**RC24063 (W0609-132) September 27, 2006**
**Computer Science**

# IBM Research Report

## Comprehensive Change Management for SoC Design

**Sunita Chulani, Stanley M. Sutton Jr., Gray Bachelor*, P. Santhanam**
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

*IBM Global Business Services
PO Box 31
Birmingham Road
Warwick CV34 5JL
United Kingdom

**Research Division**
**Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich**

**IP/SOC 2006**

# Comprehensive Change Management for SoC Design

Sunita Chulani[1], Stanley M. Sutton Jr.[1], Gary Bachelor[2], and P. Santhanam[1]

[1]IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532 USA

[2]IBM Global Business Services, PO BOX 31, Birmingham Road, Warwick CV34 5JL UK

**ABSTRACT** Systems-on-a-Chip (SoC) are becoming increasingly complex, leading to corresponding increases in the complexity of SoC design and development. SoC are composed of IP from multiple sources. These IP components and other elements of the SoC may be subject to change for various reasons and at varying rates. Uncontrolled changes can lead to widespread verification failures and performance problems, triggering redesign and reverification and greatly increasing SoC development times and costs.

We propose to address this problem by introducing comprehensive change management for SoC development. Change management, which is widely used in the software industry, involves controlling when and where changes can be introduced into a system under development and tracking the dependencies between components so that changes can be propagated quickly, completely, and correctly.

This recommendation is based on in-depth discussions with more than 30 experts in electronic design from across IBM. They identified over 20 significant change management problems in commercial chip development. A priority was to "bring everything under change control."

In this paper we address two main topics: One is typical scenarios in electronic design; these provide a framework for determining where change management can be supported and leveraged. The other the specification of a comprehensive schema to illustrate the range of data and relationships that are important for change management in SoC design.

## 1. INTRODUCTION

SoC designs are becoming increasingly complex. At the same time, pressures on design teams and project managers are rising because of shorter times to market, more complex technology issues, more complex organizations, and multi-partner and geographically dispersed teams with varied "business models" and higher "cost of failure."

Current methodology and tools for designing SoC need to evolve with market demands in key areas: First, multiple streams of inconsistent hardware (HW) and software (SW) processes are often integrated only in the late stages of a project, leading to unrecognized divergence of requirements, platforms, IP and so on, resulting in unacceptable risk in cost, schedule, and quality. Second, even within a stream of HW or SW, there is inadequate data integration, configuration management, and change control across life cycle artifacts. Techniques used for these are often ad hoc or manual, and the cost of failure is high. This makes it difficult for a distributed group of engineers to be productive and inhibits the early, controlled reuse of design products and IP. Finally, the costs of deploying and managing separate dedicated systems and infrastructures are becoming prohibitive.

We propose to address these shortcomings through comprehensive change management, which is the integrated application of configuration management, version control, and change control across software and hardware design. Change management is widely practiced in the software development industry. There are commercial change-management systems available for use in electronic design, such as MatrixOne DesignSync [4], ClioSoft SOS [2], IC Manage Design Management [3], and Rational ClearCase/ClearQuest [1], as well as numerous proprietary, "home-grown" systems. But to date change management remains an under-utilized technology in electronic design.

In SoC design, change management can help with many problems. For instance, when IP is modified, change management can help in identifying blocks in which the IP is used, in evaluating other affected design elements, and in determining which tests must be rerun and which rules must be re-verified. To take another example, when a new release is proposed, change management can help in assessing whether the elements of the release are mutually consistent and in specifying IP or other resources on which the new release depends for correct functioning.

More generally, change management provides the ability to analyze the potential impact of changes by tracing to affected entities and the ability to propagate changes completely, correctly, and efficiently. For design managers, this supports decision-making as to whether, when, and how to make or accept changes. For design engineers, it helps in assessing when a set of (new or modified) design entities is complete and consistent and in deciding when it is safe to make (or adopt) a new release.

In this paper we focus on two elements of this approach for SoC design. One is the specification of

representative use cases in which change management plays a critical role. These show places in the SoC development process where information important for managing change can be gathered. They also show places where appropriate information can be used to manage the impact of change. The second element is the specification of a generic schema for modeling design entities and their interrelationships. This supports traceability among design elements, allows designers to analyze the impact of changes, and facilitates the efficient and comprehensive propagation of changes to affected elements.

