UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

TECHNISCHE UNIVERSITÄT DARMSTADT
FACHBEREICH 18 - ELEKTROTECHNIK UND
INFORMATIONSTECHNIK

A Framework Supporting Collaboration on the Distributed Design of Integrated Systems

by

LEANDRO SOARES INDRUSIAK

Tese submetida à avaliação,
como requisito parcial para a obtenção do grau de
Doutor em Ciência da Computação

A thesis submitted to evaluation
in partial fulfillment of the requirements for the Degree of
Doctor of Engineering

Prof. Dr. Ricardo Augusto da Luz Reis
Orientador

Prof. Dr. Dr. h. c. mult. Manfred Glesner
Advisor

July, 2003.

CIP - CATALOGAÇÃO NA PUBLICAÇÃO

Indrusiak, Leandro Soares

A Framework Supporting Collaboration on the Distributed Design of Integrated Systems / por Leandro Soares Indrusiak.- Porto Alegre: PPGC da UFRGS, 2003.

180f.: il.

Tese (doutorado) - Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2003. Orientador: Reis, Ricardo A.L.

1. Microeletrônica. 2. CAD. 3. Ambientes Distribuídos. 4. Apoio ao Projeto de Circuitos Integrados. 5. Trabalho Colaborativo Suportado por Computador. 6. Java. I. Reis, Ricardo A. L. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Panizzi
Pró-Reitora Adjunta de Pós-Graduação: Jocélia Grazia
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Haro

Acknowledgments

I would like to gratefully acknowledge the support of Prof. Dr. Ricardo Reis and Prof. Dr. Dr. h. c. mult. Manfred Glesner during this work. Both served as thesis advisors, being friendly, patient and always ready to share their knowledge on every matter. Prof. Reis was also the advisor during my studies towards the Master degree and was really influential on my decision to work as a researcher. His dedication and persistence always gave me encouragement to reach for higher goals. Prof. Glesner received me in his institute for a temporary stay, but soon invited me to join his staff and trusted me to take a position of great responsibility. Always providing guidance when I most needed, he helped me to bring my research potential to another level. Working with both of them was a truly rewarding experience.

The possibility of having two thesis advisors and a binational doctoral degree was granted by a special agreement between the Universidade Federal do Rio Grande do Sul (UFRGS) and the Technische Universität Darmstadt (TUD). I thank Profa. Wrana Panizzi, rector of UFRGS, Prof. Dr.-Ing. Johann-Dietrich Wörner, president of TUD, Prof. Dr. Carlos Alberto Heuser, coordinator of the Programa de Pós-Graduação em Computação of UFRGS, Prof. Dr.-Ing. Ralf Steinmetz, dean of the Fachbereich Elektrotechnik und Informationstechnik of TUD, and Prof. Dr.-Ing. Hans Eveking, as well as their respective staff, for their support on preparing this agreement.

The members of my thesis examination committee, Prof. Dr. Flávio Rech Wagner, Prof. Dr. Ricardo Jacobi, Prof. Dr.-Ing. Rolf Jakoby and Prof. Dr.-Ing Abdelhak Zoubir, are thanked for the stimulating discussions during the research phase and for their valuable feedback on the thesis text and oral presentation.

I would like to thank my professors and colleagues for the excellent working environment I had during the last five years. From the Microelectronic Systems Institute in Darmstadt, I'd like to thank all post-doc and doctoral researchers, secretaries and technicians, as well as the Diplom and Master students who supported me on the implementation work related to the thesis. Special thanks to Prof. Dr.-Ing. Juergen Becker (who provided me and my family with essential support on our arrival to Darmstadt), Dr.-Ing. Alberto Garcia Ortiz, Dipl.-Ing. Tudor Murgan, Dipl.-Ing. Ralf Ludewig, Dr.-Ing. Ulrich Mayer, Dr.-Ing. Thomas Hollstein, M.Sc. Abdulfattah Obeid, Dr.-Ing. Peter Zipf, Dipl.-Ing. Clemens Schlachta, M.Sc. Juan Ocampo, Dipl.-Ing. Octavian Mitrea and Dipl.-Ing. Florian Lubitz. I'd also like to thank all the members of the UFRGS Microelectronics Group, specially M.Sc. Sandro Sawicki, M.Sc. Lisane Brisolara, M.Sc. Márcio Kreutz, Prof. Dr. Luigi Carro, Prof. Dr. Marcelo Johann, Prof. Dr. José Luís Güntzel, M.Sc. José Carlos Sant'Anna Palma, Prof. Dr. João Baptista Martins, Profa. Dra. Fernanda Lima Kastensmidt, M.Sc. Ana Cristina Pinto, Prof. Dr. Sergio Bampi, Prof. Dr. Altamiro Susin and B.Sc. Émerson Hernandez. The staff of the network administration and library of Informatics Institute at UFRGS are also thanked for being always friendly and attentive. For their support on my language proficiency examination, I thank B.Sc. Jana Kaiser, Prof. Werner

Heidermann and Prof. Éda Heloisa Pilla. I would also like to thank my former colleagues and students at PUCRS in Uruguaiana, for their support on the early stage of my doctoral work, specially Prof. Cleiton Tambellini Borges, Prof. Marcus Kindel, Prof. Mauro Sopeña, M.Sc. Luciano Copello Ost and Clara Valim dos Santos.

During the years I worked on this thesis, I lived in four different cities, two different countries, many temporary and permanent addresses, but my friends were always able to find me and support me. I will have to thank all of them by mentioning only a few: people from the Clã (Doro, Doneide, Wilker, Sandro, Lorene, Carlos, Naban, Lulu, Cabeça…), from Dirty Job (Álisson, Bolli, Paulo, Drew…), from the net (Erlon, Guilherme Kujawski, Claudio, Fabio, Raquel e Daniel, Andrea Itano, Oggh, Bart…), from Santa Maria (Vera, Caporal, Daiane, Gibsy, Cris, Ana Paula, Freddy, Sonia, Juliano, Nadine, Marcelo Mussoi and family, Marshal, Dani Walty, Norberto Staggemeier…) and from Darmstadt (Ilina Murgan, Peter and Lucrécia Edinger, Nicole Marx…).

