Formal Model for System-Level Power Management Design

Mirela Simonović*†, Vojin Živojnović* and Lazar Saranovac†
*Aggios Inc., †School of Electrical Engineering, University of Belgrade, Serbia
Email: {mirela.simonovic, vojin.zivojnovic}@aggios.com, laza@el.etf.rs

***Abstract*—In this paper we present a new formal model, called p-FSM, for system-level power management design. The p-FSM is a modular, compositional, hierarchical, and unified model for hardware and software components. The model encapsulates power management control mechanisms, operating states and properties of a component that affect power, energy and thermal aspects of the system. Inter-component dependencies are modeled through a component-based interface. By connecting multiple p-FSMs we gradually compose the model of the whole system which ensures correct-by-construction system-level control sequencing. The model can also be used to formally verify the functional correctness of the power management design.**

I. Introduction

The aim of system-level power management (SLPM) is to control the power consumption of a system in order to meet energy, power, and thermal constraints. SLPM design has become a key challenge due to ever escalating power and thermal issues, low-power demands, and time-to-market pressure coupled with unparalleled system complexity. The system complexity is induced by a multitude and diversity of hardware and software components, as well as their intricate mutual dependencies. Dependencies and varying operational conditions determine the permissible system states and control sequences that must be followed to ensure the correct operation of a system. On the other hand, the number of system states and control sequences is enormous due to the multitude and concurrency of system components.

We believe that the use of formal methods is essential to master the system complexity and guarantee efficiency and correctness of an SLPM design. The pillar of a formal design methodology is the underlying mathematical model of computation which is our main focus in this paper. Although SLPM techniques have been extensively explored [1], [2], and some formal methods were applied in the domain [3], [4], the SLPM design flow is still remarkably disconnected, resulting in suboptimal efficiency and high cost of the end solutions. A lot of the existing work is based on power state machines (PSMs) [2] for power estimation, development of power management (PM) policies, etc. However, PSMs do not model control mechanisms and intricate inter-component dependencies. Rather, PSM related work in general relies on existing PM mechanisms, which are in practice still developed too late in the design process without the use of formal methods [5].

In this paper we propose a new formal model, called p-FSM, which should fill the gap in the design flow, facilitate coordination between different engineering teams and enable a wider use of formal methods in the SLPM design. Such goals require that the model provides a clear component-based interface, encapsulation, reuse, modularity, and the ability to model a system at various abstraction levels. The proposed p-FSM model meets all these requirements and additionally allows formal verification of the functional correctness of an SLPM design. Moreover, the model provides a formal, common ground for analysis, early design exploration, development of efficient policies, and PM optimization.

II. Modeling Paradigm

A. Modeling Dependencies and Sharing Resources

Controlling states of shared resources has apparently become a challenge in heterogeneous multiprocessor and multi-operating system (OS) environments. Consider for example a heterogeneous multiprocessor system-on chip (MPSoC) with application and real-time processing units, APU and RPU respectively, as in Xilinx Zynq Ultrascale+ MPSoC [6]. The APU fetches instructions from the DDR and the RPU occasionally stores some data in a dedicated memory segment. For a safe system operation the DDR must be in its "on" state when any of the processing units (PUs) initiates a memory access. In addition to "on", the DDR has "self-refresh" and "off" states. The goal is to model the dependency between PUs and the DDR in order to implement the control of the DDR state. The DDR shall be in its lowest power state whenever possible (effects of transition latencies on the performance are ignored for now).

None of the subsystems shall independently direct the state control of a shared resource. Traditional OS-based power management schemes [2] and Advanced Configuration and Power Interface [7] are designed for homogeneous, OS-centric systems and typically do not support such resource sharing. Recent developments for runtime PM in heterogeneous systems, such as System Control and Power Interface [8], provide interfaces for PUs to directly control the power state, clock frequency, etc. Such an approach requires that the dependencies are resolved between the PUs. This however burdens the software of PUs with mutual awareness, communication and hardware details, hindering software portability and reuse. Furthermore, these approaches do not rely on a formal model for design, implementation, and verification.

An approach for modeling dependencies would be to model the relationships between states of different components, sim-

978-3-9815370-8-6/17/$31.00

ilar to [9]. However, such an approach requires explicit state space and dependencies enumeration, what is impracticable for complex systems. Moreover, the information about component states becomes spread across the system model, breaking the encapsulation of component properties and making reuse hardly achievable. Taking the approach [9] for our example requires the implementation of an additional arbitration logic because PUs may depend on different DDR states. For instance, when the APU is suspended the DDR could be in "self-refresh", but if the RPU intends to access the memory, the "on" state should be configured instead.

