise a computation. A resource is in this case an indivisible element required by the system in order to change its state, and it is defined by its function and availability in relation to this transition. With the word "resources" we make the point that we do not exclude computation, communication, or other facilities such as energy and time.

We propose to represent a system with a relation graph, consisting of a set of vertices and a set of edges. Each vertex represents a single resource and each edge represents a dependency between two resources. Modelling different types of resources may be achieved by labelling the graph, as illustrated in Figure 2(a).

With a reference to the hardware vision approach, we also prefer to view the system as a dynamic set of resource relations. Resources may become unavailable in certain points in time, and the model must be able to capture this behaviour. The understanding of resource availability properties helps to plan ahead and orchestrate resource consumption at a high and yet sustainable rate. A dynamic model can be represented using the state transition semantic, where the states are concrete resource allocations or configurations.

Cross-layer approach Organising systems, both practically and conceptually, as hierarchies is a popular way of thinking and engineering. The practical motivation for this is manageability. This is the "natural" way for humans to reason about, design, and organize most of our systems. Since in this project we emphasize the cross-layer aspects of our work, having a flat graph model as the foundation may seem counter-intuitive.

In fact, the flat labelled graph approach facilitates the cross-layer way of thinking, as Figure 2(b) demonstrates. A label can be viewed as a condition that includes or excludes an edge or a vertex, giving a graph *projection* onto that label. The complexity of the system can be dealt with using projections of the resource graphs. With resources as diverse as a software instruction or a single hardware gate, within the same single

graph executed in a transition, we could reason about different parts of the system at different abstraction layers. This helps a designer focus their attention on any particular details of a system they want, and build a system either top down or bottom up. This versatility is not available for methods which use explicit hierarchy within their frameworks.

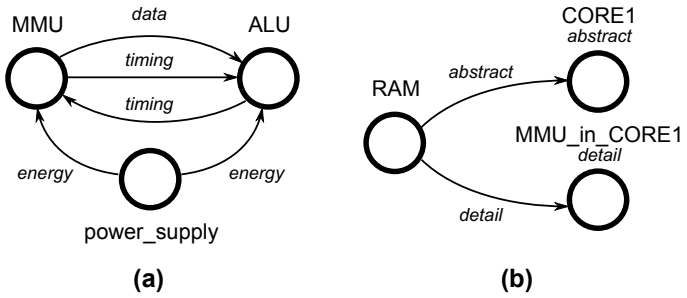

Figure 2: Examples of using labelled graphs to reason about diverse resource types and dependencies (a), and different levels of abstraction (b).

At the same time, this does not prevent designers to isolate concerns and concentrate on some layers only. For instance, all resources in one transition could be elements of the same layer, or a software engineer could arrange complex low-level software resources for detailed study with coarse-grain hardware resources provided by hardware colleagues (which are not the specific target of concern) in the same transition.

3 Model fundamentals

Putting together the outlines described in the previous section, we formally define *platform architecture* as a labelled directed graph

$$\mathscr{A} = (\mathscr{V}, \mathscr{E}, \mathscr{X}, \Phi),$$

where $\mathscr{V}$ is the set of all platform resources, and $\mathscr{E} \subseteq \mathscr{V} \times \mathscr{V}$ captures all possible (allowed) dependencies between them; $\mathscr{X}$ is a set of labels that can be assigned to vertices and edges in various ways by label assignment functions $\phi : \mathscr{V} \cup \mathscr{E} \to \mathscr{X}$, $\phi \in \Phi$. For each resource $v \in \mathscr{V}$ we also define: W_v – the set of possible resource states (can be infinite), and f_v – its node function, explained later.

An architecture is a loose, overarching agglomeration of concrete configurations. During the lifetime of an architecture instance we can observe the switching od resources, dependencies, and labelling. A configuration is understood to be a sub-graph $G = (V, E, X, \phi)$ of $\mathscr{A}$, where $V \subseteq \mathscr{V}$ is a set of allocated resources, $E \subseteq \mathscr{E}$ are active dependencies between them. Only one way of labelling $\phi \in \Phi$, $X \subseteq \mathscr{X}$ is allowed per configuration.

This is not sufficient to describe the whole state of the system though, as the resources may change their internal state over time as well. The state of the system is defined by a tuple (G, U), where G is a resource dependency graph, and U is a resource state vector giving for each $v \in G$ its state $u_v \in W_v$. We know, however, that the resource graph state space $\mathscr{G} = (G_0, G_1, \ldots, G_N)$ is finite for a finite architecture $\mathscr{A}$, while the set of different resource states $\mathscr{U} = (U_0, U_1, \ldots)$ may be unbounded. The state space of the system is a Cartesian product $\mathscr{G} \times \mathscr{U}$ giving an infinite state-transition machine, which is not convenient for a model. Therefore, we want to study two parts separately:

1) *Resource graph evolution* is a top level transition system that works on resource dependency graphs. This can be considered as an FSM where graphs represent states. Transitions between these graphs do not change the state of resources:

$$(G,U,\circ) \rightarrow \left(G',U,\bullet\right).$$

2) *Resource state evolution* is an inner transition system that operates on resource states:

$$(G,U,\bullet) \rightarrow \left(G,U',\circ\right).$$

Changing between $\circ$ and $\bullet$ means that both evolutions are alternating (as envisioned by hardware routing and execution phases in the previous section).