Finally, and most importantly, I would like to thank my family. My parents, Leon and Dalila, who gave me everything, whose example I will always follow and whom I will never be able to thank enough. My brothers Daniel and Lucas, for being my friends from the moment they've been born. My wife Giuliana, for being by my side with love and support ever since we first met. My children, Ana Laura and Luís Guilherme, for teaching me things I would never learn without them. I am grateful to my aunts and uncles, specially Lygia (in memoriam), Leocádia and Leoniza, for showing me the beauties of the world, the sciences and arts since my early childhood. I'd also like to thank my uncle Leonardo, his wife Maria Luiza and his daughters Antônia, Malvina and Francisca, for their hospitality and for bringing good vibes during my thesis defense; and my aunt Ana and her husband Gilberto, for their continuous support. Giuliana's parents, Julio and Maria Ana, as well as her sisters Márcia and Virgínia, her brother Luis Cláudio and her Peruvian grandmother Antolina are thanked for being the best in-laws one could ask for. And I am extremely grateful for the support I had from my grandfather Boleslau until his last days, enlightening me with his wise and inspired words, as well as the encouragement I have from my grandmother Malvina, giving me the best possible example of dedication, strength and at the same time tenderness.

Leandro Soares Indrusiak

Stockstadt am Rhein, 2003.

Table of Contents

List of Abbreviations 8

List of Figures 9

List of Tables 11

Abstract 12

Resumo 14

Kurzfassung 17

1 Introduction 20

1.1 Motivation 20
1.2 Thesis Organization 22

2 Design Automation Frameworks 24

2.1 Introduction 24
2.2 Integrated Systems Design 24
2.2.1 Functional Specification and Validation 26
2.2.2 Partitioning 27
2.2.3 Software and Hardware Specification, Simulation and Implementation 28
2.2.4 Hardware Synthesis 29
2.3 Design Automation Tools 30
2.4 CAD Frameworks: the Classical Concept 30
2.4.1 Operating System Services 32
2.4.2 Process Management Services 32
2.4.3 Data Representation and Management 33
2.4.4 Design and Methodology Management Services 34
2.4.5 Tool Integration and Encapsulation Services 35
2.4.6 Data Versioning Services 36
2.4.7 User Interface Services 36

3 Previous Work 38

3.1 Introduction 38
3.2 NELSIS 38
3.3 Version Server 40
3.4 STAR 41
3.5 Ulysses and Odyssey 42
3.6 WELD 42
3.7 OmniFlow 44
3.8 ASTAI(R) 46
3.9 Moscito 47
3.10 PPP 48
3.11 JavaCAD 49
3.12 Ptolemy II 50
3.13 Cave 51

3.14 Comparison of reviewed approaches 53

4 Cave2 Foundations 55

4.1 Introduction 55
4.2 Object Orientation 55
4.2.1 Object Oriented Frameworks 57
4.2.2 Design Patterns 58
4.3 Architectural Evolution - from hyperdocuments to OO 61
4.4 Cave2 Architecture 62
4.5 Java-based Approach 66

5 Framework Core 68

5.1 Introduction 68
5.2 Design tool primitives 69
5.3 Design data primitives 73

6 Supporting Distributed Design 79

6.1 Introduction 79
6.2 Resource Distribution Architecture 81
6.3 Service Space 85
6.3.1 Repository Service 86
6.3.2 Collaboration Service 98
6.3.3 Authentication Service 99
6.3.4 Prototyping Service 99
6.3.5 Additional Services 102

7 Supporting Collaborative Design 103

7.1 Introduction 103
7.2 Design visualization issues 104
7.3 Concurrency control issues 107
7.4 Versioning Support 115
7.5 Metadata Support 118

8 Case Studies 119

8.1 Introduction 119
8.2 Prototyping Service 119
8.3 IBlaDe 121
8.3.1 Interface-based Design 122
8.3.2 Supporting Interface-based Design 124
8.3.3 Implementation Issues 129
8.4 Educational Metadata 131

9 Conclusions and Future Work 135

9.1 Conclusions 135
9.1.1 CAD Frameworks 135
9.1.2 Design Databases 137
9.1.3 Collaborative Design 138
9.1.4 Summary 140
9.2 Future Work 141

Appendix 1 Cave Development Timeline 143

Appendix 2 Cave UML Class Diagrams 147

Appendix 3 Cave2 Code Statistics 150

Appendix 4 Cave2 Code Documentation 151

Appendix 5 Summary in Portuguese Language 152

References 165

List of Abbreviations

API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CAD	Computer-Aided Design
CIF	Caltech Intermediate Format
CSCW	Computer Supported Collaborative Work
EDA	Electronic Design Automation
GUI	Graphical User Interface
HCI	Human-Computer Interaction
HDL	Hardware Description Language
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IBD	Interface-based Design
IP	Intelectual Property
JNI	Java Native Interface
JFC	Java Foundation Classes
JVM	Java Virtual Machine
MVC	Model-View-Controller
NCSS	Non-Commented Source Statements
OS	Operating System
PC	Personal Computer
RF	Radio Frequency
RHS	Reconfigurable Hardware Service
RMI	Remote Method Invocation
RTOS	Real-Time Operating System
SOC	System-on-Chip
TCP/IP	Transfer Control Protocol / Internet Protocol
URL	Universal Resource Locator
VHDL	VHSIC Hardware Description Language
VLSI	Very Large Scale Integration
VRML	Virtual Reality Modelling Language
WWW	World Wide Web
XML	Extensible Markup Language

