Configurable system simulation model build comprising packaging design data

H.-W. Anderson
H. Kriese
W. Roesner
K.-D. Schubert

A high-end eServer™ consists of multiple microprocessor chips packaged with additional chips on a multichip module. In conjunction with memory and various I/O cards, this module is mounted on a card called a ***processor book****, and a few of those cards on a board finally represent a major part of the system. Before the first hardware is built, simulations must be performed to verify that all of these components work together. But before we can build the simulation models, we need to find answers to many questions and to specify constraints, such as the scope of the simulation, the representation of the packaging data, the handling of cross-hierarchical connections such as cables, and the handling of passive components such as resistors and capacitors. This* ***system model build*** *should be as flexible as possible. System verification must be done for different system configurations (both single-processor and multiprocessor systems, one-processor-book systems, and multiprocessor-book systems) with or without I/O. Therefore, not only should a configurable model build downsize the model structure, but it should provide the capability to add logic. The requirement to include special logic, such as clock macros or checker logic, is driven by the use of emulation and acceleration technology and by other speed-related elements. This paper discusses these new concepts in eServer development: a configurable simulation model build, the automatic derivation of structural model data from packaging design, and the addition of specific logic without affecting the model structure generated by the previous step.*

Introduction

One of the first lessons a new verification engineer has to learn is that no verification can be performed by any kind of simulation without a proper representation of the design under test; this representation is called the *simulation model*. In most cases, a chip design is specified using a hardware design language (HDL). Various electronic design automation (EDA) companies offer tools that compile the HDL design source into a simulation model. In this paper, we call this the *classical approach*. Designs created with this approach are not described in one large flat file, but in multiple files built in a hierarchical tree structure.

On this basis, it may seem that all of our problems are solved and we are just a compile step away from developing a simulation model when it is needed. While this might be true for small ASIC chip designs, it is not true for the development process of a large server. That involves a host of new issues, most of which derive from the fact that a large, distributed team of people follow a complex process, with many individuals working on multiple chips and the connecting packaging infrastructure in parallel. In addition, the shortcomings of the standardized languages force additional innovation. This paper addresses the technical and process-related issues

0018-8646/04/$5.00 © 2004 IBM

that we had to solve to efficiently verify the system and describes the implementation details of our solutions.

System verification aspects

The design and verification of a high-end eServer* involves a significant number of design and verification engineers. With such a large group, it is very important to ensure the consistency of the data used in the daily work. Any copy with local modifications to solve a particular problem for a subset of people or a given task increases the probability that design problems will occur and remain undetected. If local copies are permitted, local modifications may not be communicated and implemented universally, and multiple versions of the design may coexist without anyone being the wiser.

Because of these practical limitations, we established a simulation-model-build ground rule that the design sources should never be modified; modifications required to support different verification activities were to be handled by simulation-only add-on files. This rule solves the problem in theory, but it also creates new problems. The first problem is that across the global verification team, there are different needs for the content of a system simulation model. For instance, team A needs to build a model that contains one processor chip, the complete memory subsystem, and one I/O hub chip, while the model needed by team B is identical, except that it should contain two processor chips and no I/O hub chip. In the classical approach, the difference between the two models would result in different versions of certain packaging levels. The only way to achieve this is to make a copy of the original HDL files, manually change the content to accommodate the different model configuration, and build a new model. This can certainly be done, but there are a few concerns.

When manual changes are being made in a copy of the design source, there is a likelihood that errors will be introduced. Furthermore, removing a component such as a chip from a given model can lead to unconnected signals that, in turn, may lead to compiler errors if they are not handled appropriately. Fixing these errors usually requires an iterative process that is not very effective. When this method is applied to really large systems with many packaging levels and many model configurations (and therefore a long list of modified files), building a new model can become a nightmare. The history and efforts in this area are discussed in [1] and [2]. Both describe the use of a hierarchical representation for a subsystem or system structure. In addition, the authors talk about the need for different system structures that can be effectively simulated, and they state that this is a challenge for system simulation. Our solution to this problem, simply by adding configuration files for the model build, is described in the next section.

The scope of system verification in the context of eServer goes beyond just the chips. All chips developed for the system are physically connected via the packaging infrastructure, which consists of cards, boards, multichip modules (MCMs), and cables. As the turnaround time for packaging design increases and, as a result of increased signal density, the possibility for workarounds decreases, it becomes more and more important to verify not just the chips, but the packaging design itself. Packaging is often designed using graphical tools, and designs contain analog components such as resistors, capacitors, and so on. Obviously the electrical behavior—including timing and noise—must be analyzed for all packaging levels, but the logic of these designs must be verified as well. Functional verification of the packaging is therefore one of the tasks performed during system verification.

As mentioned above, the model build for any logic simulation requires that the design be specified in HDL. To achieve a reasonable simulation performance and to make sure that even hardware acceleration or emulation can be used, the input to the model build is not allowed to contain analog components. Thus, the challenge is to find a process able to convert graphical design input into HDL while eliminating all analog components in a way that ensures that the logic behavior remains identical. As a side effect, solving this challenge not only helps verify the packaging design, but the packaging data provides information on the chip pin circuitry and its logic values. Therefore, some of the testbench initialization effort is obsolete. The section on generating system structure data from package design data discusses the solution and how packaging design input is translated into very-high-speed integrated circuit hardware description language (VHDL).

