CHDStd - A Model for Deep Submicron Design Tools

D. Cottrell
Silicon Integration Initiative
Austin, TX 78759
Tel: 512-342-2244
Fax: 512-342-2037
e-mail: cottrell@si2.org

D. Mallis
Software Experts
Austin, TX 78752
Tel: 512-451-7191
Fax: 512-342-2037
e-mail: mallis@cactus.org

J. Morrell
IBM Corporation
East Fishkill, NY 12533
Tel: 914-892-9196
Fax: 914-892-2066
e-mail: jmorrell@vnet.ibm.com

Abstract— SEMATECH, a US based consortium of ten major semiconductor manufacturers, is developing a comprehensive system for the design of ICs below .25 µm, which exploits hierarchy, constraint directives, incremental processing, and concurrent design and analysis. This development of SEMATECH's Chip Hierarchical Design System (CHDS) includes major technological investments in algorithms for design planning, parasitic extraction, and signal integrity verification. The foundation of CHDS is an open design model and API upon which these tools are integrated. This model must support a number of critical requirements including:

- **A data scope that includes connectivity, electrical data, physical data, and timing;**
- **Design hierarchy and incremental access to data;**
- **A central delay architecture;**
- **Efficient, application-selected views of the data.**

This paper introduces the Integrated Data Model technology being used for CHDS Beta development and its use as the basis for the design of an industry-open specification called the CHDS Technical Data Specification (CHDStd).

I. Introduction

In order to exploit the transistor capacity of ICs with shrinking feature sizes and increasing die sizes, engineering requires advanced EDA support that can improve design productivity by a factor of 2 to 3 every three years. Without the advent of significant algorithm advances, such as the paradigm shift due to synthesis, the best chances to achieve this improvement will probably come from design reuse and constraint-driven design tools exploiting incremental cooperation. To achieve the tighter levels of integration between EDA tools required by the latter, the need for a single design model, directly accessible by both logical and physical tools, becomes increasingly critical.

Critical design loops that depend on sequential processing using ASCII files for design data transfer will suffer significant file translation overhead as those files grow along with transistor counts [8]. Faster clock frequencies and larger die sizes will mandate more sophisticated interconnect modelling, particularly for global interconnects, to analyze cross-coupling effects of mutual capacitance and reflections due to inductive properties of transmission lines. Further, the number of interconnects that behave as transmission lines will increase at a rate higher than the total number of nets on the IC. Reliability problems, due to electromigration of interconnects, must also be analyzed as current densities along signal paths increase. All of these issues will add to increased complexity within IC design loops and the number of iterations required. Today, according to over 75% of respondents to a recent survey of SEMATECH companies, the biggest detriment to design productivity is the number of design iterations required. Tomorrow, this problem will be even worse without a paradigm shift in the design system architecture.

Access to a logically central repository eliminates non-functional data translations among design and analysis tools. Hierarchical representation and processing of the design and selected views of data within the hierarchy support incremental processing, which can reduce cycle times of design iterations. Constraint driven design and concurrent analysis of design changes can minimize the number of these iterations. A common information model, combined with standardized methods for data access and manipulation, facilitates inter-company design and tool reuse by creating portability across EDA systems.

The scope of the critical IC design loop covered by CHDS includes synthesis, floorplanning, parasitic extraction, noise analysis, timing, and power/clock network design, through physical layout [5]. The CHDStd is a specification of an information model [3] and an application programming interface (API) – a comprehensive set of C/C++ classes and methods used to create, modify, or analyze design data [7]. The CHDStd does not specify the actual format of a data repository, any data caching strategies, techniques for loading data from or saving it to persistent files, or any data repository implementation details.

CHDStd requirements are extensive and far-reaching [1].

