Facilitating EDA Flow Interoperability with the OpenAccess Design Database

Mark Bales

Cadence Design Systems, San Jose, CA, USA

Abstract

Building multi-vendor interoperable EDA design flows today is difficult if not impossible. Historically, EDA companies have designed their systems to keep end users captive wherever possible. This has the long-term effect of reducing competition and innovation in the EDA industry, to the detriment of the EDA user community, the EDA developers, and their companies. It has kept the size of the overall EDA market smaller than it should be, and has led to the understanding that for each dollar spent on EDA tools and services, over three dollars is spent by EDA customers in integrating the tools from multiple vendors into a workable design flow [1]. If the EDA industry could do a better job at making this simpler, we could provide better products and services to our end customers that would lower their overall EDA bill, while making more money for our companies in the process.

This paper describes the requirements of interoperable design flows from both the user perspective and the application-developer perspective. It lists the dangers that can prevent a flow from working, even if all of the right foundational components are present. Next, the OpenAccess design database work is introduced, and emphasis is placed on its use in fulfilling the interoperability requirements that have been presented. The future work for the OpenAccess group over the next year is described, and the conclusion that an open shared database like OpenAccess is a necessary piece of technology to realize the dream of a multi-vendor fully-interoperable EDA design flow.

1 User Requirements of Interoperable Flows

The process of IC design turns an idea or concept into silicon. Throughout the design and implementation there are many different representations for the design data. It might start at an architectural or behavioral level and proceed through verification and synthesis. Physical implementation can include custom design, automatic placement and routing, and many different analysis and verification tools, ranging from parasitic extraction and physical verification to timing analysis and circuit simulation. Mask preparation can include OPC checks, PSM creation, and the data modifications done to GDSII-level data to ready it for mask creation. Ties to manufacturing and testing equipment carry along the design intent to allow more intelligent testing of the ICs during their construction. The type of the design may be pure digital (D), analog (A), analog/digital with a small analog content (D/a), or mixed-signal with a digital focus (D/A) or an analog focus (A/D). It is almost certain that multiple vendors' tools are used to create a design flow, and it is possible that the end customer has some significant tools that are a critical part of the flow. As we go from 180nm to 130nm to 90nm process nodes, the need for manufacturing-based knowledge at earlier stages of the design process grows. All of these dimensions combine to make it increasingly difficult to create the wide-ranging fully-integrated design flows needed for 130nm and smaller design. The following paragraphs present the challenges faced by the user in constructing interoperable flows, and details them in their approximate order of importance.

The primary requirement from the user point of view for an interoperable flow is a consistent representation of the design data from the start of the design through the end. Regardless of how many tools, databases, or interchange formats are used in the flow, if data is lost during a step, and then needed at a later step, there will be problems (Figure 1 (A)).

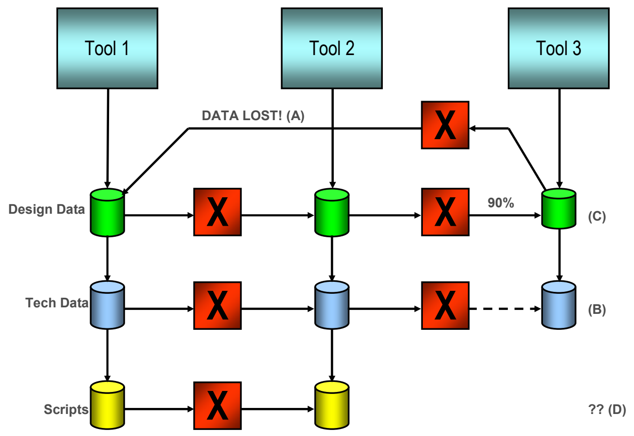

Figure 1

It might be needed to be able to back-annotate derived information to an earlier form (for example, parasitic-derived delays back to the original HDL netlist). It also might be needed to incorporate an ECO from earlier in the design flow. In these cases, the more information we keep about the design as it progresses through implementation, the better job we can do at providing an *incremental* ECO that perturbs the physical implementation as little as possible.