There is one additional packaging component that cannot be converted to HDL in the same way as the others: the cable. The reason for this is a restriction in the HDL. These design languages are defined in such a way that they can represent only designs that follow a tree structure. However, the nature of a cable is to break the tree structure and to connect two arbitrary leaves in the tree (**Figure 1**). To model a cable by the classical approach would require a kind of flattening process. One has to select the common node of the leaves that are connected via the cable and merge the connection described by the cable in all design files that lie between the node and the two leaves for the connection. Figure 1 highlights all of the design files that must be changed in that example.

To avoid having to change all of the files in the above example, we developed a proprietary extension to VHDL[1]

[1] VHDL [3] is the hardware design language of choice within the IBM Systems Group for pSeries* and zSeries* hardware development. To guarantee a smooth and seamless system simulation model-build process, all extensions and features described in the following sections are based on VHDL.

that is described in the section on Bugspray. This extension is able to describe all cases in which the design hierarchy cannot be mapped to a tree structure.

Finally, it is a very useful feature for the verification engineer to be able to add virtual logic into the model. This can be used for various actions, from implementing some types of low-level checking to adding certain driver capabilities. Also, it is a very useful feature for moving certain portions of the testbench into the model via virtual logic, particularly if the model is targeted for hardware acceleration or emulation where interaction between the testbench and the model must be minimized. With the classical approach, there are two issues in adding virtual logic. First, this can be done only by changing design sources. Second, in order to use this feature efficiently, it is often necessary to connect information from different hierarchical levels. Merging this simulation-only logic into the design source, as described for cables above, is just not practical. The obvious solution, of course, is to use the same concept we use to model cables. Thus, the solution we describe in the Bugspray section below provides verification engineers with new capabilities that can be applied for multiple purposes.

Configurable system-simulation model build

The model build for system simulation has to deal with a variety of factors defined by the given simulation environment and the coverage of the system verification. The simulation environment is based on the available software simulators and several hardware accelerators used as emulation engines. Such a heterogeneous simulation environment requires different simulation models. The model size is the important factor that determines whether a model fits into a hardware accelerator or whether the simulation speed is sufficient. **Figure 2** presents an overview of the various simulation engines, possible model sizes, and achievable simulation speed.

Because of the complexity of a high-end server, we must also answer the following questions: *Which parts of the whole eServer are included in the system model? How complete is the system model? Does the system model cover all verification needs?*

When verification starts, the first simulation model represents the smallest system configuration, for example, one CPU and a minimum of other chips—e.g., cache, cache controller, and clock chips. The first model is always a small one, because the verification environment is not yet stable, which is also true for the logic design. Restricting the error and debugging space is important for identifying problems in a short time.

Other simulation models follow, e.g., a two-way processor system, a system with a fully populated multichip module, a two-processor-book system, and finally, a multiprocessor-book system. All of these models can be configured with or without I/O cards or chips and with more or less memory.

When the model grows toward a multiprocessor-book system, the available simulation environment and verification tasks drastically influence the system simulation model build; i.e., not all of the multichip modules or cards can be fully populated. It must be possible to build various simulation models in a flexible, easy, and less time-consuming manner. If productivity is a measurement, it is impossible to design the logic descriptions (the system structures for these system

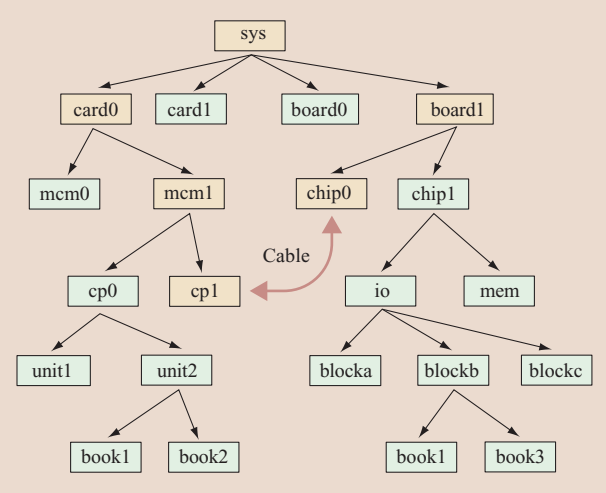

Figure 1

A cable is bridged across a tree-structured design hierarchy. The highlighted boxes indicate design files that would have to be changed in a classical approach to modeling.

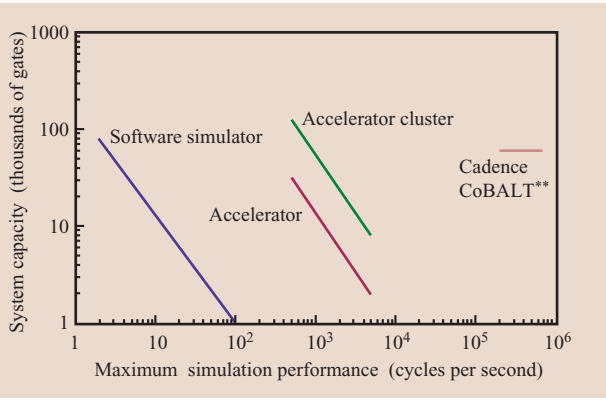

Figure 2

Comparison of the performance of simulation engines.