High performance access to design data through a common API by such a broad set of applications is not something that can be developed without a great deal of foresight and detailed planning. Indeed, several attempts to develop sophisticated central database solutions for EDA have failed. Rather than attempt to build a new solution from scratch, SEMATECH chose to leverage industry-proven solutions. The central delay architecture is implemented with the Delay and Power Calculation System Draft Standard (IEEE 1481) [6]. Interfaces for parasitic extraction data are built from technology developed by Lucent Technologies. The CHDStd information model and API are based on modular technology developed by IBM, which is being used for advanced IC design both internal to IBM and by its semiconductor customers [2]. The IBM Integrated Data Model (IDM) system was selected from submissions to a Request for Technology from Si2. Its characteristics and features are described in the following paragraphs.

II. CHDStd Overview

The CHDStd is data-centric in that complete design information and interrelationships are centrally managed in memory, maintained persistently, and surfaced to applications via standardized access methods. Components include:

- a common model incorporating required IC design and constraint information;
- an API for access to data and relationships of that model hierarchically, incrementally, and by application-related subset;
- an API that implements a central delay (and power) architecture;
- an API for exchange of geometric and parasitic data between all-net and multi-net extractors;
- a syntax and API for defining and accessing (respectively) technology parameters and constraints;
- a syntax for defining engineering change order (ECO) information.

A. CHDStd Operational Model

The CHDStd Operational Model provides access to the design data elements for the IC. This data includes the original design, application derived data, and metadata elements that describe the design. The architecture provides a modular API that allows an application to select only certain subset views of that data, subsets of the hierarchy, or even individual objects, as is appropriate for its needs.

1. Design Data Scope

The CHDS system integrates design processes from early, high-level abstract physical design and timing estimation, through timing-driven synthesis, timing-driven detailed wiring, parasitic estimation, hierarchical timing back-annotation, and design rule verification. The objects within the CHDStd Operational Model include the netlist (both a folded, hierarchical instance view and a fully elaborated, hierarchical occurrence view), physical data, electrical data (including parasitics), shape information, technology models and constraints, and design assertions. The scope includes data currently supported by various EDA industry-standard interchange formats: specifically, EDIF, SDF, SPEF, PDEF and GDSII. Because of the advanced nature of the CHDS design and analysis tools, the Operational Model also includes data not common in today's EDA exchange standards. This data includes design objects such as parasitics based on process variations and parasitic models that accurately represent effects of mutual capacitance and inductance.

The organization of the CHDStd data allows information to be presented selectively to an application. Although the design repository contains a very broad range of data objects, an application need only work with a *view* of that data that has an information scope confined more closely to its requirements. It need only work with subsets of design information defined by data type or hierarchical depth (level of occurrence expansion), ignoring data not of immediate concern. For example, one application may require the occurrence view of the design unfolded down to the transistor level, plus all electrical properties, to perform its task. Yet, another application may only require the folded view of the netlist at the gate level, with the physical characteristics, but no electrical data.

2. Core Model: Folded and Occurrence Netlist

The IDM core model contains elements typically required by all CHDS applications. It provides the base on which additional data views (electrical, physical, layout) are built. There are two related parts of the netlist: the *folded* and *occurrence* models. Each of these provides methods to create, delete, read, modify and traverse its objects. The core model also provides data extension objects (such as property, group, rule_box, and rule_pin) and registration of event-based callbacks.

Folded Netlist

The folded (or instance) model provides the basic structural description (only the netlist) of the design (blocks, ports, nets, etc.). The term folded is used because each block is described only once, even though there may be multiple instances of it within the design, so the model appears to have been *folded* back upon itself. The fundamental folded model objects include the following:

- A *context* defines the domain of active objects – blocks, nets, etc. It provides an anchor point (container) for all objects in a design. An application can create multiple contexts, if desired, switching among them dynamically.

- Definition blocks (*def_box*) define each block's interface. A definition pin (*def_pin*) further describes the characteristics of each input and output of the block. Every block used in a model has a *def_box* associated with it. There is, however, only one *def_box* for each unique block (cell, macro) used in the design, independent of the number of times it is used. *Def_box* is analogous to *cell* within the EDIF-DR Core Model for Electronic Design (CMED) [4].