List of Figures

FIGURE 2.1 – Technologies integrated on SoC in the standard CMOS process...... 25
FIGURE 2.2 – Simplified System Design Flow 26
FIGURA 2.3 – CAD Framework architecture 31
FIGURE 3.1 – NELSIS Design Flow Browser 39
FIGURE 3.2 – A flow-map example 39
FIGURE 3.3 – An hierarchical flow-map example 40
FIGURE 3.4 – Versioning in the STAR Framework 41
FIGURE 3.5 – WELD Architecture.. 43
FIGURE 3.6 – OmniFlow Graphical User Interface 45
FIGURE 3.7 – OmniFlow Task Instance Architecture... 45
FIGURE 3.8 – ASTAI(R) Workflow Editor... 46
FIGURE 3.8 – MOSCITO Software Architecture.. 47
FIGURE 3.9 – Client-server architecture on PPP... 48
FIGURE 3.10 – Platform independent IP simulation 50
FIGURE 3.11 – Ptolemy II Kernel class diagram (partial) 51
FIGURE 3.12 – Information Flow on Cave System... 53
FIGURE 4.1 – Object-Oriented Frameworks 58
FIGURE 4.2 – Observers and Subject 59
FIGURE 4.3 – UML representation of the Observer design pattern structure 60
FIGURE 4.4 – Proposed architecture for resource distribution................................ 64
FIGURE 4.5 – UML use-case diagram modeling interaction between user and design environment 66
FIGURE 5.1 – UML Class diagram of GUI primitives (partial).............................. 72
FIGURE 5.2 – UML Class diagram of visual primitives for design representation (partial)... 73
FIGURE 5.3 – Example of hierarchical construct... 74
FIGURE 5.4 – 5-box Data Representation Model.. 75
FIGURE 5.5 – Example of inheritance construct.. 76
FIGURE 5.6 – UML Class diagram of the proposed design data model 77
FIGURE 6.1 – Evolution of distributed systems 80
FIGURE 6.2 – Example on task distribution.. 80
FIGURE 6.3 – Middleware architecture... 82
FIGURE 6.4 – Resource lookup protocol... 83
FIGURE 6.5 – Service space architecture 86
FIGURE 6.6 – Alternatives on Design Data Repository 90
FIGURE 6.7 – Repository ServiceUML Sequence Diagram 92
FIGURE 6.8 – Overview of the Repository Service implementation 95
FIGURE 6.9 – Design data identification keys 97
FIGURE 6.10 – UML sequence diagram for the prototyping service..................... 101
FIGURE 6.11 – Abstraction layers between object domain and hardware domain 101
FIGURE 7.1 – Implementation alternatives for the visualization of design data.... 106
FIGURE 7.2 – Collaboration Service Overview 111
FIGURE 7.3 – Collaborative Service UML Sequence Diagram 114
FIGURE 7.4 – Embedding versioning information within identification keys....... 117
FIGURE 8.1 – Case study on reconfigurable computing on demand...................... 120
FIGURE 8.2 – Exemplifying Interface-based Design 123

FIGURE 8.3 – Communication transaction among system parts 124
FIGURE 8.4 – Modeling structural and functional semantics................................ 125
FIGURE 8.5 – Modeling structural hierarchy 127
FIGURE 8.6 – Interface definitions in hierarchical blocks 128
FIGURE 8.7 – IBlaDE GUI snapshot - structural view.. 130
FIGURE 8.8 – IBlaDE GUI snapshot - structural and functional view 130
FIGURE 8.9 – Course Editor GUI.. 134
FIGURE 8.10 – Case Study: Metadata as training material 134
FIGURE A2.1 – Cave2 Design Tool Primitives (partial)....................................... 147
FIGURE A2.2 – Cave2 Design Data Primitives (partial)....................................... 148
FIGURE A2.3 – Cave2 Repository and Collaboration Services (partial) 149

List of Tables

TABLE 1.1 – CSCW Time-space Taxonomy 21
TABLE 3.1 – Comparison between CAD systems supporting distributed, multi-user design of integrated systems 54
TABLE 6.1 – Comparison between CAD systems supporting abstraction of the CAD resource distribution 85
TABLE 8.1 – DES algorithm implementation comparison................................. 121
TABLE 8.2 – Occurrence of Transactions 128
TABLE 9.1 – Comparison between CAD systems supporting distributed, multi-user design of integrated systems 140

Abstract

The work described in this thesis aims to support the distributed design of integrated systems and considers specifically the need for collaborative interaction among designers. Particular emphasis was given to issues which were only marginally considered in previous approaches, such as the abstraction of the distribution of design automation resources over the network, the possibility of both synchronous and asynchronous interaction among designers and the support for extensible design data models.
Such issues demand a rather complex software infrastructure, as possible solutions must encompass a wide range of software modules: from user interfaces to middleware to databases. To build such structure, several engineering techniques were employed and some original solutions were devised. The core of the proposed solution is based in the joint application of two homonymic technologies: CAD Frameworks and object-oriented frameworks. The former concept was coined in the late 80's within the electronic design automation community and comprehends a layered software environment which aims to support CAD tool developers, CAD administrators/integrators and designers. The latter, developed during the last decade by the software engineering community, is a software architecture model to build extensible and reusable object-oriented software subsystems. In this work, we proposed to create an object-oriented framework which includes extensible sets of design data primitives and design tool building blocks. Such object-oriented framework is included within a CAD Framework, where it plays important roles on typical CAD Framework services such as design data representation and management, versioning, user interfaces, design management and tool integration.
The implemented CAD Framework - named Cave2 - followed the classical layered architecture presented by Barnes, Harrison, Newton and Spickelmier, but the possibilities granted by the use of the object-oriented framework foundations allowed a series of improvements which were not available in previous approaches:
- object-oriented frameworks are extensible by design, thus this should be also true regarding the implemented sets of design data primitives and design tool building blocks. This means that both the design representation model and the software modules dealing with it can be upgraded or adapted to a particular design methodology, and that such extensions and adaptations will still inherit the architectural and functional aspects implemented in the object-oriented framework foundation;
- the design semantics and the design visualization are both part of the object-oriented framework, but in clearly separated models. This allows for different visualization strategies for a given design data set, which gives collaborating parties the flexibility to choose individual visualization settings;
- the control of the consistency between semantics and visualization - a particularly important issue in a design environment with multiple views of a single design - is also included in the foundations of the object-oriented framework. Such mechanism is generic enough to be also used by further extensions of the design data model, as it is based on the inversion of control between view and semantics. The view receives the user input and propagates such event to the semantic model, which evaluates if a state change is possible. If positive, it triggers the change of state of both semantics and view. Our approach took advantage of such inversion of control and included an layer

between semantics and view to take into account the possibility of multi-view consistency;
- to optimize the consistency control mechanism between views and semantics, we propose an event-based approach that captures each discrete interaction of a designer with his/her respective design views. The information about each interaction is encapsulated inside an event object, which may be propagated to the design semantics - and thus to other possible views - according to the consistency policy which is being used. Furthermore, the use of event pools allows for a late synchronization between view and semantics in case of unavailability of a network connection between them;
- the use of proxy objects raised significantly the abstraction of the integration of design automation resources, as either remote or local tools and services are accessed through method calls in a local object. The connection to remote tools and services using a look-up protocol also abstracted completely the network location of such resources, allowing for resource addition and removal during runtime;
- the implemented CAD Framework is completely based on Java technology, so it relies on the Java Virtual Machine as the layer which grants the independence between the CAD Framework and the operating system.

All such improvements contributed to a higher abstraction on the distribution of design automation resources and also introduced a new paradigm for the remote interaction between designers. The resulting CAD Framework is able to support fine-grained collaboration based on events, so every single design update performed by a designer can be propagated to the rest of the design team regardless of their location in the distributed environment. This can increase the group awareness and allow a richer transfer of experiences among them, improving significantly the collaboration potential when compared to previously proposed file-based or record-based approaches.

