Controlling Change Propagation and Project Policies in IC Design

Yves Mathys, Marc Morgan, Salma Soudagar

Semiconductor Systems Design Technologies, Motorola Inc.

Abstract

Today's large IC designs involve highly partitioned, highly coupled and voluminous design data evolving over time. Tracking design state is becoming an essential component of design tracking systems.

In this paper we present the project BluePrint, a design data flow management tool which is an extension to the DAMOCLES tracking system. The project BluePrint is event driven. It defines project data dependencies and controls change propagation. The BluePrint allows the separation of project specific information from tool activities facilitating tool integration. The project BluePrint follows the non-obstructive approach championed by the DAMOCLES tracking system.

1: Introduction

Today's IC design process incorporates additional tools to automate the process, to improve design quality or to provide better power and timing analysis. Design data is viewed from different perspectives. For instance, the descriptions of an IC circuit as seen through the specification by the customer, the system designer, the logic designer or the layout engineer are all different but must be correlated. The increasing number of EDA tools and of design representations, also called design *views*, complicates the tracking of the project state for designers. By project state, we mean information about the data reflecting its consistency and validation related to a design flow, expressing how tools and data are correlated. As the complexity of the design flow increases, it becomes critical for design tracking systems to capture and to manage complex design flows [Katz85,Cas90,Liu90,Sil89]. The success of a tracking system depends heavily on its ability to accommodate a variety of design flows and project methodologies. Tool integration and minimal system tracking overhead are critical issues for a tracking system.

In this paper, we present a design data flow management tool, the *project BluePrint*, an extension to our tracking system for managing IC design development, DAMOCLES [Va92,Ma92]. The project BluePrint enumerates the views which are tracked, describes the relationships between views, specifies how design state changes are propagated across data dependencies, and which actions are performed upon each change detection. Thus, the project BluePrint formalizes design methodologies and enables their reuse.

As in the NELSIS data flow tool [Wolf90], we divided the project BluePrint information into:

- the **configuration information** which specifies the design data views tracked in the project flow and the relations between them, and
- the **run-time information** which controls how design activities modify the state of the project data.

We extended the configuration information of the project BluePrint by adding the meta-data model consisting of a set of properties associated to each view and the inheritance scheme used for version control.

We present a new concept of run-time information, **the run-time engine**, which controls change propagation across relationships. This engine is an **event driven** machine. Design activities transmit information (such as the modification of design data, or designer information about the interpretation of simulation results) to the BluePrint by sending events through the computer network. Upon reception of a design event, the run-time engine propagates throughout the meta-data the event by selectively traversing the data relationships. When a change propagation occurs, the state of the design is updated instantly. Designers can retrieve the state of the project by performing queries. Therefore, designers know exactly what data still needs to be modified before reaching a planned state in the project.

The event driven model simplifies the interface between the EDA tool set and the tracking system. The tools are encapsulated in wrapper programs which are independent of the design flow. The BluePrint allows to capture the entire information about the design flow and to implement design policies for enforcing the project methodology.

The rest of the paper is organized as follows. In the next section, we present the DAMOCLES meta-database. Section 3 discusses in detail the project BluePrint and its integration into the DAMOCLES tracking system. Section 4 gives a quick comparison of DAMOCLES to related work.

2: DAMOCLES meta-database

The DAMOCLES system relies on a database, where information about the design data is stored. This meta-database modelizes the project data and the relationship among design views. The meta-data model defines the **design's state** for each particular design view. DAMOCLES manages data repositories, called work-spaces by associating them to a meta-database.

The DAMOCLES meta-database contains information about the design data. To each design object corresponds a meta-data object (referenced by an *OID*, Object Identifier), which is defined by a triplet of block-name, view-type and version number.

The relationship between the design objects are represented in the meta-database by *Links*. Different types of relations can be specified, such as hierarchy, derivation, dependency, equivalence, etc. A Link object can be annotated by property/value pairs. DAMOCLES distinguishes between two classes of Links: use links which represent hierarchy and derive links which represent other relationships (derivation, etc.).