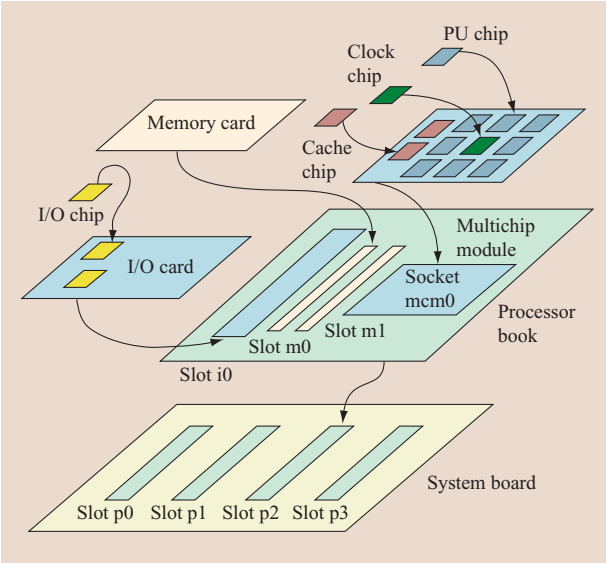

Figure 3

Example of a system structure.

configurations) manually. What source can be used to derive the data for the system structures?

As an example, **Figure 3** shows a system structure from the viewpoint of package design. As shown in the figure, this server system consists of chips, multichip modules, cards, and a system board. The modules, cards, and system board connect the chips and build the system structure, which represents the complete and fully populated system. Given this fully populated system structure, the task of building different simulation models is reduced to the task of *depopulating* a given structure.

Depopulating a system structure using VHDL "configuration"

The VHDL hardware design language [3] supports a feature called *configuration* that perfectly performs the task of depopulating a given design structure. Configuration is one of the structural modeling features of VHDL. Structural VHDL is used to describe the connections between components. In our case, we use VHDL to describe the connections between chips, multichip modules, cards, and the system board. The VHDL description of each of these components consists of an *entity* and an *architecture*. The entity describes the interface, i.e., the inputs and outputs of that component. The architecture instantiates other components and describes their connections. Configuration binds these component instantiations to an *entity/architecture* pair or to another configuration.

VHDL configuration is applied to modules, cards, and boards (Figure 3). Each of the chips is used as a kind of black box; that is, the chips are pre-compiled to an intermediate data format that can be used "as is" during system model build. **Figure 4(a)** is an example of a VHDL description for a multichip module. Port maps and other connecting signals have been omitted in order to provide a short and clear example.

The VHDL configuration file for this `multi_chip_module` may look like **Figure 4(b)**. With the configuration file shown in Figure 4(b), the architecture `arch` of the entity `mcm` is configured with a clock chip, whose entity is called `clock` and its architecture called `arch`, and this entity/architecture pair is bound to instance `CLK` of the component `clock_chip`. The same is valid for the instance `PU0: pu_chip`.

The instance `PU1`, component `pu_chip`, is bound to a different entity/architecture pair, and the architecture is called `dummy`. This architecture may not represent the whole functionality of a "real" processor chip, but may instead represent a version with some stripped-down functionality to accommodate model size limitations.

The instance `PU7`, also component `pu_chip`, shows one more possibility of a configuration. Instead of specifying an entity/architecture pair, the statement `open` is used. That means that nothing is plugged into the socket for `PU7`, and the inputs and outputs for this instance remain open.

For the cache chips, a different notation is used. Instead of separately specifying each instance of the cache chip on the `multi_chip_module`, the statement `all` is used to express that all instances on the `multi_chip_module` are populated with cache chips.

The next hierarchy that has to be configured is the processor book, whose configuration file may look like **Figure 5**. The configuration `proc_book_config` shows another detail of VHDL configuration: As already mentioned, configuration binds component instantiations to an entity/architecture pair or to a configuration. The binding to a configuration, i.e., `mcm_config`, is used for the instance `mcm0: mcm` on the processor book. This makes it possible to build up a hierarchy of configuration files. Accordingly, the memory card, the I/O card, and the board can be described with configuration files.

In summary, VHDL configuration provides the flexibility to

- Bind all instances of a component to one entity/architecture pair, i.e., the `all` statement.
- Bind each instance of a component to a different entity/architecture pair.
- Plug in nothing, i.e., bind an instance of a component to `open`.
- Bind an instance of a component to another configuration.

```
Multi_chip_module:

ENTITY mcm IS
     PORT(
     !!--- Interface to other components:
     !!--- inputs...
     !!--- outputs...
     );

ARCHITECTURE arch OF mcm IS
     COMPONENT clock_chip   PROT (...)  END COMPONENT;
     COMPONENT pu_chip PORT (...)  END COMPONENT;
     COMPONENT cache_chip  PORT (...)  END COMPONENT;

BEGIN
     !!--- instantiation of other components and their connections
     CLK: clock_chip
          Port map(
               !!-- connections of clock chip(instance CLK)  with
               !!-- multi_chip_module signals
          );
     PU0: pu_chip
          Port map(
               !!-- connections of clock chip(instance PU0)  with
               !!-- multi_chip_module signals
          );
     PU1: pu_chip
          Port map(
               !!-- connections of pu chip(instance PU1)  with
               !!-- multi_chip_module signals
          );
     !--!instantiations PU2 ... PU6 will follow here
     ...
     PU7: pu_chip
          Port map(
               !!-- connections of pu chip(instance PU7)  with
               !!-- multi_chip_module signals
          );
    CA0:  cache_chip
          Port map(
               !!-- connections of cache chip(instance CA0)  with
               !!-- multi_chip_module signals
          );
    CA1:  cache_chip
          Port map(
               !!-- connections of cache chip(instance CA1)  with
               !!-- multi_chip_module signals
          );
END arch;
```