In addition to the design data, technology information must be carried along throughout the design process. This can range from fundamental process and geometric rules through complex electrical information used to guide analysis tools (for example, parasitic analysis). There are even more problems in this area than with the design data since many tools have different ways of looking at what is essentially the same technology data. If not accounted for, this can become painfully visible to the user, who must often enter data multiple times, in multiple formats, and is given the responsibility to manage this complexity with little or no assistance (Figure 1 (B)). Add to this the fact that as the data is transformed during the design and implementation process, the required technology information is also transformed in terms of complexity (accuracy) and form. Taken together, these issues highlight the need for consistent interpretation and use of technology information throughout a flow.

Related to the technology information but stored along with the design data are the constraints and assertions that capture the external requirements for the design. As the design progresses these are translated down to lower-level blocks within the design and abstracted to have meaning for specific tools and at specific stages within the flow. For example, an original timing constraint could become a signal-integrity constraint between two adjacent signals that would ultimately become a separation constraint between the routing that implements the two signals. In most systems today the linkage between the levels of abstraction is lost, leaving the management of constraints and assertions up to the end user. Also, inconsistent (or even non-existent) transformation of the constraints can result in them being ignored or misinterpreted by tools within the flow. This can lead to users being required to enter constraint information manually in multiple forms at different stages within the flow (Figure 1 (C)).

Having a set of tools that have a consistent set of capabilities and capacity has a great effect on the ability to create an interoperable flow. For example, if every tool except one in a flow can deal with blocks having a rectilinear boundary, it can make it difficult or impossible to construct a working flow for designs using rectilinear blocks. Also, if one of the tools in the flow is much less efficient in its use of memory than the others, its reduced capacity will make it difficult or impossible to build a flow that will work for practical sizes of design. It can be necessary to resort to 64-bit machines with large amounts of physical memory, and the resulting increase in hardware costs can make the entire flow less feasible economically.

Stability is of paramount importance when building a design flow. When building a multi-vendor flow in today's environment, the use of interchange formats and proprietary databases is necessary. Unfortunately, most vendors' interchange format readers and writers are probably the least stable parts of their systems, both from the standpoint of code reliability and performance as well as in inconsistent mapping between the interchange format and the internal system. Since many of these interchange formats either treat associated technology information in an ad hoc way or through separate side files, the robustness of transforming technology data through multi-vendor flows is even lower than that of design data.

A requirement that is somewhat less tangible involves the increasing use of scripting and extension languages in EDA tools. Tcl/tk seems to be the most popular such language, although Perl, Python, and many proprietary languages are also used. Even when a single language is used throughout the tools found in a given flow, a multi-vendor flow is likely to have different embedded command structures and forms in each tool. This results in scripts that are difficult to maintain. Since scripts often carry part of the design or constraint data during implementation, building a robust flow may require translation of scripts in the flow (Figure 1 (D)). This almost certainly is left as an exercise for the end user or (if they're lucky), their EDA department.

Complicating this already complex situation even further is the need to continually evolve to meet changing technology requirements. Most EDA users set up their flows for a given process node, and 10 years ago, this would suffice for 2 years or even longer. Today, we're seeing new process nodes every 12-18 months, and changes within the context of the same node point may require modification of a flow, and are coming at 6-month intervals.

One way of dealing with the technology-driven change, and a general capability that is needed to help make use of legacy tools, is a consistent extension mechanism. If a tool within the flow doesn't have the ability to contain some aspect of design, technology, or constraint information that is needed in a downstream tool, an extension mechanism can be used to store the data for later use. Similarly, if technology advances impose new requirements that are not yet generally available, the extension mechanisms can be used to store the new data and carry it throughout the flow to the few tools that need and use it. One problem that can occur in either of these cases is that if the stored extensions are related to the design data, they can become inconsistent as the design data is changed. This can be difficult to track or even detect, especially if the tool doing the design data changes is the one that has not even a conceptual knowledge of the extension.

A last point worth mentioning is a consistent look and feel in the tools within the flow. If simple user-interface items such as selection, key binding, menu picks, dialog interactions, etc. are very different from tool to tool within a flow, this will reduce the overall efficiency and hence productivity within the flow. Fortunately, with the ubiquitous position that enterprise PC software occupies in most companies today, more and more EDA tools operating paradigms are modeled after the PC tools, and this provides an unintended but very welcome consistency.