The following section provides some background on a survey of subject-matter experts that we performed to refine the problem definition. Section 3 then presents high level use cases that in combination address the major activities in typical SoC design processes. Section 4 gives an overview of the change-management schema, describing key elements and illustrating a small example of the application of the schema to electronic-design data. Section 5 looks at a particular use case, *Implement Change*, in more detail and shows where the use case can make use of various elements from the schema. Finally, we present our conclusions.

## 2. BACKGROUND

At the outset of this investigation we conducted a survey of some 30 IBM subject-matter experts (SMEs) in areas of electronic design, configuration and change management, and design data modeling. Our SMEs identified 26 problem areas relating to change management in electronic design. These could be categorized as follows:

- visibility into project status
- day-to-day control of project activities
- organizational or structural changes
- design method consistency
- design data consistency

Major themes that crosscut these categories included:

- visibility and status of data
- comprehensive change management
- method definition, tracking, and enforcement
- design physical quality
- a common approach to problem identification and handling

We held a workshop with the SMEs to prioritize these problems, and two emerged as the most significant: First, the need for *basic management of the configuration of all the design data elements and resources of concern* within a project or work package (libraries, designs, code, tools, test suites, etc.); second, the need for *designer visibility into the status of data and configurations* in the context of the work package.

To realize these goals, at least two basic kinds of information are necessary. One is an understanding of how change management may occur in SoC design processes. The other is an understanding of the kinds of information and relationships needed to manage change in SoC design. We addressed the former by specifying change-management use cases; we addressed the latter by specifying a change-management schema. These are discussed in the following sections.

## 3. USE CASES

This section describes typical use cases in the SoC design process. Change is a pervasive concern in these use cases—they cause changes, respond to changes, or depend on data and other resources that are subject to change. Thus, change management is integral to the effective execution of each of these use cases. We identified nine representative use cases in the SoC design process, which are shown in Figure 1. (In the following text, terms in Helvetica font refer to items in the figures.)

In general there are four ways of initiating a project: New Project, Derive, Merge and Retarget. New Project is the use case in which a new project is created from the beginning. The Derive use case is initiated when a new business opportunity arises to base a new project on an existing design. The Merge use case in initiated when an Actor wants to merge configuration items during implementation of a new CM scheme or while co-working with teams/organizations outside of the current CM scheme. The Retarget use case in initiated when a project is restructured due to resource or other constraints. In all of these use cases it is important to institute proper change controls from the outset. New Project starts with a clean slate; the other scenarios require changes from (or to) existing projects.

Once the project is initiated, the next phase is to update the design. There are two use cases in the Update Design composite state. New Design Elements addresses the original creation of new design elements. These become new entries in the change-management system. The Implement Change use case entails the modification of an existing design element (such as fixing a bug). It is triggered in response to a change request and is supported and governed by change-management data and protocols.

The next phase is the Resolve Project and consists of 3 use cases. Backout is the use case by which changes that were made in the previous phase can be reversed. Release is the use case by which a project is released for cross functional use. The Archive use case protects design asset by secure copy of design and environment