(a)

```
CONFIGURATION  mcm_config OF mcm IS

     FOR arch

          FOR CLK: clock_chip  USE entity  clock (arch);   end for;
          FOR PU0: pu_chip USE entity  cpu (arch);    end for;
          FOR PU1: pu_chip  USE entity  cpu (dummy);  end for;
          ...
          ...
          FOR PU7: pu_chip USE open;                  end for;
          FOR all: cache_chip   USE entity  cache (arch);  end for;

     END FOR;

END mcm_config;
```

(b)

Figure 4

(a) Sample VHDL description for a multichip module. (b) Sample configuration file for the multichip module described in (a).

```
CONFIGURATION  proc_book_config OF proc_book IS

         FOR arch

              FOR mcm0:  mcm     USE configuration mcm_config;     end for;
              FOR m0:    memory  USE entity  mem (arch);           end for;
              FOR m1:    memory  USE open;                         end for;
              FOR i0:    io_card USE entity io_card (arch);        end for;

         END FOR;

END proc_book_config;
```

Figure 5

Simple configuration file for a card.

Each structural component of the system model consists of two files: the VHDL structure for that component and the VHDL configuration file for that component. Figures 4 and 5 show that VHDL configuration allows a given system structure to be populated (or depopulated). The configuration files accompany the system structure files. These system structure files, which contain all of the packaging design work, remain untouched. VHDL configuration provides the flexibility of adopting simulation models to the various verification aspects without affecting the sensitive system structure data from packaging design.

In addition, the VHDL configuration files are small and easy to read and give a quick overview of a structural component or the whole system. All of the structural data for a system are available in packaging design. As a consequence, all VHDL *structures* for the system components and the accordingly fully populated configuration files can be derived from packaging data. **Figure 6** shows a general model-build process with structural data from packaging design.

With a fully populated system configuration, the task of manipulating a system structure is reduced to the task of changing some statements in a given configuration file. This manipulation can be done manually or by a program.

Generating system structure data from package design data

For the logical verification of a system, the chips and their signal interconnects across the packaging components are modeled for logic simulation. As mentioned in the previous section, it is mandatory that the overall chip signal interconnects be described in hierarchical VHDL that reflects the complete system package design (chip–module–card–board–cables).

General process description

Chip designs for eServers are commonly described in VHDL. Package design methods and tools, however, are locally (more or less) different. Package designers are using Cadence** graphical design tools, but there are several varieties of these: the Concept** tool, which is used mainly for cards and boards; the Composer** tool, which is used for chip design and also for MCM design; and the Allegro** tool, with which all physical package design is created. In several IBM development laboratories, local processes had previously been developed to provide (human-readable) very-large-scale integrated models (VIMs)[2] out of the Cadence data repositories. These VIMs could be used outside Cadence for processes such as system timing or simulating special functions, for example, shift chain handling.

Traditional chip-level design verification did not include package design (with the possible exception of the MCM structure), so there was no significant requirement to provide package design data in a form that would allow an easy inclusion into the logical system verification process. Therefore, when defining the process to create VHDL from package design data, it appeared to be useful to build upon the VIMs that could be generated by the local tools. This would also decouple the package design data extraction from the translation to VHDL and model-build processes.

The first process that was implemented was the translation of VIMs to VHDL. Once it had been proven to work properly, the local tools to create VIMs were replaced by new programs. The package-design-to-VHDL process now consists completely of new programs.

[2] The VIM API is described in principle by the Chip Hierarchical Design System Technical Data Standard (CHDStd) API [4]. This work was done as a collaboration between IBM and si2 (see *http://www.si2.org/*).

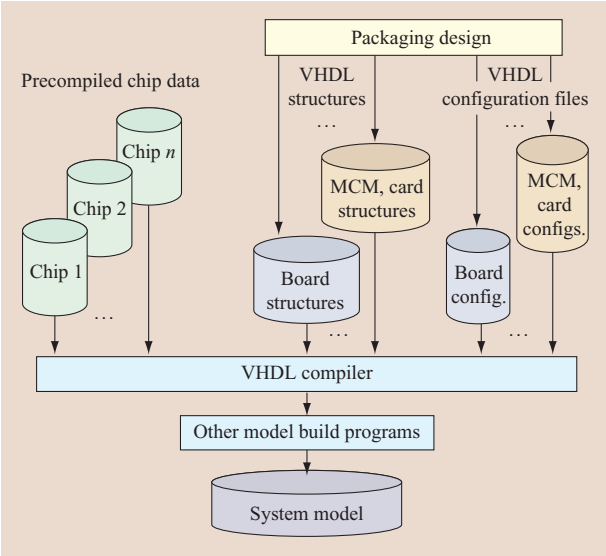

Figure 6

Sample model-build process.