Links are used in DAMOCLES to propagate events from one OID to another. The events, which are produced by design activities, can be propagated in either direction through the Link. Each Link has a PROPAGATE property which enumerates events which are allowed to propagate through it.

The third type of meta-data objects are *Configurations*, which consist of a set of database addresses, referencing OIDs and Links. This implementation results in light weight configuration objects, which can be used to store results of volume queries.

The Configuration management mechanism combines a version history of different data blocks into one configuration instance which can be considered as a higher level of description of data across time. Configurations can be used to save the state of the design hierarchy in a snapshot at each step of the design cycle. They can be built by traversing a hierarchy while following certain rules, or can be made as a result of a query, in which case they will be a non-hierarchical set of data.

3: The project BluePrint

3.1: Tracking information flow

The information produced by data transactions and tool activities, such as creation, deletion, validation or modification, is used to track the state of the design.

The integration of design tools into DAMOCLES should be kept simple since we face a large number of tools. The invocation of the tools is encapsulated into shell scripts called wrapper programs. These scripts post event messages to the BluePrint. An event message consists of an event name, a propagation direction (either up or down through the links), a target OID and optional arguments:

postEvent ckin up reg,verilog,4 "logic sim passed"

As shown in the figure below, the design activities are converted to events and sent to the project BluePrint, where they are queued. The BluePrint engine processes the queue of events by applying the BluePrint instructions to the target OID and updating the meta-data information. Events are processed sequentially, first-in first-out.

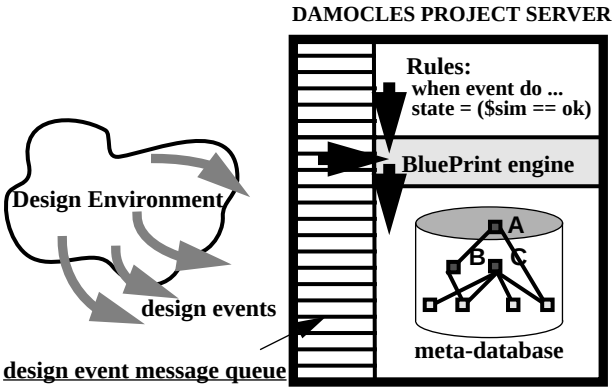


Figure 1 BluePrint architecture

3.2: BluePrint description

Prior to processing any event, the BluePrint must be initialized by the project administrator; this is done by reading in an ASCII file which contains a set of rules which the BluePrint applies to the meta-database upon reception of each event. Two types of rules are supported: template rules, which describe the configuration information, and run-time rules, which apply to the run-time engine.

Configuration information

Template rules enumerate for each view the properties and links which should be attached to the view. For example, a view LayoutGDSII might have a property DRC and a link which indicates that LayoutGDSII was derived from the view EdifNetlist.

Template rules are used by the BluePrint to setup new OIDs and Links as they are created by design activities. Each time the BluePrint is informed of a new OID being created, it finds the corresponding view in the BluePrint and attaches properties and Links to the new OID. It should be noted that OIDs are instances of views defined in the BluePrint. These new properties can either be copied or moved from the previous version of the OID, or simply

created on the new version. Property names are nearly all defined by the project administrator although certain generic property names are strongly recommended.