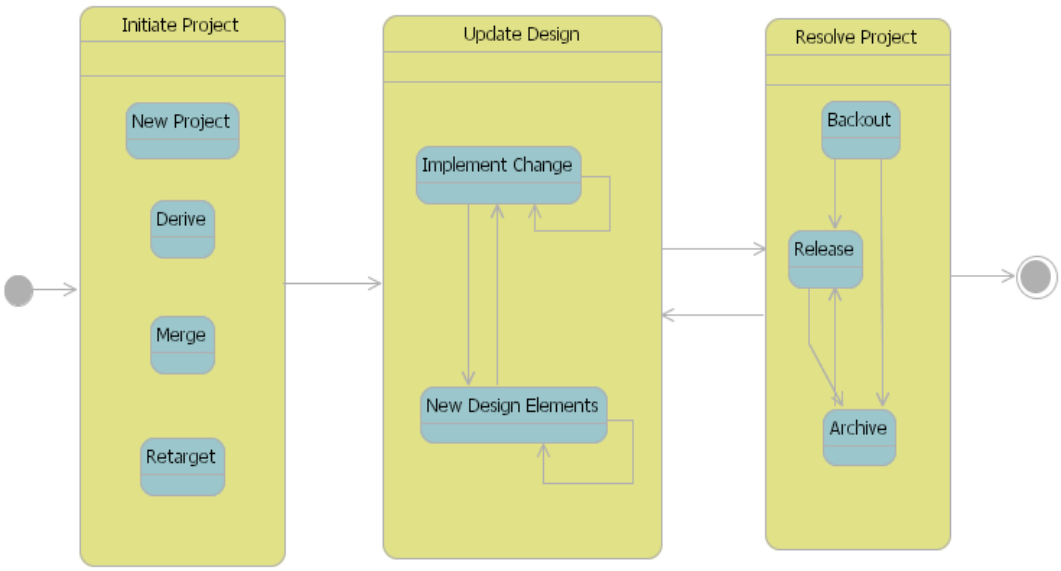


**Figure 1. Use cases in SoC design**

## 4. CHANGE-MANAGEMENT SCHEMA

The main goal of the change-management schema is to enable the capture of *all* information that might contribute to change management in electronic design.

### 4.1 Overview

The schema, which is defined in the Unified Modeling Language (UML) [5], consists of several high-level packages, as shown in Figure 2.[1]

The package Data represents various types of data elements for the representation of design data and metadata. Package Objects and Data defines types for objects and data. Objects are containers for information, data represent the information. The main types of object include artifacts (such as files), features, and attributes. The types of objects and data defined in this package are important for change management because they represent the principle work products of electronic design: IP, VHDL and RTL specifications, floor plans, formal verification rules, timing rules, and so on. It is changes to these things for which management is most needed.

The package Types defines types to represent the types of objects and data. This enables some types in the schema (such as those for attributes, collections, and relationships) to be defined parametrically in terms of other types, which promotes generality, consistency, and reusability of schema elements.

[1] In UML, a package groups related modeling elements, which can be considered to form a sub-model or sub-schema. We have used packages to group model elements relating to particular areas of concern in change management for SoC.

The package Attributes defines more specific types of attribute. The basic notion of an attribute is just that of a name-value pair that is associated to an object. (More strongly-typed subtypes of attribute have fixed names, value types, or attributed-object types, or combinations of these.) Attributes are one of the main types of design data, and they are also important for change management because they can be used to represent the status or state of design elements (such as version number, verification level, timing and noise characteristics, and so on).

Package Collections defines types of collections. These include collections with varying degrees of structure, member typing, and constraints. Collections are important for change management in that changes must often be coordinated for collections of design elements as a group (for example, for a work package, verification suite, or IP release). Collections are also used as the basis for several other elements in the schema (for example, baselines and change sets).

The package Relationships defines types of relationships. The basic relationship type is an ordered collection of a fixed number of elements. Subtypes provide directionality, element typing, and extrinsic and intrinsic semantics. Relationships are especially important for change management because they can define various types of dependencies among design data and between design data and resources. Some examples would include the use of macros in cores, the dependence of timing reports on floor plans and timing contracts, and the dependence of test results on tested designs, test cases, and test tools. Explicitly modeled dependency relationships support the analysis of change impact and the efficient and precise propagation of changes.

The package Specifications defines types of data specification and definition. Specifications specify an