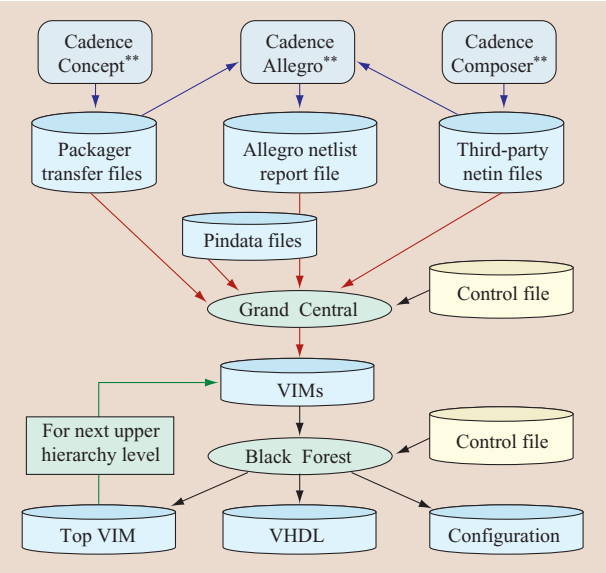

Figure 7

The package-design-to-VHDL process with its two steps: the Grand Central and Black Forest processes.

Process details

The package-design-to-VHDL process consists of two steps (**Figure 7**):

1. The export task (extracting the packaging design data to VIMs: *Grand Central*).
2. The processing task (manipulation and translation to VHDL: *Black Forest*).

The first step of the new process (Grand Central) extracts the design data from the files that are input directly to Allegro: either the packager transfer files for designs in Concept or the third-party netin files for designs provided in other ways, or also similar reports (netlist format) out of Allegro. Designs from Composer would probably use third-party netin files, but this has to be defined in detail. For some MCMs, netlist reports from Allegro are provided that no longer contain the logical pin names of the chips, so an information file is required for every chip that must correlate logical pin names to physical pins (*pindata*, typically also used for MCM design tasks) and also for the MCM I/O pins if the pin signal names are different from the connected (MCM internal) net signal names.

All of these translation processes can use additional information by means of a control file. An example would be to add information on the type of logical function of a component (e.g., resistor, capacitor, logic integrated circuit).

In the second step, the data (stored as VIM files) resulting from Grand Central or from Black Forest on designs lower in the hierarchy has to be processed (manipulated) for the translation to VHDL system simulation structures. The manipulation actions are controlled by means of a control file that uses a keyword-values scheme. There are some basic manipulations necessary to fulfill the requirements that the simulation model build puts on the simulation structure VHDL:

1. Preparation of the contents (the components or parts on an MCM, card, or board and their interconnects) for simulation. This includes removal of all parts that are generally excluded from simulation, such as mechanical parts, thermal sensors, or power-supply parts, as well as unconnected wires (signals without any connection). It also includes the handling of special function wires, such as connections to power planes (voltage or ground). All other information not needed for simulation (e.g., geometrical data) is also removed.
2. Establishing the correct hierarchy for the whole structure (i.e., system, board, or card, etc.). All physical connectors (card connectors, cable connectors) must be resolved into the corresponding card I/O pins in the card VIM and into corresponding part references in the board (or card) VIM containing the slot connector. These constructs can then be smoothly translated into a hierarchical VHDL. For connections that violate the

hierarchy, a special construct is used (see the section on Bugspray).

3. Resolving the logical function of capacitors and resistors: Only capacitors that are connected to at least one power-supply signal and resistors between power-supply signals are deleted. Capacitors and resistors between logical signals ("serial" case) are treated as a piece of wire. Resistors connected to a power-supply signal on one side result in a resistive drive to high or low in VHDL.
4. Signal direction (`IN, OUT, INOUT`) is essential for simulation but has no real meaning in package design (except for graphical pin places, left or right), so that the correct direction for each signal has to be defined from the bottom leaves in the hierarchy (the chips), and then propagated up to the MCM, card, and board, in such a way that the rules from the VHDL compiler and the simulation model-build tools are not violated.
5. Translation into VHDL. The translation is done one design (VIM) at a time. This means that there is a VHDL for an MCM, or a card, or a board, etc. The VHDL files (or the compiled form, the *dadb-protos*) for the chips/components/parts are assumed to be available. This translation includes the generation of a configuration file—a set of fully populated configuration statements for the item being translated (e.g., for all chips on an MCM). The compiler resolves all references across the hierarchy. This implies that the process has to be applied bottom-up. To be able to process parts on the next higher level, a special VIM, called *DEF VIM*, which defines the part I/O, is generated for every design that is processed (e.g., an MCM DEF VIM when processing an MCM with its contained chips to be used when processing the card where the MCM is to be mounted).

Remarks

Both processes, Black Forest and Grand Central, have been implemented as C-code that runs as a client program of the simulation database, *dadb*, and makes use of its application program interface (API). This was the natural consequence of the plan to see this conversion as a task performed by the simulation people. They use an IBM-proprietary cycle simulator for their verification tasks, and the model build for this simulator is based upon the mentioned simulation database, dadb. The principles behind that simulator and its simulation database can be found (as a brief description) in [5].

Much checking is performed prior to translation, which aids in finding problems in the packaging design, such as nets with no source/no sink or part pins that are referred to on a card but do not exist in the part definition.

Package designers do not follow conventions for identifiers as rigidly as chip designers usually do. Therefore, it is necessary to provide some information in the control file, which can be a lot of work: identify types of parts (e.g., resistor, capacitor), identify power-supply signals (e.g., V_{DD}, GND). Another task is to identify parts that are not necessary for simulation, although they have a logical behavior.