Transitions between resource states are defined using *node functions*. For each $v \in G$, the node function is defined as $f_v : (G,U) \rightarrow U'$. Typically, the node function operates on the sub-graph of G (e.g. the node's pre-set and post-set) and related projection of the resource vector U.

Node functions use labelling in order to distinguish between the arguments. For example, for a computation resource $sub \in G$ implementing subtraction $(x-y)$, it is essential to know, which resource dependency represents minuend x, and which represents subtrahend y. By respectively assigning labels "x" and "y", we can find projections $G|x$ and $G|y$, so the node function for sub would be $U'(sub\bullet) = U(\bullet sub|x) - U(\bullet sub|y)$, where $\bullet sub$ is the node's pre-set, and $sub\bullet$ is its post-set.

4 Use case example

One of the applications of our approach is simulating hardware running a software. In this case, the resources are concrete hardware units like ALU, MMU, etc. Resource states store unit data, and the resource dependencies represent data transfer between the units. The nature of the simulation is similar to what can be achieved using the state-of-the-art tools, e.g. gem5. In the initial stage of designing the framework, we need to prove the concept, so at this point we don't focus on the advanced features of the proposed model.

As an example, let's consider Euclid's algorithm for computing the greatest common divisor (GCD) of two numbers (a and b). The algorithm is very simple: *if* $(a > b)$*, then* $a := a - b$*; if* $(a < b)$*, then* $b := b - a$*; repeat this until* $(a = b)$*, which will be the result*. Its implementation in ArchOn is shown in Figure 3.

The resources are two registers `reg_a`, `reg_b`, and two ALUs: `cmp` and `sub`. A register is simply an identity function that copies its pre-set state, therefore in order to "maintain" the state it requires an explicit self-loop. This rule may be useful if we need to estimate the energy of the system: a self-dependent element remains in the graph as an active resource.

Although comparison and subtraction can be done in just one ALU, for the sake of example we consider `cmp` and `sub` to be different resources. Comparator `cmp` compares two inputs x, y and stores the result in its state, encoded *eq*, *lt*, or *gt* for "equal to", "less than", and "greater than" respectively. Subtraction `sub` is a memoryless combinational logic element, so it has no state in the model: the result is propagated to the output (post-set) node.

The method to supply this graph to the simulation software has been derived from our hardware vision, described in Section 2. We view the simulator modules as connected via the connectivity fabric, and the simulator input parser works as a router. Table 1 shows some commands for this "router", which provide step-by-step

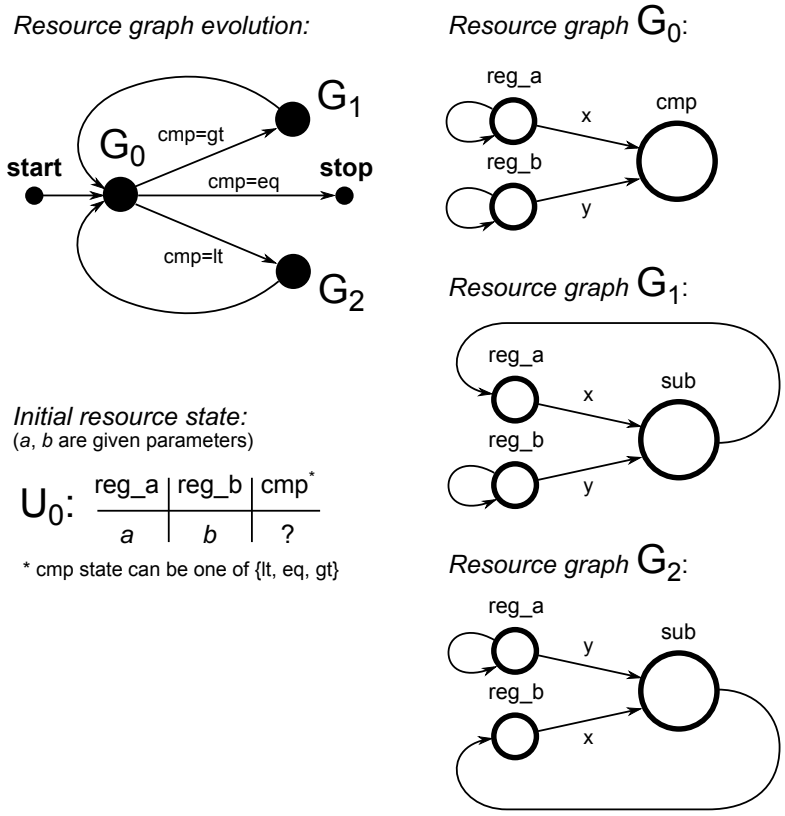

Figure 3: Simulating Euclid's algorithm for GCD(a,b). In this example, resources are hardware units with data dependencies between them.

graph configurations as well as explicit invocations of resource state transitions. Applying this method to sparsely connected graphs with many vertices gives more compact specifications than traditionally used adjacency matrices. We call it *graph assembly language*. The same GCD example written in graph assembly language is shown in Algorithm 1.