Three different case studies were conducted to validate the proposed approach, each one focusing one a subset of the contributions of this thesis. The first one uses the proxy-based resource distribution architecture to implement a prototyping platform using reconfigurable hardware modules. The second one extends the foundations of the implemented object-oriented framework to support interface-based design. Such extensions - design representation primitives and tool blocks - are used to implement a design entry tool named IBlaDe, which allows the collaborative creation of functional and structural models of integrated systems. The third case study regards the possibility of integration of multimedia metadata to the design data model. Such possibility is explored in the frame of an online educational and training platform.

Keywords: Microeletronics, Computer-Aided Design, Distributed Sistems, Electronic Design Automation, Computer Supported Collaborative Work, Java.

TÍTULO: "UM FRAMEWORK DE APOIO À COLABORAÇÃO NO PROJETO DISTRIBUÍDO DE SISTEMAS INTEGRADOS"

Resumo

O trabalho de pesquisa apresentado nesta tese tem por objetivo apoiar o projeto distribuído de sistemas integrados, considerando especificamente a necessidade de interação colaborativa entre os projetistas. O trabalho enfatiza particularmente alguns problemas que foram considerados apenas marginalmente em abordagens anteriores, como a abstração da distribuição em rede dos recursos de automação de projeto, a possibilidade de interação síncrona e assíncrona entre projetistas e o suporte a modelos extensíveis de dados de projeto.
Tais problemas requerem uma infra-estrutura de software significativamente complexa, pois possíveis soluções envolvem diversos módulos, desde interfaces com o usuário até bancos de dados e middleware. Para construir tal infra-estrutura, várias técnicas de engenharia foram empregadas e algumas soluções originais foram desenvolvidas. A idéia central da solução proposta é baseada no emprego conjunto de duas tecnologias homônimas: CAD Frameworks (ambientes integrados de apoio ao projeto) e frameworks orientados a objeto. O primeiro conceito foi criado no final da década de 80 na área de automação de projeto de sistemas eletrônicos e define uma arquitetura de software em níveis, voltada ao apoio a desenvolvedores de ferramentas de projeto, administradores de ambientes de projeto e projetistas. O segundo, desenvolvido na última década na área de engenharia de software, é um modelo para arquiteturas de software visando o desenvolvimento de sub-sistemas reusáveis de software orientado a objeto. No presente trabalho, propõe-se a criação de um framework orientado a objetos que inclui conjuntos extensíveis de primitivas de dados de projeto bem como de blocos para a construção de ferramentas de CAD. Esse framework orientado a objeto é agregado a um CAD Framework, onde ele passa a desempenhar funções tipicamente encontradas em tal ambiente, tais como representação e administração de dados de projeto, versionamento, interface com usuário, administração de projeto e integração de ferramentas.
O CAD Framework implementado dentro do escopo desta tese foi chamado Cave2 e seguiu a clássica arquitetura em níveis apresentada por Barnes, Harrison, Newton e Spickelmier. Durante o projeto e a implementação do Cave2, uma série de avanços em relação as abordagens anteriores foi obtida com a exploração das vantagens advindas do uso de um framework orientado a objetos:
- frameworks orientados a objetos são extensíveis por definição, então o mesmo pode ser dito a respeito das implementações dos conjuntos de primitivas de dados de projeto bem como de blocos para a construção de ferramentas de CAD. Isso implica que tanto o modelo de representação de projeto quanto os módulos de software processando tal modelo podem ser atualizados ou adaptados para uma metodologia de projeto específica, e que essas atualizações e adaptações ainda herdarão os aspectos arquiteturais e funcionais implementados nos elementos básicos do framework orientado a objetos;
- ambos os aspectos relativos à semântica do projeto e à visualização do projeto são partes do framework orientado a objetos, mas em modelos claramente separados. Isso possibilita o uso de várias estratégias para a visualização de um conjunto de dados de

projeto, o que dá aos participantes de uma sessão de projeto colaborativo a flexibilidade de escolha individual de estratégia de visualização;
- o controle de consistência entre semântica e visualização - uma questão particularmente importante em um ambiente de projeto onde coexistem múltiplas visualizações de cada projeto - também está incluído nas fundações do framework orientado a objetos implementado. Esse mecanismo é genérico o bastante para ser usado também pelas possíveis extensões do modelo de dados de projeto, uma vez que ele é baseado na inversão de controle entre a visualização e a semântica. A visualização recebe a intenção do usuário e propaga esse evento ao modelo da semântica, o qual avalia a possibilidade de uma mudança de estado. Se positivo, ele dispara a mudança de estado em ambos os modelos de visualização e semântica. A abordagem proposta nesta tese usa tal inversão de controle para incluir um nível adicional de processamento entre a semântica e a visualização, visando o controle de consistência nos casos de múltiplas visualizações;
- para otimizar o mecanismo de controle de consistência entre semântica e visualização, uma abordagem baseada em eventos foi proposta, buscando discretizar cada interação entre o projetista e suas visualizações do projeto. A informação sobre cada uma das interações é encapsulada em um objeto-evento, que pode ser propagado para o modelo da semântica do projeto - e então para as demais possíveis visualizações - de acordo com a política de consistência que esteja sendo usada. Além disso, o uso de eventos permite que as interações do usuário com a visualização sejam acumuladas para uma posterior sincronização com a semântica do projeto, caso haja indisponibilidade de conexão entre elas;
- o uso de objetos de proxy aumentou significativamente o nível de abstração da integração de recursos de automação de projeto, pois tanto ferramentas e serviços remotos quanto os instalados localmente são acessados através de chamadas de métodos em um objeto local. A conexão aos serviços e ferramentas remotos é obtida através de um protocolo de look-up, abstraíndo completamente a localização de tais recursos na rede e permitindo a adição e remoção em tempo de execução;
- o CAD Framework foi implementato completamente usando a tecnologia Java, usando dessa forma a Java Virtual Machine como intermediário entre o sistema operacional e o CAD Framework, garantindo dessa forma a independência de plataforma.