Until recently, package design was logically verified mainly by eye inspection of graphics or lists such as the netlist reports mentioned above. With this new process, it is possible to automatically incorporate the package design into the simulation model directly from the design data, which is either input to physical design or generated in physical design. This—and the fact that no manual data manipulation takes place (with the exception of the generation of the control files for the process)—ensures not only that the logical functions on the chips of an eServer system are verified, but also that the exact interconnections of all of the signals from and to the logic chips across all modules, cards, and boards (including cables) are verified.

Manipulate the system structure: Bugspray

The first part of this paper describes a feature of package design data that is not supported by VHDL: the need to model cable connections that cross hierarchical boundaries (Figure 1). VHDL was designed to specify structural hardware models using a strict hierarchy. Components can communicate with other components only via signals that are bound to the ports of the components.

As Figure 1 demonstrates, a cable (signal) connection between ports of *chip0* and *cp1* would require additional ports on components *board1, card0,* and *mcm1*. Finally, on the *sys* level, a signal would connect the newly created ports of the topmost components *board1* and *card0*.

Thus, the strict hierarchy constraints of VHDL force the creation of artificial ports on components that do not have corresponding physical pins. This requirement has two disadvantages:

1. Components now possibly have ports that represent physical pins and virtual ports that are necessary only to facilitate the cross-hierarchy signal flow through cables. The important property of logical-to-physical correspondence of the HDL is violated.
2. As hierarchy-crossing cables are added during the design process, constant HDL maintenance is necessary on all components of a hierarchy tree inside which lie the targets of such cable connections. Packaging data for different levels (e.g., card, MCM) typically has different owners. The insertion of these cables creates many additional interdependencies and slows down change management.

Bugspray

Bugspray is a proprietary language extension to VHDL that was designed originally to serve as a mechanism to add nonphysical specifications for the verification process to the HDL model. Bugspray annotations are coded as formalized comments inside VHDL, which makes them invisible for standard VHDL parsing.

There are Bugspray annotations that serve as directives to collect simulation coverage information and verification assertions. The benefits of these features are briefly described in [2] and are not covered in this paper because the focus here is on the simulation of packaging design data. The coverage and assertion mechanisms use the same infrastructure specified below, but are otherwise completely independent.

Bugspray also supports the instantiation of virtual components called Bugspray modules. A Bugspray module is basically a VHDL entity/architecture with some extensions, again encoded in formalized comments. Just like any VHDL entity/architecture, a Bugspray module has input and output ports. It is instantiated into a target design, the design entity/architecture in which its input ports are connected to design signals [**Figure 8(a)**].

When a Bugspray module is used to collect simulation coverage information or to specify verification assertions, the output ports of the module have predefined semantics to support that functionality. For example, every assertion drives an output port whose logic value indicates whether the corresponding assertion has been violated. In a simulation model, all fail ports of all Bugspray modules are connected via a Boolean `OR` expression, driving a single *fail* signal, which indicates whether any assertion has been violated. Concentrating all fail information to a single master fail signal allows a testbench to very efficiently check whether the design had any assertion violations during simulation.

Figure 9(a) illustrates one of two ways to instantiate a Bugspray module inside a target design. A stylized comment with a special start indicator (`--!!`) inside the VHDL architecture specifies the name of the Bugspray module to be instantiated. When the VHDL compiler is called with the command line option that invokes Bugspray processing, the `--!!` comments are recognized, and an instance of `my_bugspray` (with an instance name matching the name of the Bugspray module) is created inside the architecture.

Figure 9(b) shows the VHDL for the corresponding Bugspray module. The module shows the standard VHDL entity, architecture, and port declarations. Note the output port *fails*, which is interpreted by the model-build software in the special way described above.

Inside the architecture there is a section of stylized comments delimited by the keywords `bugspray` and `end bugspray`. Inside this section, the compiler finds the target design into which this module is to be instantiated.

