OpenAccess

1H03 Technical Roadmap

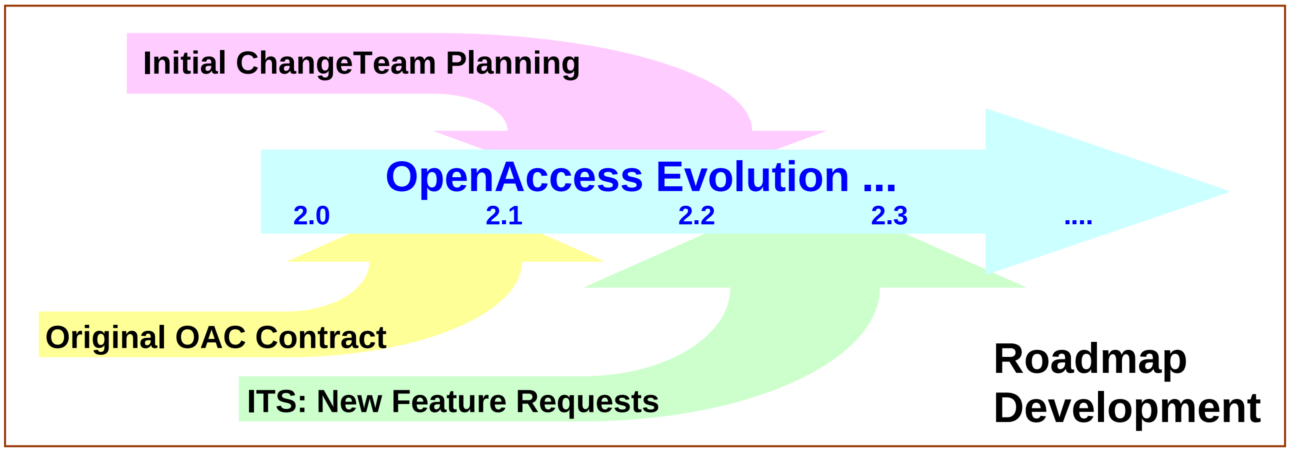
Reviewers

Representative	Company
Don Amundson	LSI Logic
Mark Bales	Cadence
Tim Ehrler	Philips
Bob Erickson	Synplicity
Michel Herment	ST Microelectronics
Aman Joshi	Sun
Joe Morrell	IBM
Joanne Schell	Motorola
Satish Venkatesan	Intel
Aleksandr Vlasov	Nassda
Jim Wilmore	HP

Table of Contents

1. Introduction

OA is already in the process of evolving beyond the original set of contractual requirements for specific API features. In accordance with the document that guides proposing and prioritizing changes – *OA ChangeTeam Change Process* – the OpenAccess Coalition ChangeTeam has agreed on the direction for the API over the next few releases, as driven by business requirements.



This document will be maintained on a continuous basis to provide an evolutionary view of the general order of planned OpenAccess development. It contains final descriptions for the features of the OA 2.1 (June 2003) release, paragraph-level descriptions for the features slated for the OA 2.2 (Dec 2003) and OA 2.3 (June 2004) releases, and paragraph-level descriptions for the features identified as being important over the next five years.

Note that the only committed items in this entire document are the details of the OA 2.1 plans. All remaining items are forward-looking proposals that will be finalized during the planning stages for the specific release in question. Some working groups (such as the DDM WG and Tech WG) are already in the process of analysis of some of these topics. In addition, it is expected that Community Contributions (fixes, enhancements, and/or new feature requests) posted in the ITS may be prioritized in addition to, or instead of, one or more of the currently planned items, as determined in accordance with the *OA ChangeTeam Change Process* document.

This document will be updated on a semi-yearly basis, approximately six months in advance of each release, to provide adequate opportunity for review and preparation by Coalition and Community developers.

2. OA 2.1 Release Feature Details (June, 2003)

2.1. Ensure OA 2.0 data compatibility

To support migration, OA 2.0 persistent data must be readable by OA 2.1. This requirement may be supported as a separate program or as a capability built into or layered on OA.

2.2. Add Embedded Module Hierarchy (EMH)

There is near-term demand (and a contractual requirement) for addition of an occurrence model to OpenAccess. In addition, there are anticipated needs to handle mixed module/block data to:

- Preserve the viability of pin test constraints after placement, optimization.
- Limit the scope of ECOs required as placement and optimization deviate from the original folded hierarchy.
- Communicate hierarchical boundary constraints for optimization.
- Allow generation of a netlist (e.g., in VHDL, Verilog) in the original module form.
- Enable back annotation of data (such as timing or parasitics) in the original module form.

Embedded Module Hierarchy provides the ability to represent a logic hierarchy and its flattened block hierarchy in a single cellView. EMH makes available three interrelated hierarchical views of a single design, any one or combination of which may be used according to the needs of a given application:

Hierarchy	**Representation**	**Domain**
Module	folded	logical connectivity, functional netlist
Occurrence	unfolded/flattened	logical plus physical analysis
Block	folded +uniquified	physical connectivity and implementation

The leaf (library) cells are not unfolded, but all of the levels of the design to the leaf cells are included. This simple definition is complicated by the desire to allow for block hierarchy, and to allow the block cellViews to be separate (for implementation by different groups).

The 2.1 implementation of EMH will provide module/block support with some restrictions and a full occurrence model for immediate use while setting up for smooth implementation of more sophisticated capabilities in a future release (see 4.1.). The EMH Functional Specification will be available for further details.

2.3. Support TrueType fonts

Currently, OpenAccess supports crude stroke fonts. They constitute a very limited set of fonts with very limited choices. A much more powerful solution would be to replace these stroke fonts with TrueType fonts. This would allow support of non-English languages, arbitrary scaling and rotation, and a much broader range of font choices. This may invalidate current oaFont usage for oaTextDisplays. The specification for TrueType support is available in Word format from: http://www.si2.org/oac/ChangeTeam/Files/Presentations/TrueTypeRev0.1.doc and in PDF from: http://www.si2.org/oac/ChangeTeam/Files/Presentations/TrueTypeRev0.1.pdf.