2 Application-developer Requirements

After looking at requirements from a user perspective, it is useful to look at what the requirements would be from an application-developer's point of view. We can look at data interoperability levels ranging from file or interchange format through a common in-memory model, and discuss which mechanism is appropriate in which circumstances. In addition we can look at the control-interface level for components within a part of an interoperable system. Given an underlying framework that supports the interoperability required at the data level, we'll discuss what is necessary at the algorithmic and application levels to insure interoperability.

File-level interoperability is the simplest kind (Figure 2).

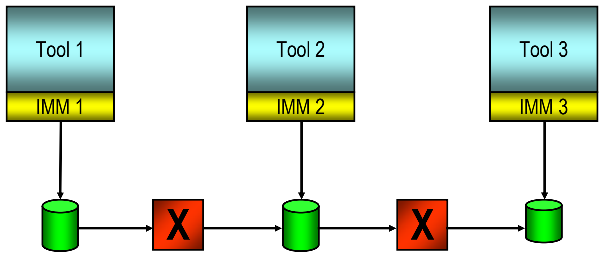

Figure 2

It has the benefit that it decouples the tools involved from each other and probably from the exact version of the interchange format used. It provides a readable format (in most cases) that can be used for purposes of debugging and testing. Using files presents many challenges and issues, however. The time needed to parse a readable format is usually much greater than that required to read a specific database file. The size of the readable files is usually much greater than the design data in a database format. Interchange formats act as "lossy filters" when compared to design databases. If a format doesn't contain a particular type of information, then a "round trip" through that format will lose possibly critical information. File-level interchange is most useful for applications that create a different form of output and are not responsible for round-trip fidelity of the data.

Using a database, but with a different in-memory model, treats it almost as an interchange format, but without several of the problems (Figure 3).

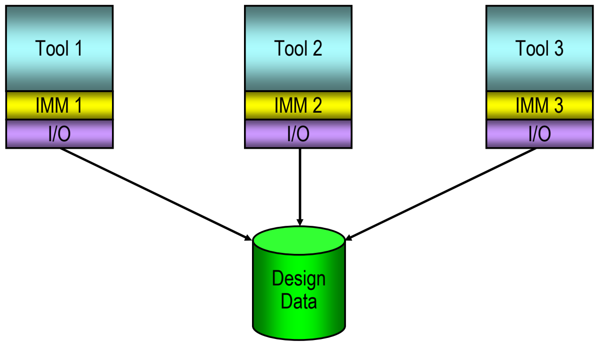

Figure 3

It shares the file-format advantage of allowing the tools to remain uncoupled. It does not directly provide a readable version of the data, but databases often provide a readable archiving or versioning format that can be used if need arises. Databases mitigate almost all of the problems found in file-based formats. They require little or no parsing, and are hence much faster to read. Applications only need be concerned with the relevant data and the database system manages the rest of the data. In this sense they aren't "lossy" like the file-based formats. Round-trip fidelity is achieved by updating the relevant information in the database system. The problems unique to database use arise when significant mapping must be done between the database information model and the in-memory model used by the application. This can make opening and saving a design too slow, and can even cause problems in the fidelity of the data mapping in the worst case. In addition, it is very difficult to share components among applications that use the same database but have different in-memory models. If this is a requirement you're much better off developing a common in-memory model.

Once you have a common in-memory model, you have the best of the worlds (**Error! Reference source not found.**).

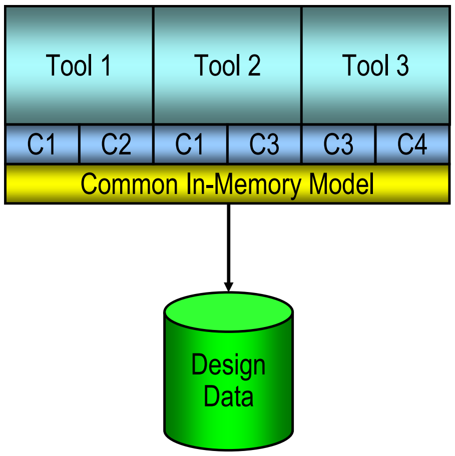

Figure 4

If you take the additional step of making the system components incremental and based upon the core in-memory model wherever possible, then you can collect an entire set of application components; technology and algorithmic engines that can be combined and reused in many different products. This lends consistency to the different tool suites within a design flow; reduces development and maintenance costs over time; makes for a modular system that can remain vital by easily upgrading modules one at a time; and provides a rich base for more rapid development of new products and technologies.