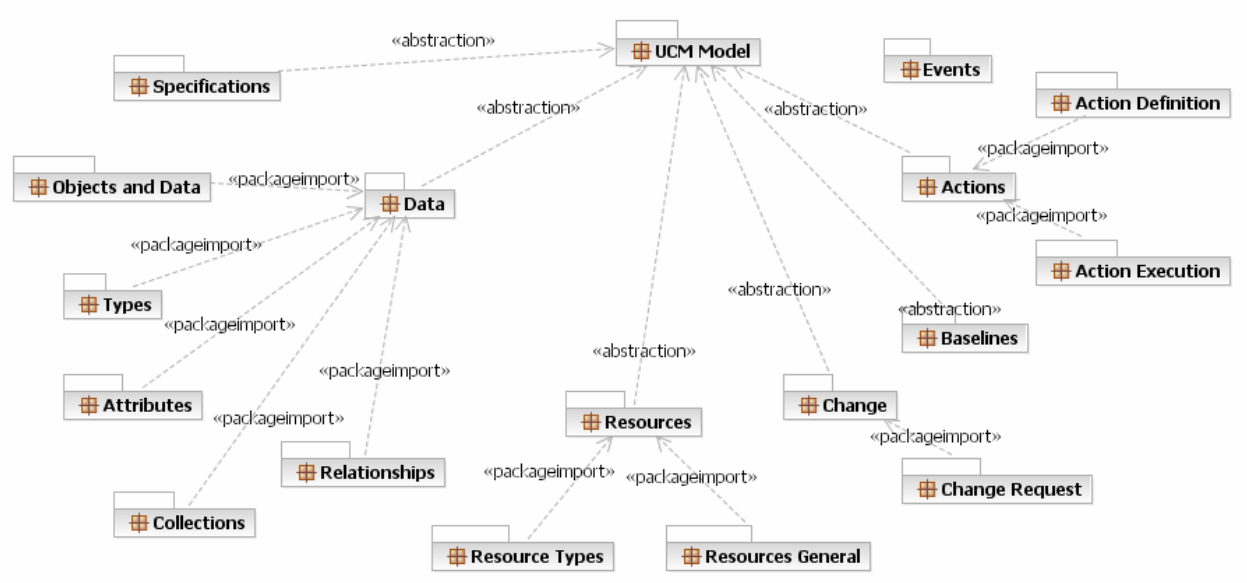


**Figure 2. Packages in the change-management schema**

informational entity; definitions are data that denote a meaning and are used in specifications

The package Resources represents things (other than design data) that are used in design processes, for example, design tools, simulators, IP, design methods, design engineers and project managers, and so on. Resources are important for change management in that resources are used in the actions that cause change and resources are used in the actions that respond to changes. Indeed, minimizing the resources needed to handle changes is one of the goals of change management. Resources are also important in that changes to a resource may require changes to design elements that were created using that resource (for example, when changes to a simulator may require reproduction of simulation results).

The package Events defines types and instances of events, including composite events. Events are important in change management because changes are a kind of event, and signals of change events can trigger processes to handle the change.

The package Actions provides a representation for things that are done, that is, for the behaviors or executions of tools, scripts, tasks, method steps, etc. Actions are important for change in that actions cause change. Actions can also be triggered in response to changes and can handle changes (such as by propagating changes to dependent artifacts).

The subpackage Action Definitions defines the type Action Execution, which contains information about a particular execution of a particular action (like an execution-log record). It refers to the definition of the action and to the specific artifacts and attributes read and written, resources used, and events generated and handled. Thus an action execution indicates particular artifacts and attributes that are changed, and it links those to the particular process or activity by which they were changed, the particular artifacts and attributes on which the changes were based, and the particular resources by which the changes were effected. Through this information particular dependency relationships can be established between the objects, data, and resources. This is the specific information needed to analyze and propagate concrete changes to artifacts, processes, resources, and so on.

Package Baselines defines types relating to baselines: mutually consistent set of design elements that can be used together. Baselines are important for change management in several respects. The elements in a baseline must be protected from arbitrary changes that might disrupt their mutual consistency, and the elements in a baseline must be changed in mutually consistent ways in order to evolve a baseline from one version to another.

The final package shown in Figure 2 is the Change package. This package defines several types that are important for representing change explicitly. These include managed objects, which are objects with an associated change log, change logs and change sets, which are two types of collection that contain change records, and change records, which record specific changes to specific objects. They can include a reference to an action execution for the action that caused the change.

The subpackage Change Requests includes several types for modeling requests for changes and the subsequent responses to those requests. A change request has a type, description, current state, priority, and owner. It can have an associated action definition, which may be the definition of the action to be taken in processing the change request. A change request also has a change-request history log.