2.4. Add iterators to find all extensions

There is presently no API that enables location of all oaAppDef and oaAppObjectDef extensions that exist in a given cellView. This is required to enable:

- Purging a database of its extensions, since it may not be possible to know every possible one.
- Use of extension data by one application that was created by a different application in cases where there might be different versions or types of data that have different semantics.
- Export of the complete database in an external format.

Simple iterators for both categories of extensions would support these needs.

2.5. Change *realloc()* scheme for performance improvement

The current scheme of memory re-allocation does not work well with the implementation of *malloc()* and *free()* on some operating systems, leading to fragmentation and the inability to reuse freed memory. Modification of the way in which OA manages memory for large arrays, eliminating the use of *realloc()* to increase the array size, will minimize the likelihood of such fragmentation.

2.6. Improve virtual memory management by partial mapping

Currently, when reading and writing a database, OpenAccess allocates sufficient virtual memory to map the entire database file into memory, even when only part of the data is being read in. This could result in a capacity problem in some designs. Allocating only as much virtual memory as required to read or write a single table of data at a time will address this problem. (Partial mapping on read is already enabled in OA 2.0.1.)

2.7. Enable data loading by layer and purpose type

Accessing a single shape in the current implementation will cause all shapes to be read into memory. This can result in a capacity problem for designs with a very large number of shapes. The problem can be addressed by creating more memory efficient processing wherein applications do not pay the memory and I/O overhead for layers and purpose types that they do not access. Adding the capability to explicitly unload data for a particular combination of layer and purpose type no longer of interest will give applications control of such memory (as is currently possible with parasitic data). At this time, this capability will be added only for shapes.

2.8. Improve performance of the built-in Region Query implementation

The bigger the design, the more critical performance becomes for applications that need to perform Region Queries to locate figures. This is particularly important for designs with routes. This enhancement will allow much quicker access to route elements on a particular layer.

2.9. Implement a plug-In interface for custom Region Query implementations

Some applications may require a specially tuned way for implementing Region Queries to achieve performance beyond that of the Reference Implementation. This will be solved by adding to the API the ability to use application provided XY-Tree plug-ins to provide custom algorithms for such applications.

2.10. Support multi-threading

Parallelized algorithms hold the promise for high performance implementations of some design activities. A thread safe version of the OA implementation is required to support such tools.

Thread safety in this context means that any OA call will be guaranteed to return with database integrity intact. It does not mean that a context switch cannot happen in the middle of such a call (since a single call, such as loading a large number of shapes from the persistent store, would then have the potential for unnecessarily delaying execution of all threads). Thread safety also does not mean that the database will prevent thread collisions on shared resources. Proper serialization must also be implemented at the application level using the oaMutex objects or similar techniques.

A separate set of OA shared libraries with appropriate serialization for multi-threading will be created . A version of the shared libraries with no serialization overhead will also be provided, for applications specifically designed for single-thread execution. Platform independent methods for creating and manipulating threads will be defined.

2.11. Create ASCII format and load/dump utilities

While translator tools already exist (and more will be created) for converting OA design data to and from standard interchange formats (like LEF/DEF, Verilog, SPEF, etc.) none represents an exact image of OA data with all its semantics. There are motivations to define some form of human-readable export format for OA, and the load/dump code to process it:

Some designers depend on visual inspection, even editing, of design data for “second opinion” spot checks outside any application tool.

A native format import/export capability would enable a simple technique for moving designs from one OA implementation to another.

A versioned archival format (based upon XML) allows design teams to archive designs in a way that assures they may be recovered in the future, regardless of evolution of platforms, operating systems, machine word length, and so on.

2.12. Add debug checking and diagnostic aids

There is an architectural tradeoff between validity checking and performance. It is convenient for debugging an application to have the database check every argument of every function and return an immediate diagnostic when some value is in error. However, such checking costs time, and once the application is debugged, the expectation is that it will no longer feed bad data to the API, and hence will no longer be willing to put up with the cost of detailed checks.

The OA architectural solution is creation of multiple versions of the library distributions in each release, one of which is a special “debug version” (statically linked identified by a ‘_g’ suffix on the archive name). This debug version is statically linked, without compiler optimization, to simplify application of debugging tools. This architecture also provides a vehicle for moving some of the validity checking out of the optimized version of the libraries and into the debug version.

There are several kinds of checks currently performed in the optimized code that can be moved to the debug version to enhance performance of the “production” version of an application. The API specification will make it clear that there are two versions of an implementation: production and debug. Any exceptions thrown or other behavioral differences between the two will be clearly indicated in the documentation as to the version to which they apply.

2.13. Create a shape occurrence object

Classes of physical applications need a model wherein levels of physical hierarchy can appear as though flattened. Support for these applications will require a new transient object that represents a shape down in the hierarchy of a design as found by Region Query. A Shape Occurrence contains the shape itself, plus a hierarchical path to the shape. Shape Occurrence is needed for hierarchical selection.

Shape Occurrences will not be stored persistently. There will be no incompatible API changes with this feature added. Although EMH (2.2.) will provide an occurrence model, this version of it will not include Shape occurrences, so this transient technique will provide the needed support.

2.14. Add open (GNU) makefile support

OpenAccess 2.0 uses Cadence-proprietary makefiles that are not appropriate for open-community use. We will switch to gnumake-based makefiles. This change does not affect the OpenAccess API, but does require that OpenAccess be built somewhat differently than today. This will simplify the setup for community members that do their own builds.

3. OA 2.2 Release Planned Features (December, 2003)

3.1. Support timing / electrical constraints

Many EDA tools that analyze and optimize designs use timing and other design constraints. The ability to apply design constraints early in the flow and to have those constraints remain with the design through the implementation steps will reduce the cost of integrating EDA tools into a flow.