Todas as contribuições listadas anteriormente contribuiram com o aumento do nível de abstração da distribuição de recursos de automação de projeto e também apresentaram um novo paradigma para a interação remota entre projetistas. O CAD Framework no qual tais contribuições foram aplicadas é capaz de suportar colaboração de granularidade fina baseada em eventos, onde cada atualização feita por um projetista pode ser propagada para o restante da equipe, mesmo que estejam todos geograficamente distribuídos. Isto pode aumentar a sinergia de grupo entre os projetistas e permitir uma troca mais rica de experiências entre eles, aumentando significativamente o potencial de colaboração quando comparado com abordages baseadas em acesso a arquivos e registros propostas anteriormente.

Três estudos de caso diferentes foram realizados para validar a abordagem proposta, cada um deles envolvendo um sub-conjunto das contribuições da presente tese. O primeiro utiliza a arquitetura de distribuição de recursos baseada em proxies para implementar uma plataforma de prototipação usando módulos de hardware reconfigurável. O segundo estende as fundações do framework orientado a objetos visando suportar projeto baseado em interfaces. Essas extensões - primitivas de

representação de projeto e partes de ferramentas - são usadas na implementação de uma ferramenta chamada IBlaDe, que permite a criação colaborativa de modelos funcionais e estruturais de sistemas integrados. O terceiro estudo de caso aborda a possibilidade de integração de metadados multimídia ao modelo de dados de projeto. Essa possibilidade é explorada no contexto de uma plataforma online de educação e treinamento.

Palavras-chave: Microeletrônica, CAD, Ambientes Distribuídos, Apoio ao Projeto de Circuitos Integrados, Trabalho Colaborativo Suportado por Computador, Java.

TITEL: "EIN FRAMEWORK ZUR UNTERSTÜTZUNG DER KOOPERATION BEIM VERTEILTER ENTWURF INTEGRIERTER SYSTEME"

Kurzfassung

Die vorliegenden Arbeit beschäftigt sich mit der Unterstützung des verteilten Entwurfs von integrierten Systemen und speziell mit den Bedarf an gemeinschaftlicher Interaktion zwischen den Designern. Ein besonderer Schwerpunkt wurde dabei auf Themen gelegt, die in früheren Lösungsansätzen nur am Rande betrachtet wurden, wie die Abstraktion der Verteilung der Entwurfswerkzeuge im Netzwerk, die Möglichkeit von sowohl synchroner als auch asynchroner Interaktion zwischen den Designern und die Unterstützung von erweiterbare Datenmodelle für die Entwürfe.
Diese Themen benötigen eine recht komplexe Softwareinfrastruktur, da mögliche Lösungen eine Vielzahl verschiedener Softwaremodule umfassen müssen: von der Benutzerschnittstelle über die Middleware bis zur Datenbank. Um eine solchen Infrastruktur aufzubauen wurden verschiedene Techniken aus den Ingenieurwissenschaften genutzt und einige neuartige Lösungen entwickelt. Der Kern der vorgeschlagenen Lösung basiert auf der Verschmelzung von zwei Techniken: CAD-Frameworks und objektorientierte Frameworks. Das erstgenannte Konzept wurde durch den rechnergestützten Schaltungsentwurf in den späten 80'er Jahren geprägt und beinhaltet ein in Schichten aufgebautes Softwaresystem, das dazu gedacht war, CAD-Softwareentwickler, CAD-Administratoren und Schaltungsdesigner zu unterstützen. Das zweite Konzept, das während der letzten Dekade im Bereich der Informatik entwickelt wurde, stellt ein Architekturmodell für Software dar, dass benutzt werden kann, um erweiterbare und wiederverwendbare objektorientierte Softwaresubsysteme zu erstellen. In dieser Arbeit wird ein objektorientiertes Entwurfssystem vorgeschlagen, das einen erweiterbare Satz von primitiven Datentyp für Entwurfsdaten und von Bausteine für Entwurfswerkzeuge enthält. Dieses objektorientierte Framework wird in ein CAD-Framework eingebunden und übernimmt eine wichtige Rolle bei typischen CAD-Framework-Diensten wie die Repräsentation und Verwaltung der Entwurfsdaten, die Versionsverwaltung, die Benutzerschnittstelle, das Designmanagement und die Werkzeugintegration.
Das im Rahmen dieser Arbeit implementierte CAD-Framework namens 'Cave2' ist nach der klassischen Schichtenarchitektur, die von Barnes, Harrison, Newton und Spickelmier vorgeschlagen wurde, aufgebaut, allerdings erlauben die Möglichkeiten, die die Nutzung von objektorientierten Konzepten eröffnen, eine Reihe von Verbesserungen, die in früheren Ansätzen nicht realisierbar waren:
- objektorientierte Systeme sind schon vom Ansatz her erweiterbar, daher sollte dies auch für die implementierten Sätze von primitiven Datentypen für die Entwurfsdaten und für die Bausteine für Entwurfswerkzeuge möglich sein. Das bedeutet, das sowohl die Repräsentationsmodell des Entwurfs als auch die Softwaremodule die damit verbunden sind, erneuert oder an eine bestimmte Designmethode angepasst werden können und das diese Erweiterungen und Anpassungen weiterhin die Architektur und die funktionalen Teile erben, die innerhalb des objektorientierten Systems implementiert wurden.

- die Semantik und die Visualisierung des Entwurfs sind Teile des objektorientierten System, benutzen aber klar getrennte Modelle. Dies ermöglicht verschiedene Visualisierungsstrategien für einen gegebenen Datensatz und ermöglicht damit den zusammenarbeitenden Parteien individuelle Einstellungen für die Visualisierung zu wählen.
- die Kontrolle der Konsistenz zwischen Semantik und Visualisierung, die sehr wichtig ist für eine Entwurfsumgebung mit mehrere Ansichten desselben Entwurfes, ist in dem objektorientierten Framework enthalten. Solch ein Mechanismus ist generisch genug, um in späteren Erweiterungen des Datenmodells für die Entwürfe benutzt zu werden, da er die Kontrollrichtung zwischen Ansicht und Semantik umkehrt. Die Ansicht empfängt eine Benutzereingabe und reicht dieses Ereignis an das semantische Modell weiter, dass dann die Möglichkeit einer Zustandsänderung bewertet. Ist diese möglich, so wird eine Änderung des Zustandes von sowohl Semantik als auch der Ansicht eingeleitet. Dabei wird die Änderung der Kontrollrichtung ausgenutzt und eine Schicht zwischen dem semantischen Modell und der Ansicht eingefügt, um die Konsistenz bei mehreren Ansichten sicherzustellen.
- zur Optimierung der Konsistenzkontrollmechanismen zwischen der Ansicht und der Semantik wird ein ereignisbasierter Ansatz vorgeschlagen, der alle diskreten Interaktionen des Designers mit der jeweiligen Ansicht speichert. Die Information über jede Interaktion ist in einem Ereignisobjekt gekapselt, das zu der Semantik des Entwurfs und auch zu möglichen anderen Ansichten weitergeleitet werden kann, je nach den Konsistenzregeln, die zu dieser Zeit benutzt werden. Weiterhin ermöglicht die Nutzung von Ereignisspeichern die spätere Synchronisierung zwischen der Ansicht und der Semantik, falls zwischen diesen temporär keine Netzwerkverbindung zur Verfügung steht.
- die Nutzung von Proxy-Objekten steigert die Abstraktion der Integration von Ressourcen zum rechnergestützten Schaltungsentwurfs signifikant, da sowohl auf entfernte als auch lokale Werkzeuge und Diensten durch Methodenaufrufe in einem lokale Objekt zugegriffen werden kann. Für die Verbindungen zu den entfernten Werkzeugen und Diensten wird ein Look-Up Protokoll benutzt, das ebenfalls komplett von der Position der Ressourcen im Netzwerk abstrahiert. Dies ermöglicht außerdem ein Hinzufügen und Entfernen von Ressourcen zur Laufzeit.
- das implementierte CAD-Framework basiert ausschließlich auf Java-Technologie. Daher benutzt es die Java Virtual Machine als Schicht, die die Unabhängigkeit zwischen CAD-Framework und dem Betriebssystem sicher stellt.