Some would think (with more than a bit of justification) that having a common in-memory model can slow progress. It is true that any sweeping change must be well planned and deployed. It is less true that additions must be well planned; they often benefit from a developmental deployment where an initial design is tested with actual products and refined before being more broadly deployed. The need for rapid progress can be met through use of the extension mechanisms mentioned in the previous section. This allows rapid prototyping, the ability to keep proprietary extensions private, and makes sure that the progress of the standard doesn't limit the ability of the tools to keep up with changes in technology.

Even though the common in-memory model provides the greatest gains, it is important to choose an integration method that is well matched to the task at hand. We continue to dismiss file-based formats for the problems previously mentioned. However, using a database as an exchange format (see Figure 3) is appropriate if:

- The subsystem isn't incremental.
- The I/O time as a fraction of the total run time is small.
- The components within the subsystem are built on a different in-memory model and switching would be expensive, or would cause a degradation in performance.

It is possible to build a highly productive design flow that uses tools plugged into a common database as a backplane that interfaces each tool in the flow in the most appropriate manner (Figure 5). When done in a careful and elegant way, the integration seems completely seamless.

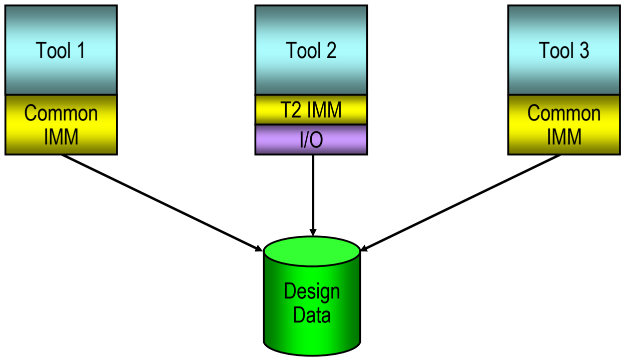

Figure 5

One advantage the common in-memory model gives over the use of the database as an interchange format is the ability to reuse system components. The biggest impact this makes on the overall design flow is the level of consistency and convergence it brings. Timing analysis gives the same answers right after synthesis (modulo a parasitic effect or two) as it does after physical implementation and extraction, *even if the analysis is done in different tools.* An additional capability not yet mentioned that makes it easier to share components is a set of common control interfaces for the components. If you standardize on the way a timing engine works it's easier to reuse it and easier to replace it when necessary.

3 Dangers affecting Interoperability

What factors can affect interoperability? Well, an incomplete database specification can keep different tools from interpreting the data correctly and consistently. It is possible that API implementations might not have enough error checking to detect cases of inconsistent interpretation. Dependence upon idiosyncratic behavior of the database system can cause some applications to fail when they are combined on a single database reference implementation, or the same application code may run well on one database implementation and fail to run on another. Overuse of localized extensions can lead to data that is no longer interoperable. This is especially true if any of the core data is being replaced. If the database system contains rapidly-changing APIs and/or semantics of the underlying data model, it can be difficult to maintain a functioning flow, especially in a multi-vendor environment. You would need to synchronize the multiple vendors' use of the design database, and this might be impossible if the development cycles of the groups contributing to the flow are widely different.

Lastly, it is important to recognize that an interoperable framework enables the construction of interoperable flows, but doesn't guarantee the end result. It is still easy to build flows that require re-entry of constraint and technology information, have inconsistent interpretation of design data semantics, use incompatible extensions that should be shared, and so forth. Application groups have to work very hard to create interoperable design flows.

4 The OpenAccess Design Database

Given the importance of a common design database in the EDA industry, the OpenAccess Coalition has been formed [2] to develop, deploy, and support an open-sourced EDA design database with shared control. The data model presented in the OA DB provides a unified model that currently extends from structural RTL through GDSII-level mask data. It provides a rich enough capability to support digital, analog, and mixed-signal design data. It provides technology data that can express foundry process design rules through 130nm, contains the definitions of the layers and purposes used in the design, definitions of VIAs and routing rules, definitions of operating points used for analysis, and so on.