OpenAccess should provide API support for design constraints compatible with the common industry use of the Synopsys SDC format (http://www.synopsys.com/partners/tapin/sdc.html). In the first release, support for the subset of constraints which address clocking and timing is required. Other constraints are of lower priority. Additional support for reading and writing SDC is also required. *(Because the use of the SDC format is licensed by Synopsys, Si2 should address the legal issues involved in OpenAccess support of SDC.)*

3.2. Support view selection / binding (configurations)

Only in the simplest sense can one say that all CellViews are "derived" from a single Cell definition. Different CellViews provide a very wide range of different perspectives on the Cell and are necessary to support a wide range of different EDA applications used in IC design.

IC design typically uses a number of levels of design hierarchy where each child Inst in a parent CellView refers to a "master" CellView. For many types of EDA verification and analysis applications, it is necessary to switch between different CellViews of any child Inst's master CellView.

An example is traversal of oacSchematic CellViews, with Insts that are oacSchematicSymbol CellView types. Descent to the next level requires "switching" from the symbol to its schematic counterpart, but the current OA standard has no convention for what the name of that View is (and only one CellView may exist in a View, so they can't be grouped there).

The automated control that maps the methodology of the user community to select among alternative CellViews is referred to as "view selection."

Methods for controlling view selection commonly include:

- a set of simple rules such as:

  for all CellViews of type CellView-A, replace with type CellView-B

- an ordered list of preferred CellViews such as:

  ParasiticDetail, artworkConnectivity, schematicDetail, behavioralModel,

  where the selection uses the highest preferred CellView type available for each master CellView (although these would have to be "names" in OA, not oacCellViewTypes, unless that type list were extended).

- override rules to the above methods for specific describer Cells.

- override rules to the above methods for specific Instances.

An alternative, or extension, to the above rules can be addition of a callback to the API method that selects the master CellView from an Inst (and/or from an InstHeader). This is the most flexible approach, and actually can be applied by the methodology customizer to implement any or all of the above view-selection methods.

View selection also supports the termination of hierarchical descent, known as "Hierarchical Stopping Rules." One technique is simply to specify a CellView that is without children, effectively stopping the descent. Alternatively, certain master CellViews can be specified as "Leaf" or "Stopping" CellViews. Again, "Stopping Rules" can be implemented using hierarchical-traversal callbacks.

3.3. Support X route

X-routing is an emerging style of routing that expands upon the automatic placement and routing notion of "preferred direction" to include layers of metallization that have a preferred direction of plus or minus 45 degrees. By staggering the layers in 45-degree increments, two advantages are achieved. First, it is often possible to take the "hypotenuse route" from one point to another rather than a Manhattan distance, which is 30% longer. This can improve timing. The second advantage is that long parallel runs on separate layers are separated by 4 layers, not just 2. This improves signal integrity coupling between layers, and can make extraction and signal integrity analysis simpler.

Most of the support for X-routing is already in OA. This includes the ability to represent 45-degree routes and have 45-degree edges on polygons. Two things must be added to complete that support:

- Octagonal **vias** are needed, as they are often used at non-Manhattan intersections.
- The end types that are needed for X-routes are much more complicated than those required for Manhattan routing. These complex end types are octagonal in nature, but have their edges moved to insure that all edges are on the manufacturing grid, and to have acute (45-degree) angle intersections "filled in" to avoid any design rule violations. This includes chamfered corners and fillets.

3.4. Standardize manufacturing criticality knowledge

With shrinking feature size, the cost and elapsed times for mask making, yield ramp, and diagnosis/repair in manufacturing is beginning to gain attention as an area requiring change. One important part of the change deemed necessary for improvement is to provide a universal data model (UDM) that can transcend design, mask making, and wafer fabrication processes. OpenAccess is the obvious best candidate for this. To support the UDM it will be necessary to extend the OpenAccess data model in areas such as the following:

- Identification of non-design shapes used for reticle enhancement
- Identification of non-design shapes required for yield
- Mask order data (aka SEMI P10)
- Mask quality data
- Fault location mapping
- Information access control
- IP protection (see also 4.3.)
- Data transfer performance

3.5. Enhance the technology database

Smaller feature sizes and increased chip complexity require rules that more accurately model the technology details to better enable a more comprehensive and sophisticated checking of the design. In addition, OPC and PSM requirements encourage OA to extend more into the mask domain (see 3.4.). For the highest performance designs, critical circuit paths require technology details at a lower (circuit) level than has previously been needed. (Tied closely to this item may be the Processing Modeling item described in 3.6.).

Besides being extended "downwards" to address the needs of smaller feature sizes within the chip domain, OA should also be extended "upwards" to support higher levels of packaging in a manner which better enables overall product level optimization and analysis. More specifically, OA should provide the ability for applications to access design data which may span multiple levels of packaging. While accessing the data for this broader "design hierarchy", applications may need the ability to clearly understand at what points in the hierarchy the specific packaging boundaries are crossed. At the same time, OA must provide access to the appropriate technology modeling descriptions for each of the packaging levels so the corresponding design data can be correctly interpreted (especially needed for the physical aspects of the design).

3.6. Standardize process modeling for driving RLC extraction

A few years ago, Si2 sponsored an activity aimed at providing specifications for an open-industry standard format, known as the *Standard Interconnect Performance Parameters* (SIPPs – see http://www.si2.org/si2_publications/SIPPs/pdf/sipps_spec.pdf), that semiconductor foundries could use in supplying process characteristic data to circuit designers. All circuit design tools could then adhere to the SIPPs standard to ensure accurate physical representation of IC processes that affect interconnect performance and thus, could be used to perform the accurate parasitic extraction required by the increasing levels of circuit integration.