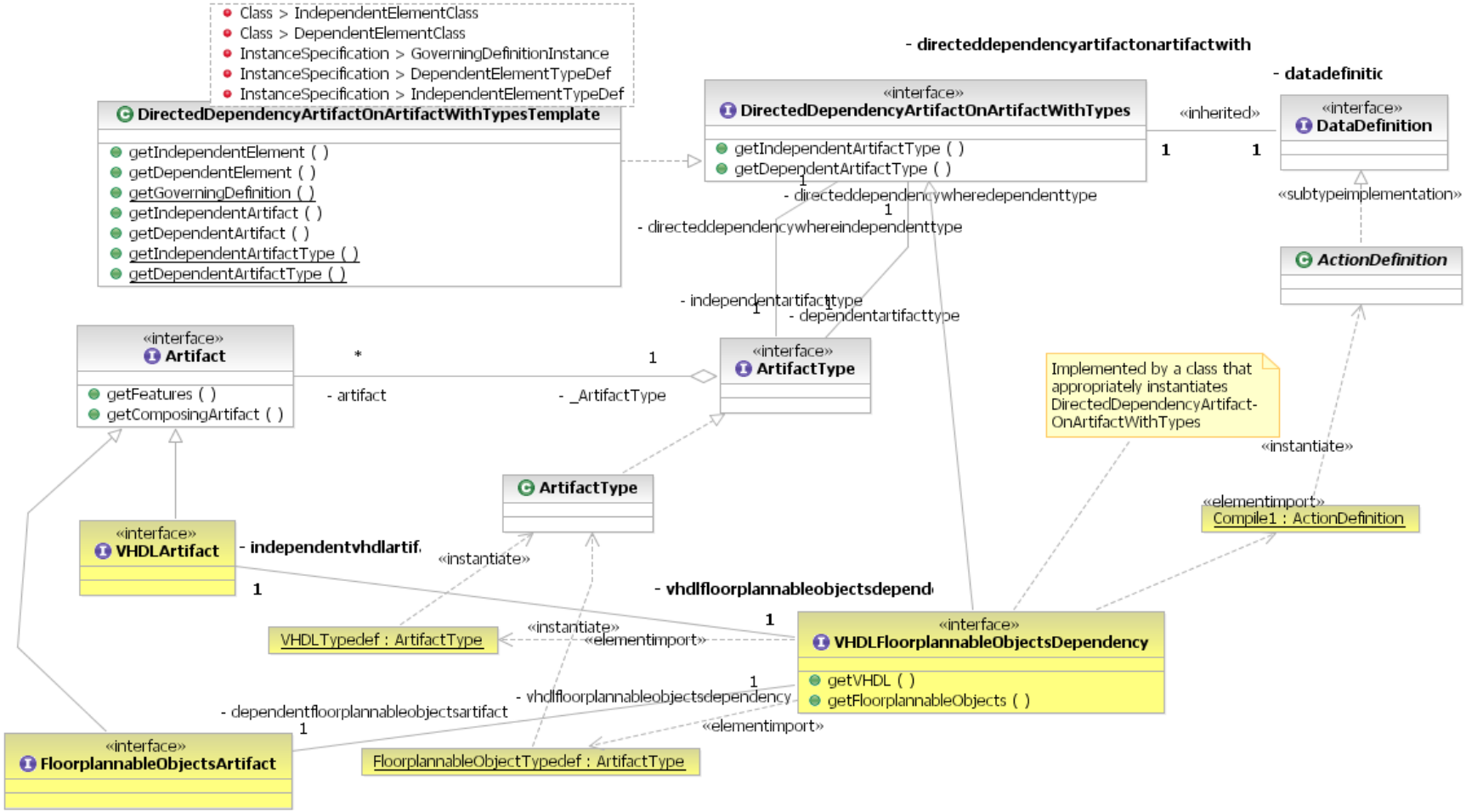


**Figure 3. Example of change-management data**

## 4.2 Example

An example of the use of the change-management schema is shown in Figure 3. Here, the clear boxes (upper part of diagram) show general types from the schema and the shaded boxes (lower part of the diagram) show types (and a few instances) specially defined to represent parts of a specific project at IBM (identifying details obscured). The example represents elements from a high-level design process. It shows a dependency relationship between two types of design artifact, VHDLArtifact and FloorPlannableObjects. The relationship is defined in terms of a compiler that derives instances of FloorPlannableObjects from instances of VHDLArtifact. Execution of the compiler constitutes an action that defines the relationship. The specific schemaelementsfor this example are defined based on the general schema using a variety of object-oriented modeling techniques, including specialization or subtyping (e.g., VHDLArtifact), parameterization (e.g. VHDLFloorplannableObjectsDependency), and instantiation (e.g., Compile1). Some additional explanation of the UML notation is found in the appendix.

# 5. USE CASE IMPLEMENT CHANGE

Here we present an example use case, Implement Change, with details on its activities and how the activities use the schema presented in Section 4. This use case is illustrated in Figure 4.

## 5.1 Use Case Details

The Implement Change use case addresses the modification of an existing design element (such as fixing a bug). It is triggered by a change request. The first steps of this use case are to identify and evaluate the change request to be handled. Then the relevant baseline is located, loaded into the engineer's workspace, and verified. At this point the change can be implemented. This begins with the identification of the artifacts that are immediately affected. Then dependent artifacts must be identified and changes propagated to them according to dependency relationships. (This may entail several iterations of change propagation.) Once a stable state is achieved, the modified artifacts are verified and regression tested. Depending on test results, more changes may be required. Once the change is considered acceptable, any learning and metrics from the process are captured and the new artifacts and relationships are promoted to the public configuration space.

## 5.2 Combination of Use Cases and Schema

The SoC design use cases and the SoC change-management schema can be used independently but they are designed to be used together. When used together, the activities in the use cases can be refined to reference specific data types defined in the schema. To give an idea of the correlation between the use cases and schema, and Table 1 shows the schema types that are used by various activities in the Implement Change scenario.