In [10] the authors have proposed a requester-aware approach where software components request, rather than directly control, the power states of hardware components. However, requests contain only the information about *when* and *which* hardware components are requested, but not *how* would the component be used. Provided that the DDR is used in both "on" and "self-refresh" states, additional information about the usage is needed to reason about the optimal DDR configuration.

B. Requirements and Capabilities

We presented the challenge in modeling dependencies on a simple example with shared memory, but the actual dependencies in a system we model are typically more intricate and cumbersome. Dependencies and resource sharing exist across the whole system, between various combinations of diverse hardware/software components and at different levels of detail. Rather than specifying state dependencies or providing direct control mechanisms, we propose a modeling paradigm based on abstraction and requesting properties of components. Essentially, a component has some properties that other components require for their operation. Properties that a component needs are called *requirements*, while properties that a component has in some state are called *capabilities*. State control as well as the mapping of properties to states is encapsulated in a component model. The terminology and similar approach is found in Apple's I/O Kit [11]. Compared to the existing approach we generalize and formalize the principles in order to enable the use of formal methods in SLPM design.

For example, the DDR has the following capabilities: in the "on" state the DDR *preserves the context* and *can be accessed*; in "self-refresh" the DDR only *preserves the context*; in the "off" state it has no capabilities. The concept of capabilities is generic, e. g. capabilities can be *clock frequency*, *valid power supply*, *capability of an interrupt generation*, etc. Although this level of abstraction may not be possible for some components, the principles are the same for any abstraction level. A capability can represent a status register value or a fact that a subsystem is responsive. The benefits of such a modeling approach are numerous. PUs can be unaware of each other and the internal details of a component, such as available states, control mechanisms, and dependencies toward other components. Moreover, all kinds of different components have common properties, allowing the DDR to be replaced with a different type of memory, while the requirements of the PUs and their models remain the same.

C. Proactive Behavior

Existing models which describe a reactive system behavior assume that the system *can react* in its current state. An important aspect of system behavior that we model here is the fact that the system or its part cannot react in some states. Our goal is to model what needs to be done to bring the system in a state in which it can react to an event and perform an operation. In other words, we are looking to model properties and actions of system components which *enable* the reaction of a system.

For instance, if the APU is suspended and its full-power domain (FPD) is powered down, the APU cannot react to a wake-up interrupt triggered by a peripheral from the low-power domain (LPD) [6]. In order to react and handle the wake-up interrupt, the APU needs to resume operation. There is a whole sequence of state changes that precedes the APU resume, e. g. FPD needs to be powered up, DDR reconfigured, PLLs locked, APU reset released, etc. We need to model and ensure the correct resume sequence. The goal is to avoid the explicit system-level sequence specification and to preserve the encapsulation of component properties.

In order to react, the component's requirements need to be satisfied, e. g. nominal power supply, running clock, etc. We formalize the satisfiability of the component requirements as the *enable condition*. Although the enable condition carries information about the dependencies between components, its form doesn't directly give an answer to the question "what needs to be done for an enable condition to evaluate to true". Using our model a system itself answers the question and autonomously triggers a chained reaction among the system components. As a result, the enable condition will eventually be satisfied, enabling the component's reaction.

III. Formal Model

A. Starting Point

We consider modeling a behavior of each system component by an extended finite state machine (EFSM) and modeling a system behavior as a composition of communicating EFSMs (CEFSMs). Formally, an EFSM is defined as a quintuple [12]:

$$M = (I, O, S, T, \vec{x}) \tag{1}$$

where I, O, S, T, and $\vec{x}$ are finite sets of input symbols, output symbols, states, transitions, and variables denoted by a vector $\vec{x}$, respectively. The combination of state and variable values represents a *configuration*. Each transition $t \in T$ is augmented with a predicate P, called a *transition guard*, and an action A. Upon an input, the EFSM evaluates a predicate $P(\vec{x})$ on current variable values. If the predicate evaluates to false, it prevents the EFSM from taking the transition. Otherwise, the EFSM generates the output, performs the action which updates variable values $\vec{x} := A(\vec{x})$, and enters the next state. Actions model the control of variables.

B. Formalization of Requirements and Capabilities

We model properties such as temperature, voltage, or clock using variables of the EFSM. Such properties are typically common for multiple components, so multiple EFSMs need to access the same variables in order to evaluate predicates and actions. In terms of accessing a variable, we adhere to the *single writer, multiple readers* principle. By adopting this principle we eliminate the problem of concurrent writes by construction. Moreover, the machine that performs an action, called the *owner* of a variable, centralizes the control of the variable and encapsulates the assignment methods. The owner also encapsulates dependencies, i. e. how a value of the variable depends on variables of other machines, and how other machines depend on that variable. By knowing the dependencies, a machine can control the variables it owns with the goal to minimize the power consumption.