OA must determine how to align with and support the SIPPs modeling. The default might be to simply take the standard and augment the OA Technology information accordingly. However, during the development of the standard, some foundries took exception to providing this level of detail about their processes in an open manner because they considered it proprietary. An alternative approach would be to use the SIPPs modeling standard to describe their processes but the specific details would only be shared with, and subsequently encapsulated by, the developers of the corresponding extractors. This may mean that, rather than incorporating the standard, OA should:

- Review the current parasitic modeling to ensure that it covers all of the needs of the target extractors.

- Allow for the registration and invocation of a given foundry's (set of) proprietary extractor(s), perhaps through a service layer, as an inherent part of the "lazy evaluation" incremental processing.

3.7. Allow terminal ordering

In the OA Reference Implementation (and many other IC design databases and file formats) a parent Cell and each of its child Cells is persisted separately. This separation alone can lead to inconsistencies between parent and child when the design data is not kept in sync. One key issue that arises is potential mismatches between the parent Cell's connections to Instances of the child and the connectivity interface of the child.

When the OA API is used to create an Instance, only an Inst is created, but no InstTerms and no InstPins are created. APIs in many other database technologies choose to replicate the Terms and Pins of the child describer CellView as InstTerms and InstPins at the time of Inst creation. A utility method that provides this common functionality for Insts would best be written once, rather than written many times over by different applications.

When the proper child describer is available to the parent, the richer behavior of Inst creation that includes creation of InstTerms and InstPins will produce a hierarchical interface representation which aligns across hierarchy. However, when a different version of a CellView of the child replaces the one used for this automatic instantiation of InstTerms and InstPins, then it is quite possible that the hierarchical connectivity between parent and the new child CellView may be misaligned.

Furthermore, when the API supports View Selection for substituting many different child CellViews for the one initially bound to the parent's Instances of that describer Cell, the potential for connectivity mismatches across the hierarchy can increase dramatically.

As long as the parent's connectivity to a child Inst explicitly includes unique identification of each InstTerm or InstPin, the connectivity across the hierarchy to any View of the describer Cell with identical Terms and Pins can generally be established. (The only exceptional situations are when there are extra or missing Terms/Pins in the target child CellView.)

Some IC databases and some file formats representing hierarchical connectivity do not identify the child's InstTerms/InstPins uniquely in the parent definition. In such cases, an order of the connections is assumed or somehow designated as to establish the connectivity to the child. OpenAccess must define a means of communicating with such file formats and/or databases. This may be nothing more than a well-defined policy that is well documented, or it may involve some sort utility class which any given DB interface can customize to support its technique for Term/Pin alignment.

It should be noted that the mismatch difficulties mentioned above of accounting for a CellView with a different connectivity interface than that which was used for the Instances of its Cell in the parent will still be a challenge. However it is conceivable that the utility class for establishing hierarchical connectivity from a DB (or file format) that uses special conventions might be generalized to also support hierarchical connectivity for CellView replacements.

3.8. Support functional (digital) models

Functional models describe the operation of a library element such as a NAND gate, a flip-flop or more complicated cells. Functional models are used in synthesis, simulation, timing analysis, power analysis and other implementation and analysis tools.

OpenAccess should include an API to define and access the functional model for a CellView. This model should be a string defining the function in a format similar to Synopsys ".lib" format or Accelera's Advanced Library Format (ALF). If reasonable, the relevant OA API should be equivalent to the API specified in Open Library Architecture (OLA).

OpenAccess should include readers for any appropriate standard text formats used to specify the functional models for a library of CellViews.

3.9. Manage the development, distribution, and documentation of extensions

Extensions are expected to be required due to application requirements for proprietary data and data objects/attributes that other tools need to use but are too new to be standardized or are still working their way through the OA ChangeTeam release process. For such extensions created by one tool and used by another, a method must be defined and supported to advertise the purpose, syntax and semantics of such extensions. This might involve something as simple as a standard template format, to a more formal web-based "registry" of such data.

3.10. Improve documentation / training

To facilitate the adoption of OpenAccess by EDA developers, there must be readily available complete documentation and training materials. Documentation of various kinds is needed, such as, complete specifications of the API (covering syntax, semantics, behavior, dependencies and constraints), tutorials, and textbooks (suitable for training), etc. Similarly, training should be available at various levels, from basic overview (introductory) to a complete course of the entire standard for developers of OpenAccess applications and users of the reference implementation – in classroom format and as self-teach on-line training.

3.10.1 Complete specification of API semantics

True interoperability among tools requires more than linking to the same API functions. Applications must also subscribe to a "common understanding" of the significance of the data manipulated by the API.

This common understanding requires documentation at a level of detail beyond that currently available in the API specification. Additional effort must be applied to surface all constraints and semantics (including whether the database or the application enforces them) for:

- Every attribute. This includes all possible enum values (and their semantics), or other value constraints. (Example: A Net has a "priority". Define what this means, whether there are any behavioral differences of any other API based on this value, and if there are any requirements of interpretation for interoperability with other tools.)

- Every relationship. (Example: A Net has a “source”. Define what this is supposed to mean and if there are any dependencies on this by any other part of the API.)

In addition, flow of control changes must be clearly documented, including:

- Every exception and every action that will throw it.
- Every event that will invoke a callback.

3.10.2 Effective module / block use

The new EMH architecture implemented in 2.1 adds significant richness to the model, perhaps beyond what some developers typically use now. In addition to the syntax and constraint documentation of each of the new (and changed) APIs, it will be important to provide descriptions of the different parts of the new EMH Architecture – modules, blocks and occurrences – that communicate clearly the requirements for interoperability of the resulting data across tools.

3.10.3 Intended use of extensions

OA has a rich set of extensibility features, some of whose capabilities overlap. Already questions have arisen regarding which features to use in what circumstances, based on performance differences as well as convenience.

While most programmers have had experience with groups and properties in other database environments, the oaApp interfaces are relatively unfamiliar. Extended explanations and “best practices” examples showing use of application defined objects and extensions to native objects to solve problems (for example pin-to-pin delays, in the absence of native objects to store this data) would be of significant benefit.