The details of the data model and the OA DB implementation have been described previously [3]. This paper will concentrate on considerations of how the OpenAccess design database promotes interoperability and fulfils the requirements already presented.

5 Facilitating Interoperability with OpenAccess

OpenAccess has several fundamental features that make it easy for applications to write interoperable code. The OA extension capabilities allow applications to attach their own data structures to the database data structures, enabling the construction of incremental algorithms that communicate through the database. These same extensions in their persistent form allow applications to rapidly prototype new constructs, and to accelerate the pace of evolution of the system. Using extensions that may even be proprietary (we are each in business, after all), provides a legal way to divert from the standard while maintaining full compatibility and interoperability with other applications for the core of the standard database. Basing applications more directly on the OA in-memory model leads toward consistent tool capacity. Applications must actively drive towards consistent use of the db capabilities to insure workable tool flows. The stability and guaranteed migration of both the OA APIs and the design data created by OA insure that investments in application development are preserved over time. This also helps product developers assemble flows that use application components that might not all be using the same level of OA db APIs (**Error! Reference source not found.**).

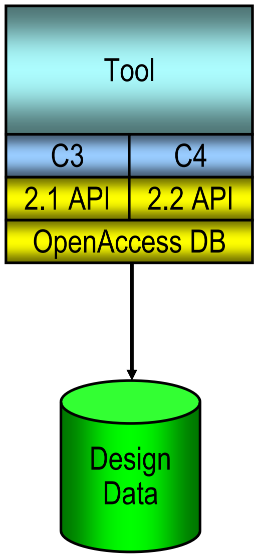

Figure 6

Lastly, extension language hooks for parameterized cells, user-extension management, and extension-language bindings are available and being used to foster interoperability through the availability of a common system language.

Non-persistent OA extensions can be used by applications to tie their application data structures to the corresponding database objects. Doing this gives a very high level of performance (but to be honest can never be as fast as a custom-built implementation created for the specific application). It also reduces the amount of memory required, as it isn't necessary to duplicate storage of data that is already in the database. The biggest advantage comes when the OA callbacks are used to manage application data structure changes in reaction to a database change. In this manner, multiple subsystems within a single process can interact and coexist entirely through interactions with the database. They never need to be directly cognizant of the other subsystem's presence.

OA extensions can be used in either a persistent or non-persistent fashion for prototyping. The OpenAccess community is comprised of EDA professionals with vast areas of knowledge and expertise. Each of us is interested in different advances that could be made with OA. The extension mechanisms provide a means for prototyping new features. They are almost identical in speed and memory performance to "native" features. This means that meaningful prototype information can be generated to not only demonstrate the functionality of a proposed feature, but also its expected memory usage and speed. This is a very powerful capability that should both accelerate the rate of progress within the OA community as well as providing the "escape valve" needed by most developers to make up for the direct control they trade away when choosing to use OpenAccess.

OA extensions can be used to provide a "legal" way of keeping some information proprietary. In the past, this desire to have advanced features and information in an EDA database has led almost every EDA company to design their own design database. As end users and their integration groups try to assemble working flows from a collection of different data models, databases, and interchange formats, this can become almost impossible. When using OpenAccess extensions, the core database is unchanged and is still fully interoperable with tools from any manufacturer. Only the extensions are not visible. This lets the OA user concentrate on building db models only for their extensions and lets them be as efficient as possible.

Building interoperable design flows using OpenAccess requires that applications do several things during their implementation. They must first agree to base their in-memory model using OA at its core. By avoiding duplication where possible and appropriate, we can keep a higher capacity for the applications, improve the performance by not requiring input and output phases for an application, and make sure that multiple subsystems cooperating in the same process don't have data synchronization troubles. The second thing applications can do is to be designed to work in an incremental way. In this way, small changes made by one subsystem won't require a large change or reanalysis by a cooperating subsystem. A final recommendation for applications is to use the database in a "legal" and consistent way. Sometimes (especially legacy) applications have a different model for some construct than is found in OA. Many application developers aren't sure where to put a piece of information, and decide to put it in whatever field appears "close" to what they need. In this case, "close" won't cut it, since the semantic meaning must be uniform throughout a flow or it won't work. The application should take the time to fully understand the usage of the system, and use the legal extension mechanisms to take care of anything that is necessary but that doesn't fit within the OA model.