In order to formalize the principle, we extend the EFSM with finite sets of input and output variables. Accordingly, we define a p-FSM as a seven-tuple:

$$M = (I, O, S, T, \vec{x}, \vec{i}_{var}, \vec{o}_{var}) \tag{2}$$

where I, O, S, and T are defined as in (1). Vectors $\vec{x}$, $\vec{i}_{var}$, and $\vec{o}_{var}$ denote sets of local, input, and output variables, respectively. All variables have finite domains of values.

According to the single writer principle, a p-FSM writes to its local and output variables, while it only reads input variables. Input and output (I/O) variables of distinct p-FSMs are connected, so that a change of an output variable gets reflected on inputs of other p-FSMs. The value of an input variable is available for reading until it is overwritten by the owner. A p-FSM can be triggered by a change of an input variable value or by an input symbol. As the generalization of these two mechanisms we simply say that the p-FSM is triggered by an event.

From the perspective of a single p-FSM, a set of machines which communicate with it can be replaced by an equivalent p-FSM. We call such an equivalent p-FSM the *environment* (see Fig. 1). Next, we partition sets of input I_{var} and output O_{var} variables into *requirements* and *capabilities* subsets:

$$I_{var} = I_{reqs} \cup I_{caps} \tag{3}$$

$$O_{var} = O_{reqs} \cup O_{caps} \tag{4}$$

Subsets are collections of variables modeling: I_{reqs} properties that the environment requires from the p-FSM; I_{caps} properties of the environment that are provided to the p-FSM for its operation; O_{reqs} properties that the p-FSM requires from the environment; O_{caps} properties that the p-FSM provides to the environment for an operation.

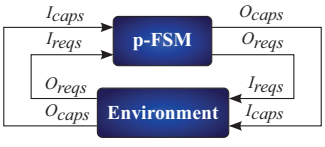

Fig. 1. Block Diagram of p-FSMs I/O connections

C. Communication

Variables from the input capabilities set represent properties of the environment that a p-FSM needs for the operation. Hence, prior to taking a transition the p-FSM evaluates the enable condition on the input capabilities. When the enable condition evaluates to false the input capabilities need to be changed to satisfy the condition and enable the transition. In order to conserve the encapsulation we follow the logic that no one knows better what a p-FSM needs than the p-FSM itself. Thereby, the p-FSM writes to its output requirements the values representing properties of the environment that would satisfy its enable condition. Note that the p-FSM specifies only *what* needs to be done, while the environment knows *how* to do that.

Upon being triggered by the p-FSM, the environment reacts in order to change the capabilities as requested. First, it reads the information about the request from its input requirements (see Fig. 1). Depending on the requested properties, the environment may change the state or just the configuration. Then, the environment writes to the output capabilities the information about its updated properties, triggering the p-FSM to finalize the transition. The equivalent sequence is followed when the p-FSM releases a property of the environment that it no longer needs. In either case, the reaction of the environment can actually consist of a chained reaction of multiple p-FSMs. Since the same principle holds between any two p-FSMs, the communication-based approach is scalable with the number of system components.

D. Computation

Upon an input event, the p-FSM first computes to which next states it may transit. Then, it evaluates the transition guards which are associated with the transitions to the next states. A transition guard dynamically constrains the behavior of the p-FSM by reducing the set of possible next states based on current values of variables. Such variables typically model operating conditions, permissions to use a resource, etc. If none of the transition guards evaluates to true, the p-FSM's reaction to an event is completed (see Fig. 2). Otherwise, the p-FSM follows the transition whose guard has evaluated to true and evaluates the enable condition to check whether all requirements for entering the next state are satisfied. The evaluation of the enable condition is performed in two phases.

In the first phase, the p-FSM determines requirements for entering the next state and assigns them to the output requirements. In the second phase, the p-FSM compares whether the current input capabilities satisfy the requirements. If the requirements are satisfied, the enable condition returns true and the p-FSM enters next state. Contrary to the transition guard, when the enable condition evaluates to false the p-FSM triggers the system to reconfigure its components in order to satisfy the condition. Then, the completion of the environment's reaction will be reflected as a change of a p-FSM's input capabilities. Upon an input change, the p-FSM repeats the evaluation of conditions and transits to the next state if all conditions are satisfied. When a machine enters the

next state, it outputs the capabilities which represent its new properties.