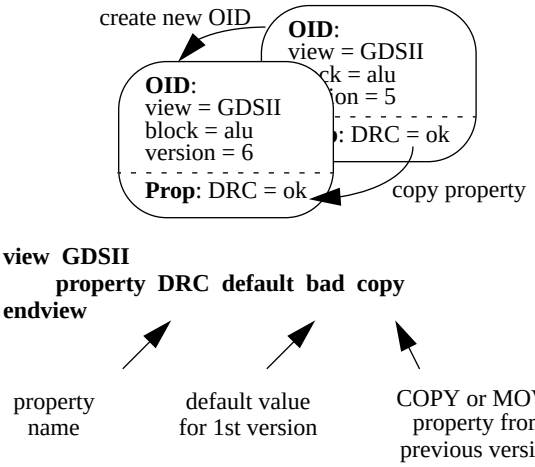


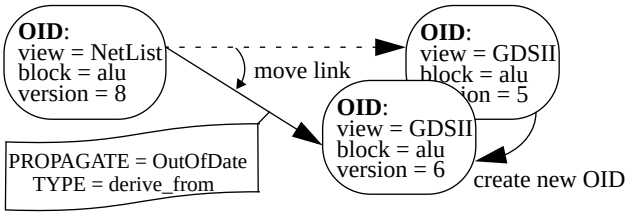
Figure 2 Example of a template rule for a property of view GDSII

Run-time engine

In the same way, each time the BluePrint is informed of a new Link being created, it finds the corresponding link in the BluePrint and attaches the template properties to the new Link. The main property of a Link is named PROPAGATE and contains a list of events which can propagate through instances of the link. Derive links also have a TYPE property which specifies the type of relationship the link expresses. A link's type is not directly used by the BluePrint. Link types are, in a way, like comments which help the user in visualizing the data flow and propagation model. Common types of derive links are:

- **composition**, which models the hierarchical decomposition of data,
- **equivalence**, which ties alternative representations together (e.g. VerilogNetList and EdifNetlist) (see also the equivalence plane in [Katz86]),
- **depend-on**, which expresses dependance on a tool version or a process file,
- **derive-from**, which expresses that a data view is derived from another view.

Note that the use link does not specify a parent view name; since a use link represents hierarchy within a view, the parent and child views of the use link are of the same view type (e.g. a use link might link OID <cpu, SCHEMA, 4> to its hierarchical component <reg, SCHEMA, 2>).



view GDSII
link_from NetList propagates OutOfDate type derive_from MOVE
endview

Figure 3 Example of a template rule for a derive link from view NetList to view GDSII

The design state of an OID is given by the value of the OID's property. When several properties are attached to the same OID, the state of the OID can be given by a continuous assignment combining the value of several properties (e.g. my_state = ($simulation == ok) and ($DRC == good)). Such an assignment is continuously being reevaluated. Continuous assignments are defined in the template rules of a view, like properties and links.

Aside from the template rules, the BluePrint supports run-time rules which specify what action is to be performed each time a new event is received:

- the properties of the target OID can be assigned new values.
  E.g.: **when** checkin **do** oid_is_checked_out = false; last_check_in_date = $date **done**

- a script can be executed (i.e. to send warnings to users, to invoke tools)
  E.g.: **when** checkin **do notify** "$owner: Your oid $OID has been modified" **done**

- a new event can be posted to a specific OID (as in example 1 below) or directly propagated from the current OID (as in example 2):
  E.g.1: **when** checkin **do post** behavioral_sim_ok **down to** VerilogNetList **done**
  E.g.2: **when** checkin **do post** out_of_date **up done**

The examples above show what actions might be taken when a checkin event is received.

When the BluePrint receives an event X which is targeted at an OID Y, it processes this event in the following manner. The run-time engine starts by finding the target OID Y in the meta-database, and the corresponding view and run-time rules in the BluePrint. Any run-time rules

with assign actions are then executed and all continuous assignments of the OID are reevaluated. The next step consists in invoking the scripts which are listed in the exec run-time rules. Finally, the run-time rules which post new events are executed. Having executed all three types of run-time rules, the run-time engine can proceed in propagating the event X as well as any new event which was posted by a post-type run-time rule.

The propagation of an event from a target OID T to other OIDs in the meta-database first consists in finding all the links of OID T. Then for each link, the event is passed on to the OID at the other end of the link if the link propagates the given type of event and if the direction of the link matches the up or down direction specified in the event message. This process is repeated for each OID receiving an event.

