DiRAC: Dynamic Reconfiguration Animation and Control

Ralph A. Marczynski, Peter-M. Seidel
Department of Computer Science and Engineering
Southern Methodist University
Dallas, TX 75275
{ralphm,seidel}@engr.smu.edu

Abstract - This paper describes our developments with DiRAC (Dynamic Reconfiguration Animation and Control), a software tool for dynamic visualization and control of reconfigurable modular systems and their simulation. The tool is built on an extension of standard Verilog Interfaces. Alternative options of module functionality can be implemented either in (1.) Verilog, (2.) a logic hardware emulation on an FPGA PCI extension board or (3.) as an executable Program with standardized interfaces that is compiled from an arbitrary high-level language. For each module interface in the system we allow the reconfiguration between alternative module realizations in two modes: (i.) as (static) user-interactive reconfiguration or (ii.) as (dynamic) run-time reconfiguration. Our tool allows the simulation and animation of the dynamic system behavior in each reconfiguration mode and the user is allowed to choose the level of detail in the visualization interactively. By these means we support the specification, development and simulation of dynamically reconfigurable systems and assist in the improved understanding of their operation. The user-interactive reconfiguration mode additionally supports the incremental modular development of digital systems allowing for partial prototyping, integration and co-design of hardware and software functionality in hybrid descriptions.

I. INTRODUCTION AND SUMMARY
For the growing complexity of current and future digital systems, the support of the digital design process by adequate CAD tools becomes increasingly important during all design phases. The design and development of digital systems is particularly complex if the structure and behavior of the system can dynamically change like in the case of dynamically reconfigurable systems [2,6,14].

For specification of digital systems, hardware description languages (HDL) are widely used at various levels, Verilog [4] and VHDL being their most prominent representatives. For the development of reconfigurable systems, additional description languages have been proposed as for example JHDL[1]. Development of reconfigurable systems in special purpose languages does not allow for the reuse of module implementation descriptions from previous developments. It is therefore our objective to develop tools with standard interfaces for simplified interaction with existing tools. We even allow the development of systems in hybrid descriptions and combine interfaces and descriptions in Verilog HDL with FPGA hardware emulations and we allow interfacing with functionality in other high-level language modules.

To extend for the development of reconfigurable systems, restrictions in the syntax of conventional Verilog HDL are redefined and a few reserved variables are added for dynamic control of reconfigurable systems.

For simulation and debugging of systems described in hardware description languages, there are numerous tools available that focus on computing and comparing values of selected signals, representing their behavior as waveforms or textual output over time. Alternative to a specification using HDLs, digital systems can also be described by schematic entry incorporating a graphical representation of their structure, interfaces and module dependencies. Schematic representations simplify discussing, analyzing and understanding a digital system, in particular to non-expert designers, but they are typically limited to designs of low complexity. Conventional simulation tools considering schematic entry are very similar to those simulating HDL designs. The graphical structure available is conventionally not used for the simulations, which had been the reason for our developments of SDV$^2$ [5,10], a software tool that enriches the HDL description of a system with schematic representations and animations for simulation and debugging. The SDV$^2$ tool allows to schematically visualize the module structure of selected parts of the Verilog design. During simulation the signal propagations between module interfaces are animated as dynamic transitions in the schematics. This dynamic simulation visualization enhances conventional debugging of Verilog systems and simplifies the understanding of their dynamic behavior, which is useful both in educational and in industrial settings.

Our developments of DiRAC are an extension of the functionality of SDV$^2$ for reconfigurable systems. The
animation of system simulations extends from signal propagations to whole system configurations and their transitions. For each module interface in the system the reconfiguration between alternative module realizations is allowed in two modes: (i.) as (static) user-interactive reconfiguration or (ii.) as (dynamic) run-time reconfiguration. Our tool allows the simulation and animation of the dynamic system behavior in each reconfiguration mode and the user is allowed to choose the level of detail in the visualization interactively. By these means we support the specification, development and simulation of dynamically reconfigurable systems and assist in the improved understanding of their operation. The user-interactive reconfiguration mode additionally supports the incremental modular development of digital systems allowing for partial prototyping, integration and co-design of hardware and software functionality in hybrid descriptions.

II. DEVELOPMENT AS EXTENSION OF SDV²

The software architecture, the simulation approach and the internal data structure of SDV² had been developed with the extension to DiRAC in mind. In the following we give a high-level overview of the architecture of SDV² and identify the modifications necessary for handling dynamically reconfigurable systems.

SDV² exploits the modular structure of a system dynamically during simulation. Rather than combining module specifications to build data structures for global simulation, at each simulation time unit, all top-level and instantiated modules are simulated independently with dynamic variable initialization - leading to the resolution of variable values relative to the state of the complete system.

Simulation of each module is accomplished by bidirectional communication with an underlying Verilog simulator [6]. The user-provided module definition and a dynamically created variable initialization module serve as input. An additional dynamically created event-monitoring module, that defines the output format generated by the Verilog simulator, is used to extract system events.

The apparent appeal and stated motivation for SDV² is the schematic visualization of the Verilog module structure, further enriched by dynamic signal propagation at the occurrence of each system event. Such a graphical representation can assist in understanding the dynamic behavior of digital systems, their proper specification in Verilog HDL, and in Verilog code debugging. SDV²’s internal organization for system simulation by its multi-stage simulation approach delivers the correct simulation results, but is not optimal regarding its performance. The justification for this architecture is threefold: (1) The simulation can be based on interfacing with an existing Verilog simulator. (2) The limitations in the lines of code when using underlying public-domain Verilog simulators can be met even for arbitrarily large system simulations due to the modular simulation approach. (3) Module functionality does not have to be fixed, but can be changed dynamically for each simulation event.