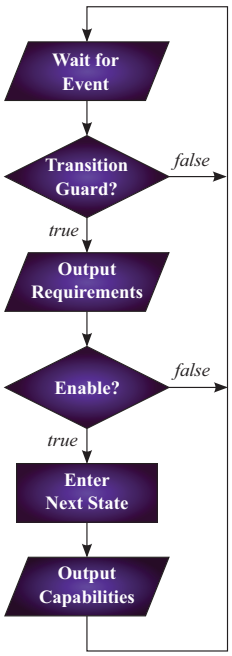

Fig. 2. Computation Flowchart of the p-FSM model

The system being modeled structurally decomposes into multiple components, each of which can be a system on its own. We form structural hierarchies of p-FSMs by clustering multiple p-FSMs into one and concealing their interconnections. The I/O interface of the resulting machine consists of inputs and outputs that remain unconnected within the cluster. Thereby, multiple teams can model the behavior of different components using p-FSMs and gradually compose the model of the whole system. The block diagram of the p-FSM model is shown in Fig. 3.

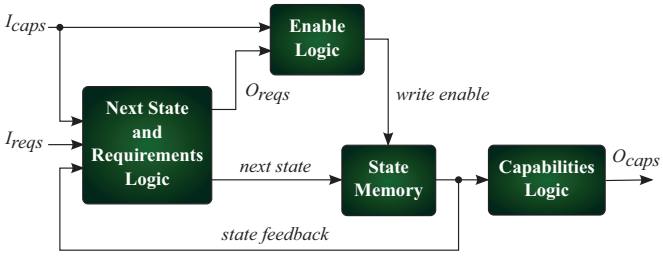

Fig. 3. Block Diagram of the p-FSM model

IV. Model Checking

We have validated our modeling approach on a case study for the Xilinx Zynq Ultrascale+ MPSoC [6]. The system model covers clocks, PLLs, interrupts, resets, power islands and domains, memories, processors, subsystems, abstracted software components, such as an OS and device drivers. Using the NuSMV model checker [13], we were able to formally verify that the model satisfies specifications such as: requirements of a component will eventually be satisfied, clock frequencies be configured as requested, a wake-up interrupt will eventually be handled, a reset to the processor is never released when the memory from which it will start fetching instructions is not accessible, etc. Additionally, we formally verified that the model has no deadlocks. We also used the NuSMV simulation mode for developing and debugging models. Counterexamples reported by the NuSMV helped understanding why a model did not satisfy some specifications. For instance, the suspend sequence may not result in powering down a domain due to a bug in a component model or an imprudent resource allocation.

V. Conclusions and Future Work

The new p-FSM model for formal SLPM design that we propose in this paper is generic and many other domains can also benefit from its adoption. We believe that the p-FSM formalism can improve coordination and methodology across all SLPM design phases. The proposed formal model supports component-based encapsulation and correct-by-construction design of the SLPM. Equally well it supports modeling of systems at various abstraction levels and enabling the formal verification of the functional correctness of the SLPM design. For future work we plan to add power consumption and latency information to the p-FSM model. Based on the p-FSM we plan to enable the use of other formal methods in SLPM design, such as hardware/software power management synthesis, automated testing and power characterization of an MPSoC, formal development of efficient power management policies, and others.

References

[1] L. Benini, A. Bogliolo, and G. De Micheli, "A survey of design techniques for system-level dynamic power management," IEEE Tran. VLSI Systems, vol. 8, no. 3, pp. 299-316, 2000.
[2] L. Benini and G. de Micheli, "System-level power optimization: techniques and tools," ACM TODAES, vol. 5, no. 2, pp. 115-192, 2000.
[3] A. Lungu, P. Bose, D. J. Sorin, S. German, and G. Janssen, "Multicore power management: ensuring robustness via early-stage formal verification," in Proc. of MEMOCODE, 2009, pp. 78-87.
[4] R. K. Gupta, S. Irani, and S. K. Shukla, "Formal methods for dynamic power management," in Proc. of ICCAD, 2003, pp. 874-882.
[5] R. Muralidhar et al, "Experiences with power management enabling on the Intel Medfield phone," in Proc. of Linux Symposium, 2012, pp. 35-46.
[6] Xilinx, "Zynq UltraScale+ MPSoC technical reference manual," v1.2, 2016.
[7] UEFI forum, "Advanced configuration and power interface specification," v6.0, 2015.
[8] ARM, "Compute subsystem SCP message interface protocols," v1.2, 2016.
[9] D. Li, P. H. Chou, and N. Bagherzadeh, "Mode selection and mode-dependency modeling for power-aware embedded systems," in Proc. ASP-DAC, 2002, pp. 697-704.
[10] Y. H. Lu, L. Benini, and G. De Micheli, "Requester-aware power reduction," in Proc. of the 13th International Symposium on System synthesis, 2000, pp. 18-23.
[11] Apple, "I/O kit fundamentals," 2006.
[12] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines - a survey," in IEEE Proc, 1996, vol. 84, no. 8, pp. 1090-1123.
[13] A. Cimatti et al, "Nusmv 2: An opensource tool for symbolic model checking," in Proc. ICCAV, 2002, pp. 359-364.