OpenAccess guarantees wherever practical that the APIs provided will be upwards compatible from one release to the next. This means that while we are adding *new* functionality on an ongoing and rapid basis, we do not *change existing* APIs without a good reason. Required changes almost always follow changes in technology. As an example, between version 2.0 and version 2.1 of OpenAccess, support for TrueType fonts has been added, requiring a change to the *oaText* APIs to support the new capability. We try to keep these changes to a minimum, however.

In OpenAccess we always guarantee to be able to bring the data forward from one database version to the next. Major versions are released about every six months (at the current time), and are defined by virtue of their direct data incompatibility. Translators are provided to go from an older version to a newer one. When making such a translation in the context of a design flow, it's important to recognize that it is not possible to go back from a newer version to an older one. This is because there is almost always more data added from OA release to release. If we provided the means to go to an older version, it is likely that information would be lost. Due to the "drop-in" capability of OA (and the ability to recompile for new major versions), it should almost always be possible to upgrade a design flow to the newest version used by the tools in the flow and still keep it working, even if there is a mixture of old and new tools.

In OpenAccess, it is possible to tie in a scripting or extension language, such as Tcl/tk, Python, Perl, or Java. These languages (as well as the core OA language of C++) find use as implementation languages for parameterized (programmable) cells; as languages to guide and manage user extensions; and as command-level extension languages for a system built around OA. Parameterized cells (usually abbreviated to *pcell*) are a construct that has existed in the EDA industry for almost 20 years now. They are cells whose contents are not defined by a static netlist or layout but by a *program* that takes a parameter set that can be different for every instance of a given pcell. VLSI Logic, Cadence Design Systems, Mentor Graphics, Avant! (now part of Synopsys) and many other companies have long had these types of cells. The benefit within OpenAccess is that there is no requirement for a specific language so that the end users and application developers are free to add whatever language is best suited for or best liked by a particular company or group. The implementation within OA allows library data to be distributed containing the parameterized cells along with an encapsulation of the language subsystem. This data can be sent to another application that has no direct knowledge of the pcell language, and the whole system just works. A similar capability is available for maintenance of user-extension data, although this is typically handled in C++ rather than an interpreted extension language, for performance reasons. The same capability can be used as a command-level extension/scripting language for a system built using OA. In this case, a binding of the OA API into the extension language makes possible scripts that modify the database. The language subsystem provides some of the most powerful and system-useful features within OA.

6 Future Work

The work of the OpenAccess ChangeTeam is documented in a 3-5-year Roadmap that is developed on a yearly basis [4]. The major items on the near-term roadmap are Embedded Module Hierarchy, Constraints and Assertions (Timing support), and expanding down into the Universal Data Model level.

Embedded Module Hierarchy is a way to take a snapshot of a design hierarchy at a given point in time and be able to look at the data from three different perspectives; the Module domain, the Occurrence domain, and the Block domain. The Module domain (think Verilog Modules) represents a logical hierarchy. It is folded, and contains all of the levels of an original netlist. The Occurrence domain is an unfolded version of the Module domain. It is fully logical and contains no physical information (Figure 7).

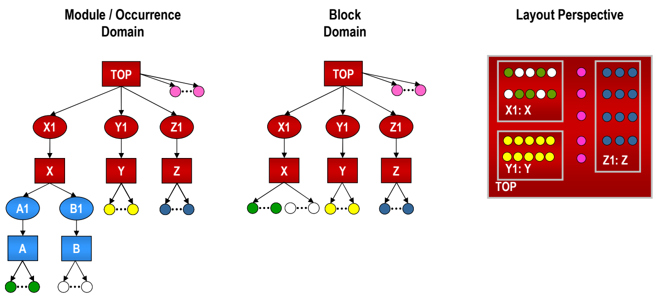

Figure 7

As changes are made in the Module domain, they are reflected forward into the Occurrence domain. When changes are made directly in the Occurrence domain, this causes uniquification of the Module occurrence and the changes are reflected backwards into the Module domain. The Block domain is folded, and most often contains the shape and connectivity information for a physical view. The three domains represent a single data hierarchy with three different ways of looking at the data. The "Embedded" term comes from the fact that some of the Module nodes are also Block nodes. The benefit of EMH is that it allows physically-oriented tools to look at multiple levels of logical hierarchy as though they had been flattened, without losing the information about the logical levels (including intermediate nets and terminals).