Control over a module’s functionality is obtained from the independent treatment of each module. This offers little advantage when one considers implementations of static hardware. Considering reconfigurable designs, however, the benefits of such functionality isolation and control become evident. At the simulation of each module, reconfiguration control can be introduced that dynamically determines and changes the module’s functionality. The development of DiRAC is built on this property, in effect allowing for the simulation of dynamically reconfigurable systems.

Extensions of several aspects had to be considered in order to obtain the DiRAC architecture:

1) Compiler elaboration supporting new language definitions.
2) Efficient storage representing the system’s dynamic module structure.
3) Intuitive graphical representation of system configuration options.
4) Graphical interface extensions allowing system traversal.
5) Internal, adaptive system state representation.
6) Simulation accounting for dynamic reconfiguration.
7) Simulation of modules specified in multiple formats.
8) Simulation visualization representing reconfiguration.

The resulting application, DiRAC, contains the visualization features of SDV³, without the limitations of static hardware simulation. Our approaches to address these issues and the resulting architecture for DiRAC are further discussed in the following section.

III. STRUCTURE AND IMPLEMENTATION

A. Software Module Structure
The architecture of DiRAC is built on five main components that are depicted in figure 1.

B. Simulation Approach
The DiRAC tool partitions the simulation process into individual, dynamic simulations of each instantiated and top-level module. Rather than applying a system-level simulation approach, each module undergoes dynamic variable initialization, simulation, event extraction, and analysis of the resulting effects on the system-level state. This process is illustrated in figure 2.

By maintaining a modular system representation at all stages of execution, manual or automated control over module functionality leads to a simulation capable of dynamic reconfiguration. Such a modular separation allows for the flexibility of implementation description. Specific to Verilog based implementations; maximum code length requirements for the underlying simulator are relaxed. Such code size restrictions are a very limiting characteristic shared by many public domain Verilog simulators. Figure 3 shows the user interactive dialog to choose between alternative implementations for static reconfiguration.

C. Extraction of Module Dependencies
The original variable assignments within a module often directly or indirectly depend on outputs from instantiated modules. To allow for independent atomic module simulations that lead to a correct overall system behavior representation, any such dependency must be extracted to create an implicit simulation order allowing for the recognition of variable values when they are needed for further simulation.

The module dependencies are captured as a multi-dimensional tree structure with the top-level module at the root and instantiations serving as child nodes, ordered in terms of their dependency. Additionally, each child node represents any number of possible module implementations, further extending each level into a 3rd dimension. After each module’s implementation has been chosen either statically or dynamically, a bi-directional in-order traversal of the system structure takes place with simulation being performed at the visiting of each node: variable assignment is done during the forward direction, collection of instantiation results is done in the backward direction.

The computing of reconfiguration control signals is performed at two node-related graph traversal stages. During the initial node visit, shell simulation takes place to resolve reconfiguration events dependant on present state and input. On the reverse visit, a second shell simulation is performed additionally considering the most recent entered configuration. When a reconfiguration condition is met, the module instantaneously enters a “busy” state for a user specified period of time. While “busy”, further traversal is halted until reconfiguration has been achieved. At this point, traversal continues using the newly specified module implementation.

To consider the delay in reconfiguration, we define a variable within the module shell implementation. By making the reconfiguration delay parametric, DiRAC allows to adopt the simulation to a particular technology. Several tools have been developed to extract such delay information [12,17] for specific technologies.

Alternative methods for run-time reconfiguration management have been presented in [14,15]. Our tool provides the necessary abstraction of reconfigurable systems before such low-level investigations take place. By encapsulating the reconfiguration control within a Verilog module, DiRAC allows for a straightforward specifications of dynamic reconfiguration, a capability asked for in several previous efforts, such as [16].

D. Atomic Module Simulations
With the dependency extracted, during each individual module simulation, the input and internal variables can be initialized and the module can be simulated in one of three methods, dependant on module implementation: Verilog based, an executable program with standardized file I/O communication, or external FPGA hardware emulation.

In Verilog based implementations, variables can be initialized to their current values through the dynamic creation of a top-level module. Special considerations must be made in order to ensure proper event recognition by condition monitoring statements, such as always@(...) statements found within the event extraction methods. If during the simulation of time t-1 we extract a value change to occur at time t, value initialization for time t cannot simply occur in an initial block as this will not lead to the
recognition of change. A delay must be therefore introduced into our dynamic top-level module to force the occurrence of the event. The created initial block must be formatted to set all values from time $t-1$ as the initial values, and after the introduction of a single unit delay ($#1$) the initialization of time $t$ values must take place. Further extracted event times, in terms of the DiRAC simulation, are computed by:

$$t(E) = t(S) + t(V) - 1$$

with $t(S)$ being current DiRAC simulation time and $t(V)$ being the event time extracted from the underlying Verilog simulator.

To allow for the event extraction, a separate top-level module is created for the purpose of monitoring all variable changes and outputting the event to a log file at its occurrence. The set of dynamic modules along with the static module definition are sent as inputs to the Verilog simulator at each time unit simulation, and current and future event extraction takes place through the processing of the simulator's dynamically formatted output log file.

A similar method is employed when processing executable-based functionality specifications. In place of Verilog simulation, the provided module definition is simply initiated, with identical variable initialization and event extraction taking place through standardized initialization and log file communication with the executable.

REFERENCES