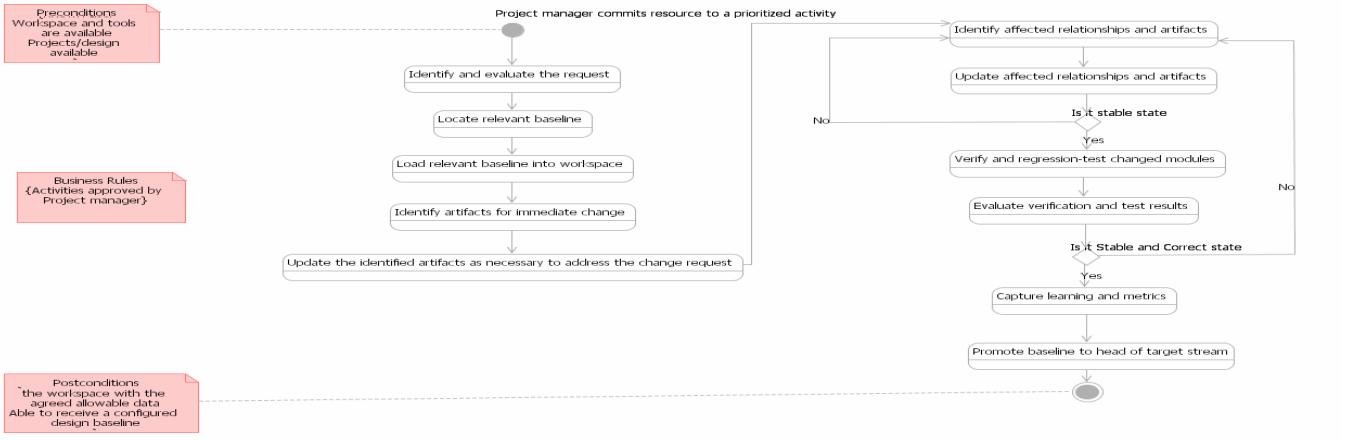


**Figure 4. State diagram for use case Implement Change**

| Implement- Change Activity | Object Types Referenced | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Arti-fact | Base-line | Base-line Cata-log | Base-line Set | Change Log | Change Record | Change Request | Change Set | Obj-ect Attri-bute | Rel-ation-ship | Resource Instance |
| Identify and evaluate change request | | | | | | | ■ | | | | |
| Locate relevant baseline | | ■ | ■ | | | | ■ | | | | |
| Load relevant baseline into workspace | | | | ■ | | | ■ | | | | |
| Identify artifacts for immediate change | ■ | | | | ■ | | ■ | | | | |
| Update identified artifacts as necessary | ■ | | | | ■ | ■ | ■ | | | | ■ |
| Identify affected relation-ships and artifacts | ■ | | | | | | | | | ■ | ■ |
| Updated affected relation-ships and artifacts | ■ | | | | | ■ | | ■ | | ■ | ■ |
| Verify and regression test changed modules | ■ | | | | | | | | ■ | | ■ |
| Evaluate verification and test results | ■ | | | | | | | | ■ | ■ | |
| Capture learning and metrics | ■ | | | | | | | | | | |
| Promote baseline to head of target stream | | ■ | | | | | ■ | | | | |

Table 1. Correlation of use-case activities and change-management schema elements for use case Implement Change (shaded elements indicate that the activity in the indicated row uses the data element in the indicated column).

## 6. CONCLUSIONS

This paper explores the role of comprehensive change management in the design, development, and delivery of SoC. Based on the comments of over thirty experienced electronic design engineers from across IBM, we have captured the essential problems and motivations for change management in SoC projects. We have described design scenarios, highlighting places where change management applies, and presented a preliminary schema to show the range of data and relationships change management may incorporate. Change management can benefit both design managers and design engineers. It is increasingly essential for improving productivity and reducing time and cost in SoC design.

## ACKNOWLEDGMENTS

Contributions to this work were also made by Nadav Golbandi and Yoav Rubin of IBM's Haifa Research Lab. Much information and guidance were provided by Jeff Staten and Bernd-josef Huettl of IBM's Systems and Technology Group. . We especially thank Richard Bell, John Coiner, Mark Firstenberg, Andrew Mirsky, Gary Nusbaum, and Harry Reindel of IBM's Systems and Technology Group for sharing design data and experiences with us. We are also grateful to the many other people across IBM who contributed their time and expertise.