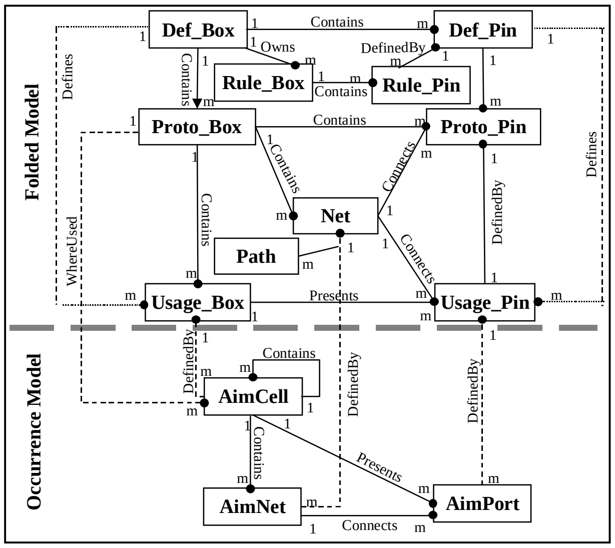

FIGURE 1: NETLIST MODEL

- Prototype blocks (*proto_box*) define the implementation details for a particular *def_box*. This includes the next hierarchical level of sub-blocks that are used, and the interconnections between them. Every *def_box* in memory that is not a *leaf* (lowest order node in a hierarchy) will have one or more *proto_box* objects associated with it. *Proto_box* is analogous to the *cell_representation* within CMED.

  Multiple *proto_box* objects can be used to define different implementations of the same *def_box* (each with the same I/O interface). This feature allows an application to manage multiple alternative representations of a *def_box* to exist concurrently during run-time and persistently in the repository.

  The *proto_box* has *proto_pins* that correspond to the *def_pins* on the *def_box* bound to it. These *proto_pin* objects provide the connection points to interconnects (*net* object), and are *ports* in CMED.

- Instance blocks (*usage_box*) are the sub-blocks used by *proto_box* objects. A *usage_box* is a particular instantiation of a *def_box* object, and is analogous to *instance* within CMED.

  The *usage_box* has *usage_pins* that can be connected to a net. A *usage_pin* compares with a *portInst* within CMED.

- *Net* objects describe how *proto_pin* and *usage_pin* objects are connected to each other to form a logical interconnect.

Occurrence Netlist

The occurrence (or articulated) model provides the expanded representation of the design, presenting a fully articulated set of occurrence-specific objects (blocks, pins, and nets). This is sometimes referred to as a "flattened" version of the netlist, however, in IDM all hierarchical boundaries and relationships remain, and their integrity within the Operational Model is maintained. Occurrences are full-fledged, data-managed objects. Within the occurrence model, each block and pin is represented as often as it is used in the design (as an *AimCell*, *AimPort*, respectively), along with the nets that interconnect them (*AimNet* objects). (CMED specifies how to expand the occurrence hierarchy via occurrence, port.occurrence, signal.occurrence.)

3. Facilities for Extensibility

The CHDStd must include features for application extension of the design model without requiring changes to the underlying model implementation. Extension facilities available within the IDM are being tested against real design situations as they arise within the CHDS development, and being analyzed for their applicability within the final CHDStd specification.

While private extensions to the model can be made on demand by any application for its own use, some extensions are generally useful for other applications, as well. Conventions for the names, data content, and semantics of specific extensions, such as properties, groups, and rule_box data, will be included formally as part of the CHDStd specification. This is done in order to support commonly needed features in an open plug-and-play environment.

Property

IDM provides a general-purpose *property* mechanism for adding application-defined information of arbitrary type (including character, integer, double, float, and application-defined structures) to most design objects. Properies can be made persistent or may exist only for the duration of the session in which they are created. Storage and retrieval of persistent property data is integrated with the standard load/save mechanisms for their owner objects (*def_box*, *proto_box*, etc.), so that other application processes can use this data (assuming it is intended to be public and there is agreement on its semantics) without separate I/O actions. Property data may be publicly advertised, or used only in a proprietary manner.