In addition, the potential impact to interoperability when extensions outside the published API are used, and ways to cope with this issue, should be surfaced.

3.10.4 Working with multiple OA database versions

Design data portability has several dimensions that developers will need to address as they set up flows incorporating data and tools from an ever widening array of sources, and as design data is increasingly distributed.

It will be useful to explain techniques for achieving design portability across different versions of the Reference Implementation, different platforms running the same implementation, and even different implementations of the OA API via:

- Use of export/import tools for established interchange formats.
- Use of oaDump methods and/or the native ASCII format for OA.
- Use of ad hoc serializations and IPC between processes linked to different DB implementations to load/dump design data.

- Use of RPC layers for serialization across implementations.

3.10.5 Customizing the OA distribution with plug-ins

Functional layers can be written that link with the OA API to provide helper functions or services that naturally extend that API. Such layers can be used a prototype a design/implementation for eventual native integration into the API itself for higher performance (if the ChangeTeam approves).

3.10.6 Working with databases from multiple sources

Reusability of design data is the focus of much effort today. Many manufacturers are looking to incorporate bigger portions of a design from other sources (e.g., IP such as cores, and cell libraries). To assist the exchange of such data in OpenAccess formats, it will be necessary to have detailed descriptions of methods to merge OA database instances from different origins. This might include ways to resolve conflicting parameters in Tech databases, as well as semantic differences where the database doesn't enforce constraints. Issues of IP confidentiality (see 4.3.) must also be resolved.

3.10.7 Using callbacks for database-centric Inter-Tool Communication (ITC)

Callbacks can be exploited as an ITC mechanism to enable modularization of tasks into dynamically loaded or linked libraries (DLLs). This does create several issues that need to be addressed to allow standardized deployment of such DLLs:

- driver control requirements,
- standard entry points,
- "secondary" APIs (such as a "getDelay" API for a timer module),
- version control management.

3.10.8 Developing and distributing Pcells

OA provides a completely open-ended definition mechanism for programmable (parameterized) cellViews. There are no standard parameters or semantics; instead the application must define an evaluation engine that is called to flesh out the inst of a Pcell when needed.

It would be helpful for developers to see examples of, and issues involved with, the different ways of writing such evaluation engines – from self-contained OA functions to DLLs to extension language processors – and coping with portability issues that arise when there are dependencies on either compiled code or external evaluators.

3.11. Utility Items

3.11.1 Create data and “toolbox” items for the Community

To facilitate the adoption of OpenAccess by EDA developers, there should become available a rich set of aids to facilitate development and test of OpenAccess compliant software. Such aids may be available for free as open-source offerings or as part of marketable kits. The OpenAccess Coalition should encourage development and availability of such software and strive to assure that packages are available (for free or reasonable charge).

3.11.2 Example technology data, design libraries, design data

There have been repeated calls for sample data, both as examples of OA data model use, and for test evaluation purposes. Location and/or creation of small, medium and large examples of design data that can be distributed free or at low cost for the purpose of testing and benchmarking OA code and database implementations would help satisfy this demand.

3.11.3 Database viewers

There has been repeated demand for some tool to help visualize OA design data for training purposes as well as debugging programs and designs. Viewer functions are needed for following types of information:

- Module data: Folded logical netlist objects and connectivity, including their attributes, and relationships.
- Block data: Folded physical implementations, connectivity, and figures, including wires (implemented both as paths and as routes), standard and custom vias, other shapes (such as fills, reticles), floorplanning objects, etc.
- Occurrence data: Fully unfolded representations of objects and location-specific analysis data associated with them (parasitics, extensions objects).

Techniques for showing the correspondence points among the different views would be important, as would other reasonable viewing capabilities.

3.11.4 Text/graphical debugger

Hiding data members in the OA C++ classes is good for interface design, but can complicate viewing attributes and relationships of various objects in typical "dbx" sessions. A utility that simplifies debugging OA code would be of benefit to students and developers alike. This utility might be similar or related to (or make use of) the viewer code described in 3.11.3.

3.11.5 Netlist comparator

Engineers sometimes have a need to compare two design databases (or CellViews) for equivalence. A tool that would report differences (like the UNIX `diff` tool) would be useful in debugging version discrepancies, as a cross check for a design edit, and even as an instructional aid. Categories of differences that might be detected include:

- Recognize equivalence of sets of objects (even though iterators might return them in different orders).
- Identify cellViews equivalent except for object names or specific pieces of the hierarchy.
- Show two nets the same but with different routes.
- Two routes the same but different widths.

3.11.6 Verilog/Spice translators

To assist with OA proliferation, Verilog and Spice translation tools would help migration in some situations.

4. OA 2.3 Release Planned Features (June, 2004)

4.1. Enhance EMH

EMH as implemented in the OA 2.1 release has a restriction that all Module hierarchy embedded in a Block must have originated from within the limits of the Block. Expressed another way, if a hierarchy is viewed in a unified (occurrence) way, with some of the nodes being Modules, and some being both Modules and Blocks, then any partitioning cannot move a Module past the boundary of a combined Module/Block and still maintain the EMH form.

In IC implementation, it is often necessary to take a few cells that are associated with a timing critical path, promote them to the top level of the design, and then optimize them, in order to meet timing requirements. OA 2.1 doesn’t support this type of use model.

In OA 2.3, EMH restrictions will be reexamined in an attempt to discover a way to support the use model described above without causing undue performance, capacity, or robustness problems with the system. When considering so-called “full EMH”, where there is no direct relationship required between the Module and Block hierarchies, this is considered too complex to implement in a way that can take advantage of the OA 2.1 EMH restrictions when they are present, which they are in most cases. Rather than attempt to implement “full EMH”, a simpler relaxation of the restrictions that allow the use model described above will be sought.