Different BluePrints can be defined for each project, or for each phase of a project, by writing a new set of rules in an ASCII file and re-initializing the BluePrint mechanism. In this way, early in the design cycle, when the data has not yet been validated and changes occur very often, the BluePrint can be "loosened" thereby limiting change propagation.

3.3: Tool scheduling

Tool scheduling is implemented by the wrapper programs. The program queries the meta-database, requesting the permission to access data and to run the tool. The permission is given based on the state of the input data. For example, prior to running a simulation, the wrapper makes sure that the input netlist is up to date.

The run-time information specifies the action to be performed upon the reception of a design event. This simple, yet powerful, scheme leads naturally to implementing automatic tool invocation. Let's take an example where the netlister has to be invoked every time a new version of schematic is promoted (checked in) to the project workspace. The run-time rule would be:

when ckin **do exec** netlister.sh "$OID" **done**

where the netlister.sh is a shell script invoking the netlister tool and $OID is a built-in environment variable specifying the schematic OID which received the ckin event.

Tool scheduling supports partially or fully automated design flows which reduce both the risk of errors and the design cycle time.

3.4: Example of a BluePrint

This section discusses the case of a simple design flow. The figures below show a classical representation of the flow, which is based on tools and views, and the representation for the BluePrint, which is based on views, links and event messages. The golden view of this design flow is the schematic which can be generated automatically by synthesis and/or manually with the editor.

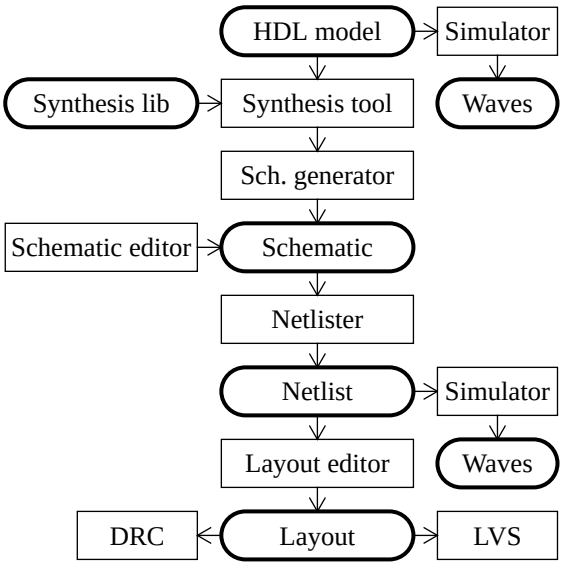


Figure 4 Classical representation of a sample design flow

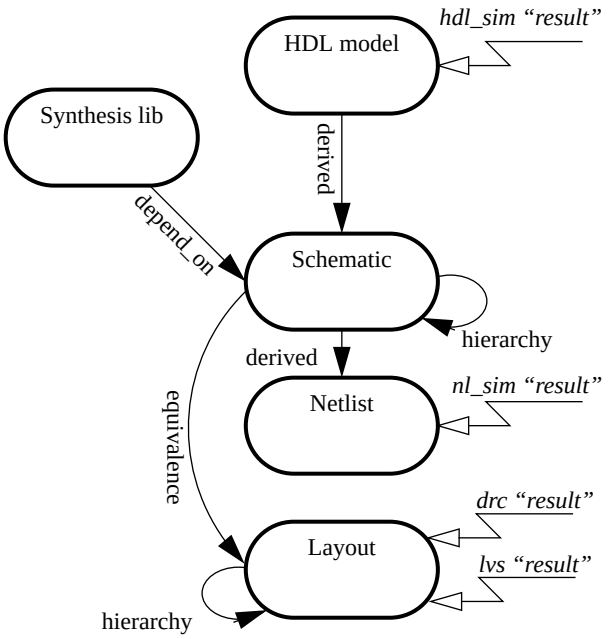


Figure 5 BluePrint representation of the same design flow

In this example, the project administrator has chosen to track five views. The views for the output of simulations were deliberately left out and replaced by event messages which indicate how the results were interpreted by a designer. The synthesis library is tracked so that the installation of a new version of the library will automatically invalidate data which depends on it. The netlist view is tracked in order to receive the event message with the result of simulation.