The property has proven to be a very important feature in support of quick extensions to the model independent of its implementation. For example, properties have been used by CHDS to extend the IDM Operational Model to contain decoupling adjustment factors and net sensitivity to glitches, which were not in the IDM definition. Further, properties can be used for metadata about the design. CHDStd uses properties on *proto_box*, *usage_box*, and *net* objects to provide persistent, incremental change information that allows downstream applications to process only those objects in the design that have changed since those applications were last run.

Rule_box

IDM also has *rule_box*, *rule_pin*, and associated objects that allow definition of one or more named sets of data extensions independent from the persistent load and save of other design data objects. Unlike property data, *rule_box* data will be loaded/saved only if an application requests it. This allows data of interest only to a certain class of tools to be available independently. Tools that don't need *rule_box* data do not have to pay any performance overhead for it.

IDM also allows association of custom I/O routines for load/save of *rule_box* data to/from a persistent repository, which may be physically separate from the netlist data. (Of course, public use of such a custom I/O scheme would require either knowledge of the data format or a linkable library implementing the access.)

Hide

IDM also provides an ability to *hide* data objects, so that they will not appear in normal iteration lists. Hiding power nets is a typical use, since some applications may not be prepared to deal with these, should they appear during a traversal of the design. Special iterators do exist, however, to allow identification of all objects, hidden or not.

Callbacks

IDM supports a *callback* feature that allows an application to register methods to be invoked on specific object events. For example, in an application that has been modularized to separate design and analysis subroutine components, a routine that analyzes interconnects can be invoked automatically whenever the pins on any interconnect are changed. Callbacks can be registered for add, delete or modify events on many objects. Examples include creation or destruction of a *usage_box*, connection of a *pin* to a *net*, or setting a particular *property* value. Pre-callbacks, invoked just before, and post-callbacks, invoked just after a specified event, are available. Callback registration includes the function to be called, optional application-data to be passed, a priority, and the "application name", which is a unique tag that identifies a callback as part of a set associated with a particular application function. This allows convenient disabling or removing of one set of callbacks without affecting other sets.

Within a single tool, program code can be easily modularized to take advantage of this event-driven processing. However, to open up such incremental cooperation between two separate programs, allowing plug-and-play of either, it would be necessary to standardize control interfaces among the component modules. This level of standardization has not been performed and is beyond the scope of the CHDStd. Therefore, plug-and-play substitution of application modules integrated in this way is only granular to the total set of modules sharing the single process space. Over time, standard control interface specifications for common service tools, such as timing analysis, noise analysis, and parasitic extraction, may be specified allowing these functions to be developed analogous to the Delay and Power Calculation Standard (DPCS).

Group

In certain cases, use of the callback feature can have adverse effects on performance. For example, a placement function might go through many iterations before it is ready for timing evaluation. Recalculating delay and analyzing path timing for each placement move, incrementally as it happens, could cause unacceptable performance degradation. Instead of using callbacks at an object level of granularity, application developers may wish to collect affected objects into a set, and then process that set when it is complete.

IDM provides a *group* mechanism for logically collecting objects. One application may, for example, create a group of all interconnects within a noise sensitivity range, so that a successor signal integrity application can access this subset of interconnects directly, without having to search for them throughout the design.

As an example, the parasitic extraction tool within CHDS has a choice of different extraction algorithms that can be invoked based on the required level of accuracy for an interconnect. Highly accurate extraction is required for global interconnects within a critical timing path, but this level of accuracy typically requires more processing time than does a lower accuracy algorithm. Local interconnects not within a critical path typically require a much less accurate extraction, and can take advantage of higher speed extraction algorithms. Within CHDS, design plan applications must communicate the required level of extraction accuracy via the persistent repository to a parasitic extractor. Accuracy values could be saved as net properties, but this would require the extractor to traverse all interconnects within the design to locate those accuracies of interest to it. Since the design planner traverses the nets needing extraction as a natural part of its processing, it can easily collect all interconnects requiring the same level of extraction accuracy together into groups. The extractor can then go right to the accuracy groups it wants and access the nets without a second traversal of the design. The use of group can be effective for a number of other similar situations as well, such as indicating sets of interconnects that need to be re-routed due to electromigration problems, etc.