Diese Verbesserungen garantieren einen höheren Grad der Abstraktion bei der Verteilung von Ressourcen zum rechnergestützten Schaltungsentwurfs und führen ein neues Paradigma für die entfernte Interaktion zwischen Designer ein. Das resultierende CAD-Framework unterstützt die feingranulare Zusammenarbeit, die auf Ereignissen basiert, so dass jede einzelne Entwurfsänderung, die von einem Designer durchgeführt wird, an den Rest des Entwicklungsteams weitergeleitet wird, unabhängig von ihrer Position in dem verteilten Umgebung. Dies kann das Gruppenbewusstsein stärken und ermöglicht einen besseren Erfahrungsaustausch, was das Potential zur Zusammenarbeit, im Vergleich zu früher vorgeschlagenen dateibasierten oder recordbasierten Ansätzen, signifikant verbessert.

Drei verschiedene Fallstudien wurden durchgeführt um die Gültigkeit des vorgeschlagenen Ansatzes zu überprüfen, bei der jede auf einen Teil der Beträge dieser Arbeit fokussiert ist. Die Erste benutzt die Proxy-basierte Architektur zur Ressourcenverteilung um eine Prototyping-Plattform mit rekonfigurierbaren

Hardwaremodulen zu erstellen. Die Zweite erweitert die Grundlagen des implementierten objektorientierten Frameworks, um den Interface-basierten Entwurf zu unterstützen. Diese Erweiterungen, nämlich die Primitive zur Repräsentation eines Entwurfs und die Werkzeugblöcke werden genutzt, um ein Programm namens 'IBlaDe' zur Entwurfseingabe zu implementieren, dass die gemeinschaftliche Erstellung von funktionalen und strukturellen Modellen eines integrierten Systems ermöglicht. Die dritte Fallstudie betrifft die Möglichkeit einer Integration von Multimedia-Metadaten in das Datenmodell für den Entwurf. Diese Möglichkeiten werden im Rahmen einer Online-Ausbildungs- und Trainingsplattform erforscht.

Stichwörter: Mikroelektronik, CAD, verteilte Systeme, rechnergestützter Entwurf integrierter Systeme, rechnergestützter Kooperation, Java.

1 Introduction

1.1 Motivation

The interoperability between design tools has been one of the most important research topics covered by the design automation area in the last thirty years. Recently, the interoperability between designers started to get attention: collaborative design. There is a need for techniques tailored to support the communication, coordination and data sharing between groups of designers. The reason is obvious: the complexity of integrated systems design is increasing much faster than it was predicted [BRW2000]. The market is also shifting slowly from the PC-based paradigm to a scenario where the computational power one needs is distributed among several smaller so-called information appliances. While these smaller information appliances may look simpler than a PC to the user – and this is what makes the new paradigm better – the complexity of the design of those devices is still high.

As an additional problem, the shortage of qualified design engineers obliges the companies to make use of the working force, no matter where they are. This adds to the complexity of the collaborative design environment, as it must handle design teams which are geographically dispersed.

Taking such scenario into account, this thesis covers technical issues that arise on the design and implementation of the infrastructure to support the collaborative design of integrated systems over a distributed environment.

The research topic called collaborative design can be considered as a blend of computer-aided design and computer supported collaborative work - usually associated to the terms CAD and CSCW, respectively. Both fields incorporate already a significant amount of knowledge, so a complete review is unfeasible. The interdisciplinary research involving both areas is also well developed, specially in the areas of CAD for mechanical and civil engineering. In the area of specialization of this thesis – collaborative design of integrated systems - the research activity is still incipient, and most of the approaches available in the literature are reviewed in [IND2002]. The most relevant among them are also covered within this text.

The approaches on collaborative design can be characterized according to the time-space matrix of CSCW (Table 1). Considering the scenario where designers are dispersed geographically, we assume that the software infrastructure should provide support for the two types of collaboration in the second line of the matrix.

TABLE 1.1 - CSCW Time-space Taxonomy

Space/Time	Same Time	Different Time
Same Space	face-to-face interaction	asynchronous interaction
Different Space	synchronous distributed interaction	asynchronous distributed interaction

All the previous approaches on integrated systems collaborative design reviewed in [IND2002] can be characterized as supporting asynchronous distributed interaction among designers. Furthermore, many of them – such as WELD [CHA98], ASTAI(R) [CLA2001] and OmniFlow [LAV2000] - are mainly based on workflow concepts, thus not very suitable for processes that cannot be modeled as a regular chain of tasks.