Constraints and assertions are a wide-ranging topic, covering items that are timing, electrical, and/or physical in nature. Work is proceeding to expand the capabilities of the OA Technology Database to represent constraints and assertions for timing and signal integrity. We're in the process of adding physical (foundry) rules for 90nm and 65nm process nodes. We expect to ultimately add support for timing models (.lib and/or ALF support, OLA support). The current state can support physical technology rules for 130nm. The goal is to be able to support RTL-to-Silicon technology rules for 65nm. This will be an ongoing effort with continuous improvement in each of the OA DB releases over the next several years.

Another area of great interest to the OpenAccess Coalition is the work taking place in the semiconductor mask-making and equipment areas called the *Universal Data Model*, or *UDM* (Figure 8 [5]).

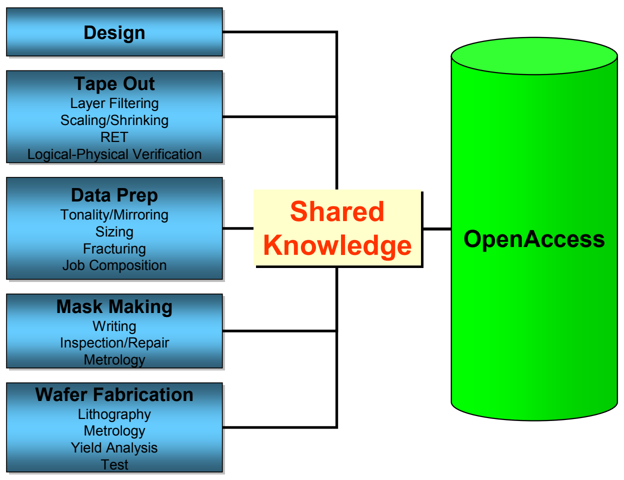

Figure 8 [5]

While OpenAccess currently supports IC design from RTL through GDSII levels, it stops short of supporting tasks from tape-out through wafer fabrication. Support will be added for tape-out tasks such as Layer Filtering, Scaling and Shrinking, and Reticle Enhancement Technology. Support for data preparation tasks such as Tonality and Mirroring, Sizing, Fracturing, and Job Composition will be added. Support for mask-making tasks such as Writing, Inspection and Repair, and Metrology will be added. Finally, support for wafer-fabrication tasks like Lithography, Yield Analysis, and Test will be added [5]. This is an ongoing topic, and the advances in this area will be evolutionary and will take place in each of the OA DB releases over the next several years. We will incorporate available standards wherever possible, such as the OASIS (or NSF) work, the SEMI P10 standard, and of course, the SEMI UDM work itself [6].

7 Conclusions

The requirements for interoperability take two main forms: framework capabilities and technology, and application-level resolve. OpenAccess provides the interoperability at the framework levels, and sets the stage for construction of highly-interoperable flows that use a variety of models. The models range from using the database as an interchange format, to using the database as an in-memory model. A greater level of integration with OpenAccess allows greater modularization and reuse of application level components in products, but is not required or even appropriate for each application within a flow. Lastly, even use of a common in-memory model is not a guarantee of interoperability. Applications must take continual care to make sure they have the correct semantic interpretation of the data and that they use the data in a way that enables use of the greatest number of downstream tools. If we make good use of infrastructure technology and keep clear application discipline, we should finally be able to build fully-interoperable multi-vendor flows.

References

[1] S. Schulz, "Open Access 2003 Conference Keynote", OpenAccess 2003 Conference, February 2003

[2] D. Cottrell and A. Graham, "What is OpenAccess?", http://www.si2.org/openaccess, January 2003

[3] J. Santos, "OpenAccess Architecture and Design Philosophy", OpenAccess 2002 Conference, April 2002

[4] OpenAccess Change Team, "OACT 2003 Roadmap", http://www.openeda.org/openaccess, April 2003

[5] D. Cottrell, "UDM Business Case", February 2003

[6] T. Grebinski, "SEMI IC Design/Photomask Data Path Task Force", International Sematech MASC Strategy Meeting, February 2003