4.2. Support manufacturing test

Tools like automatic test pattern generation (ATPG) programs will need to access necessary design data from OA to create required test patterns that can be stored back into the data base. The OA API and its Reference Implementation must be expanded to accommodate these requirements of manufacturing test. The extensions to OA must be broad enough to cover a range of test methods as listed below, with provisions for extensions to accommodate new methods in the future:

- DC stuck-at fault testing
- AC or at-speed testing
- Iddq testing
- Built-in Self Testing (BIST)
- Memory testing
- Analog and mixed signal testing for AMS and SoC designs
- Functional testing

The API must serve as the interface for storing/retrieving the design data specifically needed to support manufacturing test. These include structures identifiable to support:

- Design for test (DFT)
- Boundary scan, and
- BIST, including memory BIST (MBIST)

The API must handle special constraints to support manufacturing test. These include:

- Constraints to enable DFT, such as, clear identification of “test mode” I/O’s and their relationships to each other
- Constraints to enable boundary scan, similar to the above
- Constraints to enable BIST and MBIST, including, for example, initialization sequences for pseudo-random pattern generators and specific memory test sequences
- Special constraints on ATPG, such as, limits on driver switching to manage power consumption at the tester
- Specific manufacturing tester constraints

Last, the test patterns and all tester-related constraints must be stored in a form that can easily be translated into an acceptable/standard format for transfer to manufacturing.

4.3. IP security

An important requirement is to protect technology-specific or otherwise-proprietary OpenAccess data when sending it to external customers. Targets for such security include reusable core IP (see 3.10.6) and vendor-specific extension data. Encryption may be one alternative, however, the issue needs further analysis:

- ARM and other IP providers should be approached for their ideas/insights on the requirements and possible solutions.

- The level of interest versus cost to implement a solution must be evaluated.

4.4. Enhance technology / docs for OA-based SW integration

It is very possible to build an integrated, interoperable system that uses the OA database to indirectly communicate between different subsystems that are not aware of each other's presence. They do this by registering callbacks on object creation/deletion/modification that are used to keep the subsystem up-to-date as changes and edits are made to the data.

This can be done in a very efficient way, but the unfortunate truth is that if the system-level architects are not careful, it can be extremely *inefficient*. We need enhanced documentation and the technology to support it that details the best ways to make use of the database and the callbacks that are available. This document will explain the pitfalls that can occur, and will provide standards and techniques that, if followed, will result in an extremely efficient yet modular system.

4.5. Specify engine control-level communication APIs

An industry-standard data model is certainly a fundamental pre-requisite for enabling effective "plug and play" interoperability among tools, particularly in the incremental design closure arena. However, it is not completely sufficient. To achieve this ultimate objective, standards for controlling the interoperability among the tools must also be addressed. This could be as simple as tool initialization and termination or as complicated as enabling explicit control over incremental processing when the typical "lazy evaluation" approach isn't sufficient. In addition, access to tool specific information which may not be totally within the OA domain (e.g. timing/power/etc. results, placement and/or wiring density) and dynamic adjustments to tool processing controls (e.g. priority orders, constraints) may also be necessary. Of course, to make any of this work at all, there must also be standardization of the overall execution environment. That is, how are the tools selected for a specific design task started and dynamically configured at runtime? Addressing these issues is no small task, and should be started as soon as possible. (Related topics include: 3.10.1, 3.10.3, 3.10.7)

4.6. Support shared memory

From OA 2.1 onward, OpenAccess is fully thread-safe, and can be used in multi-threaded applications. A shared-memory model would enable support of a set of cooperating processes working on the same data simultaneously. In this model, separate processes could have access to the same cellViews, sharing the in-memory versions, so that changes within one process will immediately be visible to the other processes.

The proposed model will support a single writer and multiple readers for each cellView. The memory-mapped approach used in OA from its inception should facilitate the construction of a shared-memory system. Before introducing such a system, the new feature will be tested for all supported platforms. Note that this feature presumes that the set of cooperating processes is all on a single computer. There is no intent at this time to support network-based shared memory.

4.7. Support EM analysis

Further shrinking of sizes in deep-submicron designs leads to increase of currents' densities. Sustained high currents especially in power nets may cause atomic diffusion in conductors, known as electro-migration. Such mass transport can lead to open circuits in the form of voids and short circuits through so-called hillocks.

To perform EM analysis an application should be able to calculate current densities through wires and compare them with threshold values of an appropriate layer (mask). For EM visualization applications nowadays use a so called temperature map, which colors chip areas in accordance with the degree of EM threshold violation.

Consequently, the OA Information Model and API should provide the possibility to create and process the following information:

Technology:

- thickness of wires' layers (masks)
- EM threshold for each layer

Parasitic net:

- geometry of devices (resistors)
- layer number as device attribute
- current density through device
- reference to device original layout

4.8. Support library timing models

At this point, there is not a consistent approach for addressing the overall tool data access requirements, especially when it comes to library models. OA may deal effectively with the physical modeling for example, but falls short in other areas such as timing, perhaps even transistor-level physical modeling (see 5.1.). To make things a bit worse, these other areas currently have differing approaches concurrently in use. That is, there is the executable approach, as evidenced by OLA, and the more static approaches, as managed by file formats such as .lib, SPDM, and ALF. OA needs to determine how to best address these other areas. This could range from simple "co-existence" to actually extending the OA APIs into these other domains. The suggestion is that the OA Change Team should take a first pass at making this determination but then, as quickly as possible, should get with representatives of these other efforts (e.g. the OLA work group) to jointly work out the architecture to the satisfaction of all involved.

4.9. Support library power models / thermal analysis

This activity should be addressed in concert with the “Library timing models/support” (discussed in 4.8. because it is just another aspect of the general library modeling problem.

4.10. Create public-domain sample code (utilities, etc.)

Although the OpenAccess API ships with extensive documentation, for many the best way to learn is by example. Making available a set of sample programs, will help break down barriers to OA adoption.