The work presented in this thesis intends to overcome such limitations by supporting the following features, which could not be found in the currently available products and research prototypes:

support for both synchronous and asynchronous collaboration, because we can expect different levels of collaboration in the different parts of the integrated systems design flow. For instance, it is expected a high level of collaboration on the first steps of the design - where product functionality and constraints are defined - because of the inherent multidisciplinary nature of such activities. Hardware engineers, software developers, marketing staff and product management are among the types of professionals that may participate synchronously in such collaborations. On the other hand, during the implementation steps - coding, hardware debugging, etc. - the collaboration level is expected to be low, because developers tend to work individually and asynchronously;

support for fine-grained collaboration, in order to increase the potential of concurrent development. Most of the systems found in the literature are file-based, so the potential of synchronous collaboration is reduced as the concurrency control is done by the underlying file system and is based in simple locking;

handle design models in a variety of formats and abstraction levels without reducing the potential of collaboration and exploration. When many data formats are available, the common practice found in the literature is to handle all formats as black boxes, ignoring the semantics of each model. An ideal solution should be able to handle design data in different formats, but still recognizing the semantic constructs which are peculiar to all of them;

support multiple forms of design visualization, because collaboration can be much more effective if the right visualization technique is used. The "right" visualization depends on many factors, such as the background of the designer, the type of data, etc., so the proposed

system should be flexible enough to allow a design data block to be rendered as a diagram, 3D graphic, text, equation, etc.;

support the integration of multimedia metadata in the design data model, because it can increase the potential of collaboration. In the current designs, most of the asynchronous collaboration is based on metadata included as comments by the designers within the design models. The proposed system should support this feature, and should also expand the possibilities of learning and collaboration by allowing multimedia metadata - text, hypertext, graphics, sound, video, etc.;

allow a variety of functionally-equivalent implementations, in order to avoid excluding software and hardware platforms from being part of the design environment. So, the proposed architecture for the design environment should regard abstract relationships rather than concrete implementations.

Other requirements, which are fulfilled in some of the design automation solutions found in the literature, should be also met by the solution presented in this thesis: extensibility, adaptability, integration of external tools and services, support execution over Internet and support multiple distributed users in a scalable way.

Such are ambitious goals, and can only be achieved by a complex software infrastructure. To build such structure, several engineering techniques were employed and some original solutions were devised. The core of the proposed solutions is based in the joint application of two homonymic technologies: CAD Frameworks and OO frameworks. The former concept comprehends a layered software environment which aims to support CAD tool developers, CAD administrators/integrators and designers. The latter, a software architecture model to build extensible and reusable object-oriented software systems. To avoid misunderstandings, the former will be always referenced with the first letter capitalized throughout this text.

The following subsection describes how such techniques and solutions are organized within this thesis.

1.2 Thesis Organization

The thesis text starts by reviewing the concept of CAD Frameworks in Section 2, first by providing an overview on the integrated systems design tools and methods, and then describing the actual CAD Framework services supporting those tools and methods. This section is followed by a bibliographical review on CAD Frameworks and similar approaches aiming to support multi-user and/or networked access to design data and automation tools. The time span of the reviews included in

this section starts in the mid-80's and goes up to the early 2000's, highlighting the most significant contributions and shortcomings of each approach. The original implementation of the Cave Framework, which is used as starting point of the work presented in this thesis, is also reviewed.

Section 4 is a transition between the initial part of the thesis, where the previous related work is reviewed, and the core of the thesis where the actual contributions of this work are described. It shows the transition between the original Cave Framework and what became the Cave2 Framework, thus addressing the techniques that made possible the architectural evolution.

In Section 5, the core of the Cave2 architecture is described: its object-oriented framework. This framework lays the foundations for two extensible sets of object classes: design data primitives and design tool building blocks.

Sections 6 and 7 describe in detail all the extensions and services built on top of the object-oriented framework aiming to support distributed and collaborative design, respectively. The achieved contributions on design databases, distributed CAD systems and collaboration support are introduced in those sections.

Section 8 presents three case studies, where the core of the framework was extended and/or services were added in order to achieve specific goals. The first case study uses the support for resource distribution to create an online platform for prototyping in FPGA systems. The second one extends the framework of data primitives and tool blocks to create a design entry tool tailored to the interface-based design methodology. The third case study describes an extension of the data primitives framework to support the reuse of design metadata as learning material.

Section 9 closes this thesis by summarizing the its contributions, deriving conclusions and pointing out directions for future work.

2 Design Automation Frameworks

2.1 Introduction

Integrated electronic systems are among the most complex artifacts created by men. Decades ago, when the first integrated circuits were developed, small groups of engineers could handle the design without sophisticated computer aid. Nowadays, integrated systems can be seen as a heterogeneous composite of programmable modules, packaged together in a single device. As important as the modules themselves, the programming information for each module is also a product of the design process. Thus, large groups of designers are needed to design such systems, and they depend heavily on a variety of design automation tools. Furthermore, a methodology based on abstraction layers is also critical to allow the modularization of the design activity.

The concept of Electronic CAD Frameworks was crafted from the need to support the numerous tools which are needed in the design cycle of an integrated system. Such concept evolved over the years, incorporating new engineering techniques to better serve its purpose of:

support tool developers by providing tool building blocks (to accelerate implementation and to grant homogeneity) and tool interfaces (to grant interoperability with other tools and data repositories);

support tool administrators by providing a platform where tools and data repositories can be integrated and managed together;

support designers, by providing an integrated environment for the complete design flow.

In the following subsections, a review on the concepts of integrated systems and integrated systems design is presented; the classical definition of CAD Frameworks is given; and an overview on the techniques applied to CAD Frameworks is included.

2.2 Integrated Systems Design

Integrated systems can be described as heterogeneous composites of programmable modules, packaged together in a single device. Those modules can be, for instance, digital or analog circuitry, micromechanical parts, radio frequency (RF), electro-optical and even electro-biological structures. As important as the modules themselves, the programming information for each module is also a product of the design process.

Figure 2.1, published by the Semiconductor Industry Association, shows the technologies which currently can be integrated to the standard CMOS fabrication process and those which are going to be integrated in the following years. Such process allows the production of ICs where different types of modules can be put together in a single die - the so-called System-on-a-Chip (SoC).