4. Articulated Instance Model (AIM) Views

The classes, methods, functions, and objects comprising the folded and occurrence netlist models are the terse, minimal set required to represent the design structure. This information is augmented by additional collections of classes called *views*. Views contain additional methods and data objects that maintain appropriate relationships to base objects within the netlist model. Views elaborate the basic netlist with additional design information and methods for operating on it. The organization of these additional view classes was done to provide an optimal tradeoff between performance and memory. Views have been carefully constructed to be coarse enough to load efficiently, but granular enough to conserve memory by allowing applications to load only those parts of the design that are relevant to them. Design objects that are typically used together are grouped into a view, and an application may load zero or any number of views, depending on its requirements.

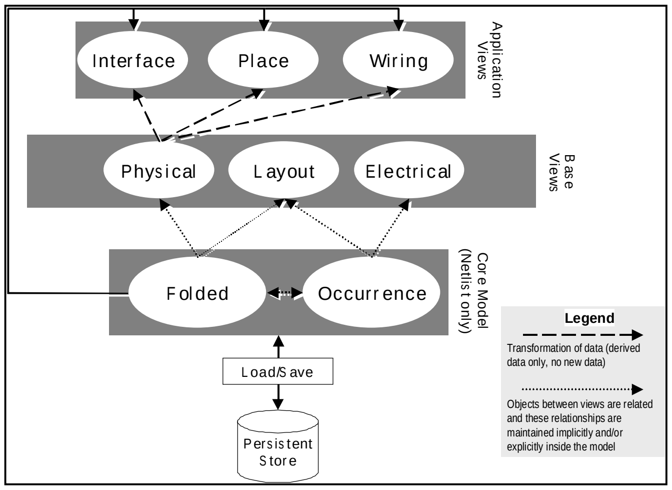
Interface
Place
Wiring
Application Views
Physical
Layout
Electrical
Base Views
Folded
Occurrence
Core Model (Netlist only)
Load/Save
Persistent Store
Legend
Transformation of data (derived data only, no new data)
Objects between views are related and these relationships are maintained implicitly and/or explicitly inside the model

FIGURE 2: IDM VIEWS

Two classes of views are defined within IDM. *Base views* define new design objects beyond those in the core model and present them to applications as a group of related information. *Application views* do not contain new design objects, but present data transformed from base views, plus application derived information, in a way designed to be more useful to a particular type of application. Some views are oriented toward specific kinds of tools. For example, a *wiring view* may be of use primarily to the different wiring tools available in a design process. Other views, such as an *interface view*, are likely to be needed by many different tools. IDM defines three application views, however, additional views could be defined via standard C++ subclassing.

Base Views

The base views manage application access to data relating to:

- Electrical – Electrical properties for blocks, pins and nets, such as parasitics, voltage drops current, delays; constraints information such as min or max capacitance for a net or max delay for a source-sink pin pair; technology related data, such as wire characteristics (ohms per length, farads per length, coupling characteristics, etc.), via and pad characteristics.

- Physical – Substrate characteristics, such as pad and via models, reserved areas, layers, levels, planes, constraints, technology information, and signal plane or porosity constraints.

- Layout – geometric coordinates and shapes.

Application Views

Three "higher-level" views manage data of the following types:

- Interface – (i.e. "external" description) constraints, legal locations, port area/shapes, and terminal port initialization.

- Placement – (and floorplanning) constraints, images (including hierarchical), snap-to-grid, aspect ratios, fixed-in-location properties, move/blocked bounds, wiring buffers, floorplan groups/stacks, checkpoints, external circuits, placeability.

- Wiring – images, porosity, power, constraints, wire/via models, power shapes, area/net labels, spacing, blockages, demand, congestion, wiring buffers (for placement), quad trees, and wiring layers.

5. Model Services

Implicit and explicit services in IDM maintain data integrity and relationships among objects in the Operational Model in the face of complex run-time changes to design data. These include creation/deletion/change of objects, load/store data between the Operational Model and the persistent repository (at the granularity of both the hierarchical and individual box object levels), elaboration of occurrence model objects, selective view expansion, and technology specification. A set of memory management functions also allows application control over in-memory resources.