These programs ought to vary in complexity as well as in API coverage, and perhaps even be available in additional languages such as Python. The list should include a small program showing the minimal way to call up the database, and also at least one large program that might create and then manipulate a large hierarchy of objects. It would also be beneficial to have programs demonstrating the proper use of a variety of different objects – particularly things that are new in OA that may not build on anyone's prior experience with other databases.

4.11. Define extension-language bindings (Tcl, Perl, Java)

The use of extension languages (or scripting languages) is popular for many sorts of analysis performed on IC design data. It is common for existing flows to depend on hundreds of thousands of lines of extension code. A requirement to convert all that to OA C++ is an impediment to adoption. To encourage customers to adopt OpenAccess, it is required that the OAC assure the availability of extension language bindings to the OA API.

The OAC has already made an RFT on this topic. Bindings that stay consistent with the C++ API and yet natural to their language will be favored. Only one standard binding per language will be allowed. However, any number of implementations of that mapping can compete with each other. One problem to solve will be how to evolve language bindings with new OA releases.

4.12. Develop more translators (others from RFT list)

Existing sites will not have data in OpenAccess format; they will have it in any number of other formats, many of them industry standards. To make OA easier to use, it is important to have translation programs that bring data into OA from standard formats, and vice-versa.

A "translator" is defined as the program doing translation from one format to or from OA, along with a set of terminology mappings and a set of recommended tests for verification purposes.

Each translator to be implemented has a priority, in part to reach the largest possible audience of potential OA adopters, and in part to favor the needs of the Coalition companies, who have invested the most in OA to this point.

Only one cross-reference specification per translation will be considered part of the Standard, although many implementations may be endorsed.

The translators in priority order are:

1. DEF => OA
2. Verilog => OA (see 3.11.6)
3. LEF => OA
4. SPICE => OA (see 3.11.6)
5. OA => SPICE (see 3.11.6)
6. OA => DEF
7. GDSII => OA
8. OA => LEF
9. OA => Verilog (see 3.11.6)
10. SPEF => OA
11. OA => GDSII
12. OA => SPEF

4.13. Enable application-controlled memory management

During previous projects (DR, CHDStd), tool developers made it clear to Si2 that they wanted control over DB memory allocation and freeing in order to optimize both capacity and performance. This is addressed in OLA, for example, with a hook for a "user-defined" allocator. The OLA implementation then always uses this hook (if defined) instead of its own allocator. In this way, the application can use optimized allocation algorithms, and track memory use better.

Even with a user-defined allocate hook, some DBs don't always "free" destroyed object storage, reusing it later under their own memory management algorithms. This can pose problems for a tool hungry for memory that has destroyed things it doesn't need anymore, but still can't get the memory back. One compromise is a hook to force the DB to free all destroyed object memory. Other hooks at finer levels of granularity could be defined to satisfy tool needs for control.

Since at this time an application has no control of the memory allocation of managed objects, these requirements from projects past should be evaluated to see if they still apply in the context of OA.

4.14. Provide UDM full support / interoperability

With shrinking feature size, the cost and elapsed times for mask making, yield ramp, and diagnosis/repair in manufacturing is beginning to gain attention as an area requiring change. One important part of the change deemed necessary for improvement is to provide a universal data model (UDM) that can transcend design, mask making and wafer fabrication processes.

OpenAccess is the obvious best candidate for this. To support the UDM, it will be necessary to extend the OpenAccess model in areas such as the following:

- Identification of non-design shapes used for reticle enhancement
- Identification of non-design shapes required for yield
- Mask order data (aka SEMI P10)
- Mask quality data
- Fault location mapping
- Information access control
- IP protection
- Data transfer performance

4.15. Support system-in-package

Given that OA will be extended to address the multi-package technology modeling needs (see 3.5.), OA should also address any package-specific design data requirements as well. These requirements may span support for simpler tasks such as “image/package co-design”, where the growing trend towards image customization is creating the need for performing this customization in the context of the package upon which the chip will be mounted, all the way to full “in situ” system design and analysis, where an entire system’s worth of design data covering multiple levels of traditional/advanced packaging descriptions will need to be addressed.

4.16. Provide name-mapping enhancements for simulators

Since its inception, OA has supported namespaces to effect name mapping from one namespace to another. This system uses an algorithmically-invertible mapping to convert from one namespace to another. There has not been an easy way to work with namespaces that are not name-based, but are numbered. Spice is an example of a (netlist) namespace that uses numbers to represent nets rather than names. Capabilities should be added to build cross-references that can be used by the name-mapping system to establish a correspondence between the OA internal namespace and the numbered names used in the number-based namespace. The cross-references will be associated with the cellView data, so they may be kept in synch with the OA data.

4.17. Redesign library access

Requests both for improved performance and for enhanced functionality are driving us to a significantly-different library architecture. There are opportunities to improve performance for reading many cells in a design hierarchy. Profound performance improvements can be achieved for updating properties on library containers, and we can add additional library structure for many different requests. This is particularly needed in the environment of EMH, where the standard 5.X library structure is inadequate for block cellViews containing module hierarchy, for occurrence models, for hierarchies with thousands of small cells, and for handling cellView variants. One implementation approach being considered for this task would use a client -server architecture for querying the library.

This topic is intimately tied to design data management issues. See 4.18.

4.17.1 Formalization of Uniquification

Uniquification is the process of taking many occurrences of a master cellView and turning them into a related set of copies that can be modified independently. Currently, any uniquification process is ad hoc, and is usually done through a convention (such as adding "_%d" as a suffix to the view name). Uniquification should be formalized so that the related cellViews would be kept together and there would be a way to create, delete, and open uniquified versions of a cellView master. The uniquified versions should be named, and there should be a different namespace for each different cellView that is uniquified. This will at least be implemented as needed by EMH. There is no need (yet!) for this to work with PCells. It must only work for regular cells.