Table 1: Some commands of graph assembly language

command	description
U [a] $= value$	set resource a state to $value$
a $\rightarrow$ b	set a dependency between resources a and b
a $\overset{x}{\rightarrow}$ b	set a labelled dependency between resources
a $\nrightarrow$ b	unset a dependency
$G = \emptyset$	clear all dependencies
go!	"execute" graph: fire all resource state transitions
go to X	continue assembly from label X (jump)
if condition **go to** X	conditional jump

In addition to toy examples like the one above, models for real computers like 8051-series microcontrollers [11] and ARM processors are being built. Mapping processor instructions into graph assembly language manually has encountered no problems. These instruction-elemental graphs can be automatically combined

Algorithm 1 GCD(21,35) in graph assembly language

U [reg_a] = 21
U [reg_b] = 35
G0:
reg_a $\rightarrow$ reg_a
reg_b $\rightarrow$ reg_b
reg_a $\xrightarrow{x}$ cmp
reg_b $\xrightarrow{y}$ cmp
go!
reg_a $\nrightarrow$ cmp
reg_b $\nrightarrow$ cmp
if U [cmp] = "eq" **stop**
if U [cmp] = "gt" **go to** G1
if U [cmp] = "lt" **go to** G2
G1:
sub $\rightarrow$ reg_a
reg_b $\rightarrow$ reg_b
reg_a $\xrightarrow{x}$ sub
reg_b $\xrightarrow{y}$ sub
go!
reg_a $\nrightarrow$ sub
reg_b $\nrightarrow$ sub
go to G0
G2:
reg_a $\rightarrow$ reg_a
sub $\rightarrow$ reg_b
reg_a $\xrightarrow{y}$ sub
reg_b $\xrightarrow{x}$ sub
go!
reg_a $\nrightarrow$ sub
reg_b $\nrightarrow$ sub
go to G0

together into large resource evolutions representing actual platform software. Initial success has been achieved in populating resource states with energy consumption and unit delays allowing us to work on estimating physical parameters of the simulated systems. Results from such on-going work will become reportable in the near future.

Our immediate targets for the future include mapping ArchOn models into CPOGs [8] and Petri Nets for formal verification [3, 7].

5 Conclusion

The ArchOn method is developed to help designers of complex systems with multiple design objectives. Its unique resource-graph based approach represents the first known attempt at layer-crossing friendliness by being explicitly and implicitly layer- and level-agnostic. Fundamentally, a resource group represents a step of execution, and in any one such graph, resources could be diverse elements including components at all levels of detail and from all different abstraction layers. Resources could also include items outside of hardware and software, such as power, energy, reliability, time available, thermal budget, etc. This allows large design teams of experts from different disciplines to both concentrate on detailed problems explicitly and reason about inter-related

issues at more abstract levels.

Being a graph-based model, it presents users with a friendly interface for understanding. With its state-transition foundation it both matches the traditional philosophy comfortable for discrete event system designers and facilitates potential mappings onto the problem and solution spaces of existing modelling methods such as CPOG's and Petri nets, making it unnecessary to develop entirely new methods for analysis and verification. Its usability in system simulation is demonstrated though an example in this paper. The example also illustrates the method as a case study.

The development of the method is at an initial exploratory stage. Future topics of investigation include simulation plug-ins such as with gem5, and analysis plug-ins such as with CPOG's and Petri nets, and more explicit representations of concurrency, synchronisation, etc. and the best methods to incorporate important physical properties, such as energy, time and reliability, as resources.

References

[1] The gem5 simulator system. http://www.m5sim.org.

[2] The PRiME project. http://www.prime-project.org.

[3] Workcraft tool. http://workcraft.org.

[4] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, 2005.

[5] P. Bose et al. Power management of multi-core chips: Challenges and pitfalls. In *Proc. to Design, Automation and Test in Europe (DATE)*, pages 977–982, 2012.

[6] H. Corporaal. Design of transport triggered architectures. In *Proc. to Design Automation of High Performance VLSI Systems*, pages 130–135, 1994.

[7] V. Khomenko. *Model Checking Based on Prefixes of Petri Net Unfoldings*. PhD thesis, University of Newcastle upon Tyne, School of Computing Science, 2003.

[8] A. Mokhov and A. Yakovlev. Conditional partial order graphs: Model, synthesis, and application. *IEEE Transactions on Computers*, 59(11):1480–1493, 2010.

[9] T. Murata. Petri nets: Properties, analysis and applications. *Proc. of the IEEE*, 77(4):541–580, 1989.

[10] I. Poliakov. *Interpreted Graph Models*. PhD thesis, University of Newcastle upon Tyne, School of Electrical, Electronic and Computer Engineering, 2011.

[11] M. Rykunov, A. Mokhov, D. Sokolov, A. Yakovlev, and A. Koelmans. Design-for-adaptivity of microarchitectures. In *Proc. to Application-Specific Systems, Architectures and Processors (ASAP)*, pages 314–320, 2013.