B. Auxiliary Interfaces

Two auxiliary interfaces augment the Operational Model.

1. PDL (Physical Design Language)

PDL is an ASCII language format for specifying technology parameters and constraints. A standard file syntax provides a convenient way to transfer the technology characteristics with a design, augmenting design reuse. PDL includes:

- Technology and package groundrules – wiring layers, placement area, wiring area, I/O area, terminal placement, via and pad models, wiring models and constraints, power models and constraints, placement models and constraints, and any pre-placed elements.

- Constraints – cell size, placement and wiring porosity, net length, net resistance, net capacitance, and net delay.

- Application areas – placement and power area constraints.

IDM Model Services automatically populate the application selected data views with most generally required PDL data elements, such as constraints and wire/via models. Additionally, API functions are supplied that allow applications to directly and selectively read any of the data elements specified by PDL.

2. ECO (Engineering Change Order)

ECO is an ASCII command language for engineering change order specifications against the design model. Changes can be specified to add, delete, or modify cells, ports, nets and properties. Using ECO, changes to the design need not be committed immediately but can be saved persistently for later use. ECO is used to record the results and sign-offs required for these changes in a managed way across the design team.

III. Interfacing with IDM

Application developers can choose from a number of methods by which they can interface with IDM. Each of these methods has certain strengths and weaknesses with respect to performance and applicability to specific problems.

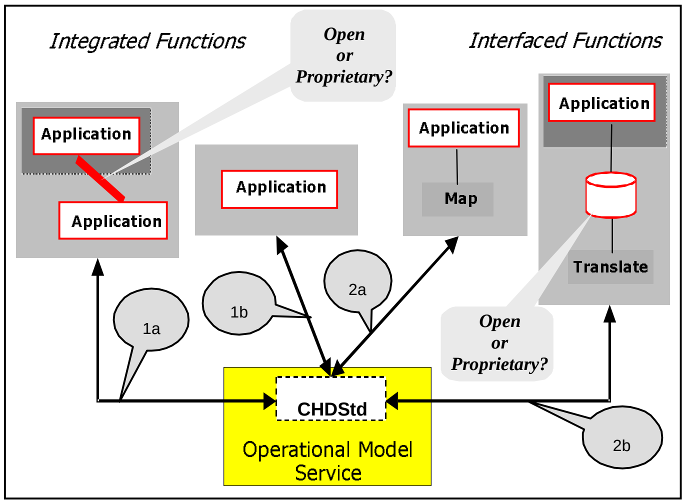

FIGURE 3: MODEL INTEGRATION

A. Integrated Applications

"Integrated applications" are those with core algorithms that have been modified to use the IDM Operational Model as the primary (native), in-memory representation. (This does not preclude the use of auxiliary, tool-specific data structures, e.g., for performance and algorithm reasons.) There are two variations on this method:

(a) Two or more applications in a single executable (and, therefore, the same process space) share the same, in-memory Operational Model. This allows a high degree of incremental cooperation on the design data. One application can be made aware of changes made to the design by another application by means of callbacks. The level of granularity can be down to a change event on a single model element, as soon as it happens or at any arbitrary point in the process. This eliminates storing/reloading design information from the repository between iterations.

For this to work, EDA functions originally existing as physically separate application programs, must be configured into modules of a single program and managed by driver code. Such a configuration allows exploitation of very high performance incremental processing. This interface technique offers the greatest potential for performance improvement in the “tight loops” of a design flow.

(b) An application uses the IDM Operational Model for its analysis, communicating with other tool processes solely via the persistent repository. This could be accomplished in a nested execution scenario (where one tool fork/execs another) if design changes are first saved to the repository. Incremental change information can be communicated between these applications only via the persistent repository (via properties, or groups, for example).

B. *Interfaced Applications*

In this style of integration, applications use the API to access design data from the logically central repository, but that data is translated to the application’s native format on which its algorithms operate. This technique is likely to be used as a migration strategy to allow rapid interfacing, without having to change core product code in the short term. The degree of integration with the application’s core algorithms may vary:

(a) The application uses its own in-memory representation of the design, but still uses the IDM API as part of a native code layer for access to the persistent data.