Lets take a closer look at a typical scenario. A group of designers starts out by writing an HDL model for their new design. The top block name is CPU. So they create an OID <CPU.HDL_model.1>. They then simulate the model and get a negative result. In order to tag the OID as not passing simulation, we add a sim_result property to the HDL_model view in the BluePrint. This property has a value of "bad" each time a new OID is created and is modified each time an hdl_sim event is received. The syntax for the HDL_model is therefore:

property sim_result **default** bad
when *hdl_sim* **do** sim_result = **$arg done**

The variable $arg contains the message passed by the wrapper program of the simulator. It could typically contain messages like "4 errors" or "good".

The designers then modify their model and save it as a new version <CPU.HDL_model.2>. They run the simulation again and this time get a "good" result. They then synthesize the design from their model. This creates OIDs <CPU.schematic.1> and <REG.schematic.1>. The second OID is part of the hierarchy of the CPU schematic. It has a use link (hierarchical link) which points to it from the CPU schematic. Now the designers look at their CPU schematic and decide to change part of the design so they modify their HDL model thereby creating a new OID <CPU.HDL_model.3>. In order to mark the schematic as being out of date, we put a derived link in the BluePrint between the HDL model and the schematic and we have this link propagate an outofdate event. In this way, when they check in their new model <CPU.HDL_model.3>, the ckin event is used to post an outofdate event to all the derived views. This is implemented in the BluePrint by adding a run-time rule to the HDL_model which posts an outofdate event to all views which are linked to the HDL model upon reception of a check in event:

when *ckin* **do post** *outofdate* **down done**

and by adding to the schematic view an uptodate property and a run-time rule which takes into account the outofdate event:

property uptodate **default** true
when *outofdate* **do** uptodate = false **done**

The uptodate property has a default value of "true" so that each time the designers check in a new version of the schematic, the uptodate property will be set to "true". In fact, these two rules are added to all the views (or rather to the special default view which applies to all the views).

The syntax for the schematic view would also include the hierarchical use link and the derived link from the HDL model which are mentioned above:

use_link move propagates outofdate
link_from HDL_model **move propagates** *outofdate*-**type** derived

The two links propagate the outofdate event so that when such an event is posted from CPU HDL_model, the CPU schematic and all of its hierarchical components receive the event. Both links are tagged with the **move** keyword to indicate that when a new version of an OID is created, these links are automatically shifted from the old version to the new version. For instance, if a new OID <REG.schematic.2> were created, the use link between <CPU.schematic.1> and <REG.schematic.1> would be shifted to link <CPU.schematic.1> to <REG.schematic.2>.

The BluePrint in this example has been set up to automatically create a new netlist each time a new schematic is checked in. This is done with the syntax:

when *ckin* **do exec** netlister "**$oid**" **done**

where $oid contains the name of the OID which was just checked in and is passed to the netlister script which is a wrapper program for the netlister tool. In this way, when the designers synthesize their design, the OID <CPU.netlist.1> is automatically created. In order to mark this OID as out of date when they modify either the CPU HDL model or its schematic, a derive link is added from the schematic view to the netlist view which propagates the outofdate event.

Having described most of the features of the BluePrint for this example, we include below the complete description of the BluePrint:

```
# note: keywords appear in bold and
# event names appear in italics
blueprint EDTC_example
view default
   property uptodate default true
   when ckin do uptodate = true; post outofdate down
      done
   when outofdate do uptodate = false done
endview
view HDL_model
   property sim_result default bad
   when hdl_sim do sim_result = $arg done
endview
view synth_lib
```