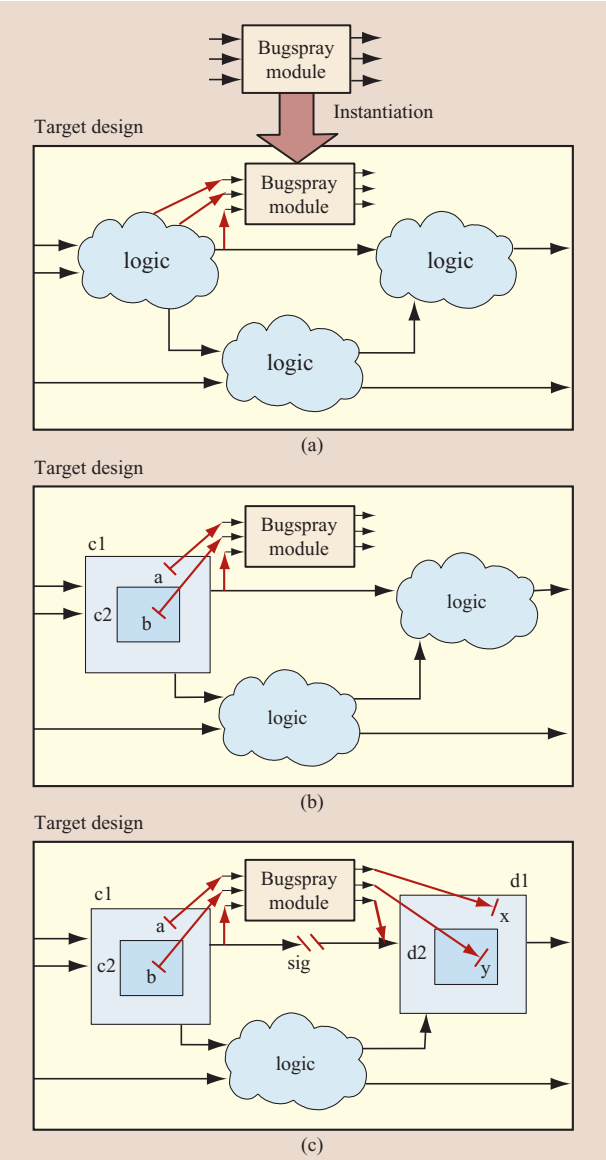

Figure 8

(a) Bugspray module instantiation. (b) Bugspray module connections that tunnel through the VHDL hierarchy. (c) Bugspray module with driver output connections.

Figure 9(a) shows the mode in which we instantiate a bugspray module directly inside the target design. While this method lets the designer clearly hook in Bugspray checker modules directly, it has the disadvantage that any addition of a Bugspray module requires a change in the source file of the target design. To support a Bugspray annotation mode that leaves design source files untouched,

```
ENTITY target_design  IS
     PORT(...);
END target_design;

ARCHITECTURE arch OF target_design  IS
     SIGNAL a: std_ulogic_vector(0 to 7);
     SIGNAL b: std_ulogic;
BEGIN
     --!! bugspray module : my_bugspray;
     .....
END arch;
```

(a)

```
ENTITY my_bugspray IS
     PORT(in1 : IN std_ulogic_vector(0 to 1);
          in2 : IN std_ulogic;
          fails: OUT std_ulogic_vector(0 to 0)
     );
END my_bugspray;

ARCHITECTURE arch OF my_bugspray IS

     --!! bugspray design : target_design;

     --!! INPUTS
     --!!     in1(0 to 1) => a(5 to 6);
     --!!     in2         => b;
     --!! END INPUTS;
     --!! end bugspray;

BEGIN
...
END arch;
```

(b)

```
ENTITY my_bugspray IS
     PORT(in1 : IN std_ulogic_vector(0 to 1);
          in2 : IN std_ulogic;
       fails: OUT std_ulogic_vector(0 to 0)
     );
END my_bugspray;

ARCHITECTURE arch OF my_bugspray IS

     --!! bugspray design : target_design;
     --!! INPUTS
     --!!          in1(0 to 1) => c1.a(5 to 6);
     --!!          in2         => c1.c2.b;
     --!! END INPUTS;

     --!! end bugspray;

BEGIN
...
END arch;
```

(c)

Figure 9

(a) Target design VHDL annotated with an instance of a Bugspray module. (b) Example 1 for a Bugspray module. (c) Example 2 for a Bugspray module.

```
ENTITY my_bugspray IS
     PORT(in1 : IN std_ulogic;
          in2 : IN std_ulogic;
          out1: OUT std_ulogic;
          out2: OUT std_ulogic;
          out3: OUT std_ulogic;
     );
END my_bugspray;

ARCHITECTURE arch OF my_bugspray IS

     --!! bugspray design : target_design;

     --!! INPUTS
     --!!     in1   => c1.a;
     --!!     in2   => c1.c2.b;
     --!!     in3   => sig;
     --!! END INPUTS;

     --!! DRIVER OUTPUTS
     --!!      out1 => d1.x;
     --!!      out2 => d1.d2.y;
     --!!      out3 => sig;

     --!! end bugspray;

BEGIN
     out1  <= in1; out2 <= in3; out3 <= in2;
END arch;
```

Figure 10

Example 3 for a Bugspray module.

the VHDL compiler also accepts a simple list of Bugspray modules. The `bugspray design:` specification inside the Bugspray section of the module anchors the module in the correct target design.

Every instantiation of a module requires that it specify which signals are to be connected to the input ports of the module. This is done inside the `INPUTS . . . END INPUTS` section in Figure 9(b). This is syntactically similar to a VHDL port map in that we require a list of formal ports associated with the actual signals in the target design. All design signals are visible by name in exactly the same way as they would be if the Bugspray module had been instantiated directly inside the target design.

Figure 8(b) shows a special feature available for port maps of Bugspray modules. The target design in this example has a component hierarchy inside, with component `c2` instantiated inside `c1`. The Bugspray port allows the user to access signals not only from inside the target design, but also anywhere inside the component hierarchy that is anchored by the target design.

Conceptually, this puts the author of a Bugspray module in a hierarchical name scope with visibility to all signals from the target design downward. If a module port

has to be connected to a signal inside a hierarchically nested component, a hierarchical naming scheme, which uses component instance names as prefixes, is used to disambiguate the name of the signal. **Figure 9(c)** shows an example of the syntax. The use of hierarchical target signal names allows a user to *tunnel* through the component hierarchy of the design without forcing explicit ports on the design components.

Bugspray driver outputs

Bugspray support to instantiate virtual components and the signal tunneling feature created a solid base to support the special needs required by packaging data specifications, especially cross-hierarchy cables, as described above.

We extended the Bugspray features by adding user-definable output ports that are allowed to drive signals inside the target design. The same signal scope rules apply to these driver output signal port maps as were established for the input ports, specifically the capability to tunnel through the design hierarchy without requiring explicit ports.