(b) None of the original application code is modified to use the API. Instead, data import/export tasks from the persistent repository are handled by a separate, stand-alone translation utility which, in turn, interfaces to the application’s native data format. The translation, including management of any intermediate files (whether in proprietary or industry-standard formats) must be automated and transparent to the user. Though this method may add additional overhead, it may be the easiest way to port existing tools to CHDStd.

IV. Summary and Conclusions

A goal of the CHDS Program is to achieve a 4X productivity gain over existing design systems while supporting an effective capability for "best of kind" tool substitution within a design flow (plug-and-play). In measuring that productivity, it is important to move from a focus on individual point-tool performance to a more realistic measurement of the total time from design-start to silicon. A faster tool algorithm may not help the overall flow if it has to wait while a two-gigabyte ASCII exchange file is being created and then re-parsed. Nor will it help if some other step in the flow must be repeated because adequate timing information is not available early enough, or if the lack of incremental capabilities force redundant processing of design data.

The IDM central repository and open API architecture offers the means to meet the CHDS productivity goals. However, loosely-coupled applications that do not eliminate the costly data translations, or exploit those flow-oriented IDM features will fall short in achieving those goals. Applications that treat CHDStd as just another interchange format may fail to attain the desired benefits.

Legacy migration may necessitate some applications being only marginally integrated to CHDStd. Business considerations may dictate that an application be quickly interfaced to CHDStd and more tightly bound later. Consequently, the CHDStd architecture supports these different levels of integration, as well. However, for the biggest productivity gains, applications need to exploit the power of tight integration to a common model, incremental processing, hierarchical design techniques, and use of event-driven callbacks to support critical design loops. Otherwise, the overhead required to support a plug-and-play API and the data integrity demands of a common model will not be leveraged effectively.

IDM has been providing valuable experience about what features are more or less critical to an open-industry solution. Certain IDM capabilities may still be too advanced to incorporate into CHDStd without definition of additional standards. Commercial adoption and support may require some changes to the existing model, and gate other features from near-term acceptance. The sheer richness of the specification may have an impact on the time required for commercialization. Selection of which features should be promoted to CHDStd and those that should not, will be based on both technical and business considerations. The final CHDStd specification is, therefore, subject to refinement as the CHDS Beta testing takes place across 1998.

V. Acknowledgements

We are indebted to IBM for the donation of IDM as the basis for development of the CHDStd specification, and as the CHDS Beta development vehicle. We thank P. Adams, J. Morrell, P.T. Patel, and J. Sayah for their contributions to the development of IDM and its documentation.

VI. References

[1] S. Grout et al, *SEMATECH ECAD Program CHDS Technical Data (CHDStd)*, 1995, unpublished (http://www.sematech.org).

[2] IBM, *An Integrated Data Model for Hierarchical Design Assembly*, 1996, unpublished (http://www.si2.org/CHDStd).

[3] A. Williams (University of Manchester) and David Barton (Intermetrics), *Information Model of IDM, Version 9/01/97*, 1997, unpublished (http://www.si2.org/CHDStd).

[4] H.Kahn (University of Manchester), "Design representation in EDIF version 3 0 0 and CFI version 1.0", Electronic Design Automation Frameworks Volume 4, Chapman & Hall, 1995, ISBN 0 412 71010 2.

[5] R.G.Bushroe et al, "Chip Hierarchical Design System (CHDS): A foundation for timing-driven design into the 21st century", SEMATECH, 1997, ACM 0-89791-927-0/97/0.

[6] Si2, *Delay and Power Calculation System*, IEEE Project Authorization Request No. 1481, 1997, in press (http://www.si2.org/dcl).

[7] Si2, *CHDStd Reference Specification*, 1997, in press.

[8] D. Mallis, "Benchmark studies of bulk data transfer methods", Si2, 1995 (ftp://si2.org/public/si2/Information/itc-benchmarks.ps).