4.17.2 Storing of PCell variants

It is desirable to be able to store on disk evaluated PCell variants. This will become more important in the future as more sophisticated (meaning longer run-time) features and commands become available for use in Pcell generation. There should be some way to "lazy evaluate" the PCells, so that if a PCell supermaster is changed, the submasters are only re-evaluated when it is necessary. There will undoubtedly be some limitations on this functionality based upon the read/write permissions on a library that should be either considered and addressed with some sort of "transparent filesystem" capability, or merely surfaced as known limitations.

4.18. Support design data management (DDM)

(The following is a general overview meant to characterize the issues involved. The DDM working group, currently under way, will determine the final requirements for this topic.)

DDM encompasses a broad spectrum of requirements for design groups which vary depending on the needs of the individual groups. General requirements include the ability to:

- Manage data throughout the design process;
- Manage all different types of data, including binary, text files, OA and non-OA data;
- Support a variety of versioning schemes such as branching, linear versioning, keeping x number of versions around;
- Scale from small team to large team;
- Customize to any methodology or use model;

- Support various levels of security;
- Quantify performance requirements resulting from the use of various index file mechanisms;
- Efficiently handle data distributed across multiple computers across geographic regions;
- Support different storage requirements depending on the types of data managed, which may result in the use of different access mechanisms like symbolic links, and so on.

Several DDM systems, commercial and home grown within a company's organization, have focused on different aspects of these requirements to satisfy their customer base. Since these requirements vary so greatly, OA takes a hands-off approach to data management and allows the DDM system to manage the data in the best way it sees fit. However, OA provides basic hooks to ensure the DDM can do its job. These basic hooks include pre and post triggers which are called when creating new objects, checking out objects for modification, and deleting objects. Objects in the OA world would remain as library, cell, view, and cellView, and DDM would be responsible for how its own objects are defined and used, such as version, configuration, etc.

Also, since DDM may need to manage both OA and non-OA data, managing the data via the file system directly is required. OA's mission is to provide access to the data via APIs, but OA takes a hands-off approach to managing the data. Therefore, OA will structure the data in the file system in such a way that the library, cell, view, and cellView objects are distinguishable without the need for an API.

5. Long-term (2-5 year) Roadmap

5.1. Support transistor-level modeling

Deep submicron technology, shrinking linear sizes, and decreasing voltages, aggravate such problems as IR drop, electro-migration, coupling noise, etc. All of these problems require more accurate simulation, which can be achieved only at a transistor level of design data representation.

Transistor-level simulation for active devices (transistor, diode, etc.) uses models which are represented as some program module with a set of predefined parameters, solving an appropriate system of physical equations. As a rule, modules are built as shared or dynamic link libraries. Model parameters are specified as text strings in ASCII format. This requires translations at every simulation start and may cause errors and lost of data.

Since transistor modeling is very sophisticated it is almost impossible to represent and process a model as a set of interrelated objects. To support transistor level modeling OpenAccess should expand its information model and API to cover such entities as (*note: all names are abstractions*):

- “model describer” (model name, module full name, version, list of parameter describers);
- “model instance” (name, reference to model describer, list of actual parameters);

- “parameter describer” (name, unit, default value, min value, max value);
- “parameter instance” (name, value by type, reference to parameter describer), which is very close to oaParameter.

The oaLibrary or oaTechnology object should be a container for “model describer” and “model instance”. The *oaCellView* object should have a reference to “model instance”. The *oaInst* object should have a list of “parameter instance” which has specific values for the given “model instance” referred by oaCellView.

These enhancements will enable access to model information directly from the database and therefore eliminate inconsistencies and errors. Moreover, it will allow building some kind of navigation over models.

One of the important issues concerning transistor level modeling is protection of Intellectual Property of models (values of parameters), since some companies do not want to disclose such information. Thus, OA should provide some mechanism of IP protection. (See 4.3.)

5.2. Support behavioral-level data models

The OA API through the 2.2 release is capable of representing data from a structural-RTL level (a netlist) through OASIS-level (mask-level) data. In this release, OA should provide positive mechanisms that can be used to support behavioral-level data as well. Since most behavioral designs include some structure, this will leverage the existing connectivity and hierarchy models to represent any structural components of a behavioral design. The behavioral components will be represented more directly.

There are several ways this could be designed. Keeping a pointer from within the cellView boundary of a behavioral cell to the text file representing the behavioral definition might be enough. It might be desirable to keep the original HDL form (which might aggregate several behavioral modules into a single file) and have multiple cellViews refer to the same larger file. It would also be possible to store the behavioral text directly inside the cellView, which would make sure a module could not easily get out of synch with the structural shell.

A more explicit alternative would model the HDL parse trees and/or the control-dataflow graph that represents the behavioral component. This method would pose a real challenge in picking a representation that is capable of storing both Verilog and VHDL. This must be a requirement, however.

5.3. Support architecture-level data models

The OpenAccess API and the reference data base should start looking at support for architecture level design. The goal would be for system level simulation, analysis, and design tools to access and exchange necessary data to perform system level simulation, and architectural exploration. SystemC, being promoted by the Open SystemC Initiative (OSCI) is emerging as a language standard to bridge the gap between system specification and RTL implementation. Levels of abstraction above RTL are being defined in this context to facilitate architectural level exploration, simulation and analysis tools. Significant abstraction levels above RTL include the transaction level, and untimed specification level, that enable the following:

- Early development of embedded software
- Design and exploration of system architectures, prior to implementation in RTL
- System level IP reuse

The task of standardizing system level modeling at the transaction level is currently underway. The goal of this activity would be to provide a platform definition of hardware at this high abstraction level that would permit software development. It would then be used as a starting point for hardware implementation and verification. The requirements being addressed include having IP at a high level of behavior and communication abstraction that exploits existing bus functionality. There should also be a clear path to implementation of this platform, enabling block and communication synthesis. In order to enable interoperability of tools at this high level of abstraction and across levels of abstraction, there is a need to explore potential extensions to the OA model and reference database to capture relevant aspects of system level design: transaction level models, APIs to interchange architectural analysis information, embedded software, and communication topology of the system.