The Bugspray module in **Figure 8(c)** reaches into components `c1` and `c2` to connect signals to input ports. The module now also has driver output ports that connect to signals on three different levels of the component hierarchy, including a tunneling access into components `d1` and `d2`. The model-build software connects driver outputs in such a way that any original source that was connected to the target signal is *replaced* by the signal coming from the Bugspray module, e.g., signal `sig` in Figure 8(c). **Figure 10** shows the corresponding VHDL for this example.

To summarize, the special needs of packaging data representation and simulation required some proprietary extensions to the VHDL language with the Bugspray mechanism. While the overwhelming part of a Bugspray module is still written in VHDL, an organic, VHDL-style extension has been implemented that lets a user define virtual, simulation-only components and connections that cross hierarchies, such as the cross-packaging cables in the hardware.

Concluding remarks

In this paper we have described a methodology that enables the verification team to build simulation models based on packaging design data in an efficient, flexible, and less error-prone way.

A correct and complete model structure is the basis for extensive logic verification. This is achieved by generating the structural model data directly from packaging design. This process generates the data for the whole system structure.

VHDL configuration provides the efficiency and flexibility to derive, in a nondestructive way, different system structures from this complete structure for the various needs of the verification team. With an additional tool, called Bugspray, cross-hierarchical connections and additional logic can be inserted into the system structure. This flexibility can be used to cover special needs in system simulation model build.

References

1. H. Kohler, "Design of a Multichip Module Containing a 12Way S/390 Microprocessor Subsystem," presented at the Ninth Annual IEEE International ASIC Conference, 1996.
2. J. M. Ludden, W. Roesner, G. M. Heiling, J. R. Reysa, J. R. Jackson, B.-L. Chu, M. L. Behm, J. R. Baumgartner, R. D. Peterson, J. Abdulhafiz, W. E. Bucy, J. H. Klaus, D. J. Klema, T. N. Le, F. D. Lewis, P. E. Milling, L. A. McConville, B. S. Nelson, V. Paruthi, T. W. Pouarz, A. D. Romonosky, J. Stuecheli, K. D. Thompson, D. W. Victor, and B. Wile, "Functional Verification of the POWER4 Microprocessor and POWER4 Multiprocessor Systems," *IBM J. Res. & Dev.* **46,** No. 1, 53–76 (January 2002).
3. *IEEE Standard VHDL Language Reference Manual, IEEE STD 1076-1993*, IEEE Press, New York, 1993.
4. The Chip Hierarchical Design System Technical Data Standard; see *http://www.si2.org/index_files/si2_publications.htm#CHDStdThe_Chip_Hierarchical_Design/*.
5. J. Darringer, E. Davidson, D. Hathaway, B. Koenemann, M. Lavin, J. Morrell, K. Rahmat, W. Roesner, E. Schanzenbach, G. Tellez, and L. Trevillyan, "EDA in IBM: Past, Present and Future," *IEEE Trans. Computer-Aided Design* **19,** No. 12, 1476–1497 (December 2000).

Received September 22, 2003; accepted for publication November 3, 2003; Internet publication April 6, 2004

Hans-Werner Anderson *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (anderson@de.ibm.com).* Mr. Anderson is an Advisory Engineer in the eServer zSeries Processor Development Group. In 1985, he received his Dipl.-Ing. degree in electrical engineering from the Technical University of Stuttgart. He joined IBM in 1985 and works on chip physical design, logic simulation, and verification tools.

Hans Kriese *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (hkriese@de.ibm.com).* Mr. Kriese is an Advisory Engineer in the eServer zSeries Processor Development Group. In 1973, he received a diploma in physics from the University of Tuebingen. He joined IBM in 1973 and works on logic and packaging design entry, logic simulation, and verification tools.

Wolfgang Roesner *IBM Systems and Technology Group, 11400 Burnet Road, Austin, Texas 78758 (wolfgang@us.ibm.com).* Dr. Roesner is an IBM Distinguished Engineer. He is the technical leader for System Group verification tools development, and the verification methodology leader for the next-generation server. He received his Dipl.-Ing. and Dr.-Ing. degrees from the University of Kaiserslautern in 1980 and 1983, respectively. Dr. Roesner developed simulators and hardware design languages at IBM in Boeblingen, Germany, later joining the POWER processor development team, where he co-developed the TexSim simulation system. His verification tools have been used on all IBM CMOS microprocessor projects, and since 1996 he has been responsible for the strategy of verification tools development. Dr. Roesner has received three IBM Outstanding Achievement Awards and one IBM Corporate Award for development in hardware design language processing and simulation.

Klaus-Dieter Schubert *IBM Systems and Technology Group, IBM Deutschland Entwicklung GmbH, Schoenaicherstrasse 220, 71032 Boeblingen, Germany (kdschube@de.ibm.com).* Mr. Schubert is a Senior Technical Staff Member in the Hardware Development organization in the Boeblingen laboratories. He received his M.S. degree in electrical engineering in 1990 from Stuttgart University. He subsequently joined IBM in Boeblingen and has been responsible for hardware verification of multiple S/390* systems. He was the technical leader for the hardware verification of the z900 2064 system and is currently responsible for the system verification including the VPO activities for all eServers including the z990 system. Mr. Schubert holds three patents and has received two IBM Outstanding Technical Achievement Awards for his work on zSeries verification.