endview
view schematic
property nl_sim_res **default** bad
property lvs_res **default** not_equiv
let state = ($nl_sim_res == good) **and** ($lvs_res == is_equiv) **and** ($uptodate == true)
link_from HDL_model **propagates** *outofdate* **type** derived
link_from synth_lib **move propagates** *outofdate* **type** depend_on
use_link move propagates *outofdate*
when *nl_sim* **do** nl_sim_res = **$arg done**
when *ckin* **do** lvs_res = **"$oid** changed by **$user"**; **post** *lvs* **down "$lvs_res" done**
when *ckin* **do exec** netlister "**$oid**" **done**
view netlist
property sim_result **default** bad
link_from schematic **propagates** *nl_sim, outofdate* **type** derived
when *nl_sim* **do** sim_result = **$arg done**
endview
view layout
property drc_result **default** bad
property lvs_result **default** not_equiv
let state = ($drc_result == good) **and** ($lvs_result == is_equiv) **and** ($uptodate == true)
link_from schematic **propagates** *lvs, outofdate* **type** equivalence
when *drc* **do** drc_result = **$arg done**
when *lvs* **do** lvs_result = **$arg done**
when *ckin* **do** lvs_result = **"$oid** changed by **$user"**; **post** *lvs* **up "$lvs_result" done**
endview
endblueprint

4: Related work

In the NELSIS framework the data flow management is driven by design activities, whereas DAMOCLES has an observer approach to design flow control. This approach makes DAMOCLES a light weight system which is perceived as non obstructive to the designers since it does not impose a methodology.

In contrast with NELSIS, the project BluePrint provides a flexible scheme for controlling the propagation of design changes. This scheme allows to define the state of data as the result of a sequence of design tasks and to model a variety of relationships between the design views.

HILDA [Hil90] and ULYSSES [Ul89] have provided mechanisms for selecting the appropriate CAD tools to achieve current design goals. In practice, we found that designers prefer to have full control over design activities.

5: Conclusion

We introduced a project BluePrint concept capturing data flow information and providing full control over change propagation to the project administrator. The project BluePrint defines and maintains the definition of the state of the project.

Our approach differs from other works by including the meta-data model to the data flow definition and by providing a very flexible run-time engine which allows the propagation of design changes across the data relationships. This mechanism precisely captures a design flow and monitors the state of the data during the design process. The flexibility of the run-time engine allows to automate tool execution or to enforce tool scheduling. The separation of project policy specific information from tool activities leads to a generic interface which facilitates the tool integration.

A prototype of the project BluePrint has been developed and integrated to the DAMOCLES meta-data server. We are currently investigating ways to incorporate the notion of design tasks to the project BluePrint which gives a higher level of description of design activities and their environment. In addition, we are working on a graphical interface to visualize the design state relative to its flow.

6: References

[Cas90] Casotto, A. et al., *Design Management Based on Design Traces*, in 27th Design Automation Conference, pp.136-141, 1990.

[Hil90] F. Bretscheider et al., Knowledge Based Design Flow Management, Proc ICCAD 90 (1990).

[Katz85] Katz,R.H., *Information Management for Engineering Design*, Springer-Verlag, 1985.

[Katz86] Katz, RH, *A Version Server for Computer-Aided Design Data*, in 23rd. Design Automation Conference, pp 27-33, 1986

[Liu90] Liu,L.C. et al, *Design Data Management in a CAD Framework Environment*, in the 27th Design Automation Conference, pp 156-161, 1990.

[Ma92] Y. Mathys, V. Vasudevan, Tracking design methodol ogy in DAMOCLES, EDAC 92.

[Sil89] Mario Silva et al., *Protection and Versioning for OCT*, in 26th. Design Automation Conference, pp 264-269.

[Ul89] M. Bushnell et al., "Automated Design Tool Execution in the Ulysses Design Environment", IEEE, Trans. on Computer-Aided Design 8(3) March 1989.

[Va92] V. Vasudevan et al., *DAMOCLES, An Observer-Based Approach to Design Tracking*, ICCAD 92.

[Wolf90] P. Van der Wolf et al, *Meta data Management in the NELSIS CAD Framework*, ACM/IEEE Design Automation Conf. 1990.