## REFERENCES

1. http://www-306.ibm.com/software/awdtools/changemgmt/enterprise/index.html
2. http://www.cliosoft.com/products/index.html
3. http://www.icmanage.com/products/index.html
4. http://www.ins.clrc.ac.uk/europractice/software/matrixone.html#matrixone
5. http://www.uml.org/

## APPENDIX

Here we give a very brief additional explanation of the UML notation (especially as used in Figure 2).

The boxes mainly represent data types. Types are typically represented as either interfaces (which may have multiple implementations) or classes (which embody specific implementations). Operations or fields on the types are shown by elements within the boxes. Types can be parameterized, in which case they are shown with parametric elements in a superimposed box to the upper right.

Three of the boxes represent instances of types shown, namely Compiler1, FloorplannableObjectTypeDef, VHDLTypeDef. These are specific data elements that are used in defining other data elements in the example.

The lines between the boxes represent various kinds of association between the types:

An arrow with a closed head indicates that the type at the tail is a subtype of the type at the head (e.g., VHDLArtifact is a subtype of Artifact).

Solid lines without heads represent bi-directional associations between elements, e.g., a VHDLFloor-PlannableObjectDependency is associated with both a VHDLArtifact and a FloorplannableObjectsArtifact, and instances of VHDLArtifact and FloorplannableObjects-Artifact may be associated conversely with a VHDLFloorplannableObjectsDependency. Solid lines with open narrow heads indicate a directed association.

Arrows with open heads indicate directed associations.

Lines with a diamond shape at the head indicate that the element at the head has a collection or aggregation of elements at the tail (such as an ArtifactType constituting a collection of Artifacts).

Arrows shown with dotted lines represent various kinds of dependency, the nature of which is indicated by stereotype labels "<<…>>". Thus, Compile1 is an instantiation of the type ActionDefinition, the type ActionDefinition (a class) is a subtype implementation of the type DataDefinition (an interface), and the type VHDLFloorplannableObjectsDependency imports the element VHDLTypeDef (among others).