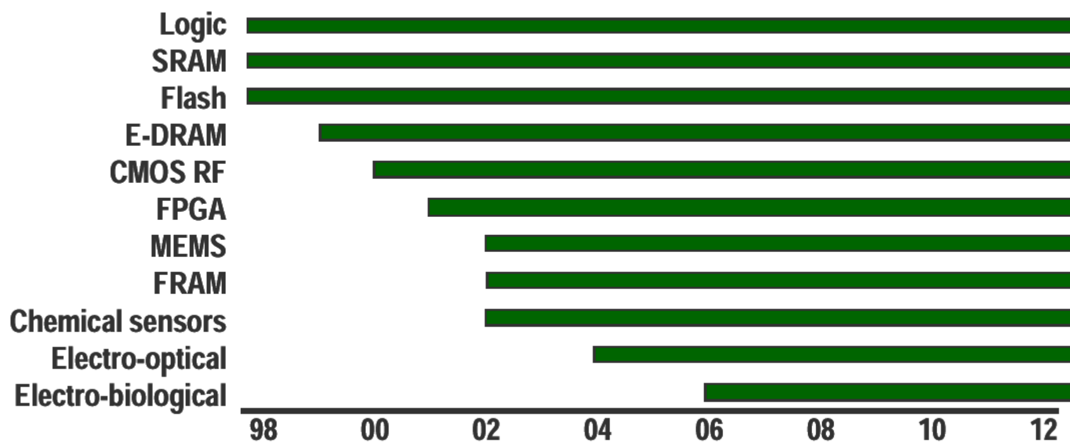


FIGURE 2.1- Technologies integrated on SoC in the standard CMOS process [SIA99]

The design of integrated systems of such complexity can only be achieved by adopting adequate methodologies that allow the decomposition of the problem in smaller parts. Current practices are often a mixture of a bottom-up approach, where the system is composed by the integration of previously designed blocks (IP cores), and a top-down approach, where an initial description is synthesized into a detailed implementation model. In either case, a layered approach is necessary, in order to ensure that the design flows across a series of well defined abstraction layers. The design methodologies define the systematic use of a set of transformations, from the initial description to the final system. Some of the transformations add new information to the system description, while others are aimed to verify the correctness of the description or extract from it information which wasn't explicitly there. The former type is usually called synthesis while the latter, analysis. To cope with the increasing productivity requirements, more abstraction levels, and thus transformations, are added to the design process.

In Figure 2.2, a typical design flow for integrated systems is depicted, showing the transformations between different kinds of descriptions.

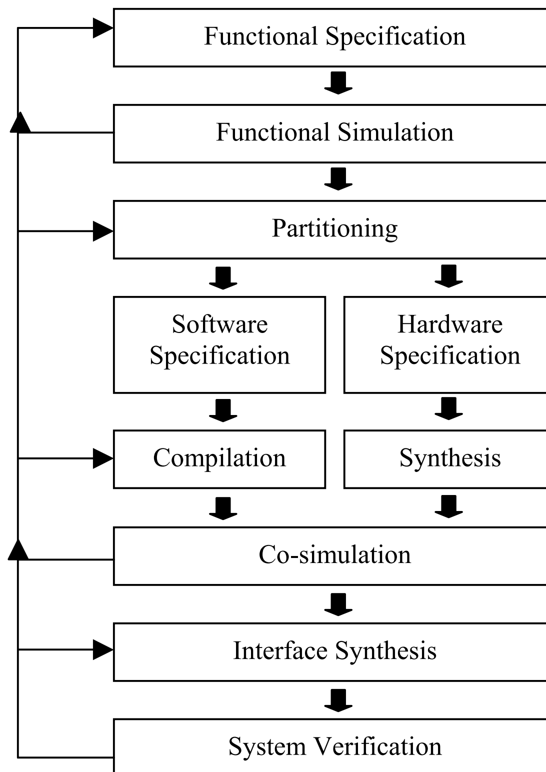


FIGURE 2.2 - Simplified System Design Flow

2.2.1 Functional Specification and Validation

The design usually starts in a very high level of abstraction, by describing the intended functionality of the system: - system-level specification [SAN2000]. This description disregards every implementation detail, focusing only in the system behavior and its interactions with the external world. The system description can be done using one or more languages. The SystemC approach [SWA2001], for instance, advocates for a single specification language, in order to ease the interoperation of design tools and reduce the costs of the design within the industry. In other hand, the TIMA research group [JER99] and the Ptolemy Project [LEE2001] focus in the interoperation of languages and modeling styles. Other approaches for system level design include Ocapi [DES2000], SpecC [GAJ2000], SDL [ELL97] and Forge [DAV2001].

Some of the languages used for system specification have formal semantics, with underlying mathematical structure - e.g., Petri nets, finite state machines - while others derive from previously developed HDLs or programming languages. Visual languages and/or visual extensions for textual languages are also among the alternatives for system modeling.

After the modeling step, a functional validation takes place. This is done by simulating or executing the system model, so that the functionality can be verified. No performance tests are executed on this phase, because no assumptions about the implementation were made yet. If the functional requirements of the system are not met, the model should be reviewed, otherwise the next step of the design flow - model partitioning - is started.

2.2.2 Partitioning

The partitioning problem can be defined as the mapping of the expected system functionality to the components which are expected to build the system. Examples of components in typical hardware/software systems are standard processors or microcontrollers - and the software to be executed on them -, custom ASIC chips, memories, busses, configurable logic. So, the partitioning procedure takes as input a functional model of the system and separates the functions which are going to be implemented by each one of the components. It is important to notice that the procedure actually starts by the decision on which components will actually be part of the implemented system. This decision, obviously, strongly influences the partition itself.

The concept of platform-based design [SAN2000] was introduced in order to reduce the complexity of this task. According to this concept, the set of components which is used to build a system is strongly related to its application domain. So, by establishing a well defined set of components - a platform - and by validating it in a particular application type, it could be reused in future designs within such domain. By relying on already developed and validated platforms, the partitioning step can be done more easily, by mapping automatically the system functionality to the platform modules. Companies such as Coware [VAN2001] and Cadence [CAD2001] are known to support the concept of platforms.

Besides the choice of the system components to which the functionality will be mapped, other key issues on the partitioning step must be highlighted: abstraction level of the functional specification (task level, behavioral level, etc.), granularity (amount and complexity of the functional units resulted by the decomposition of the functional specification) and the details about the partitioning algorithm itself (metrics of quality, cost function, solution space covering strategy, etc.) [BEC97].

2.2.3 Software and Hardware Specification, Simulation and Implementation

Usually, a great amount of the system functionality is mapped into software during the partitioning step. [ARN2000] states that up to 80% of a system is software running on a platform of general purpose or custom processors (CPU and/or DSP) tightly coupled with unique dedicated hardware. While the software part show more flexibility, allowing simpler error correction and upgrades, the part implemented in dedicated hardware has superior performance, so it is used for the time-critical functionality of the system.

The software specification generated from the partitioned system description is usually programming language source code. When the platform where the software is going to run is pre-existent, there is usually a compiler available to generate object code, as well as a set of software drivers, so the software modules can access the dedicated hardware parts transparently. The use of a RTOS may also be considered, as it provides commonly needed functionality on the software/hardware communication layer.

In many cases, a simulation engine is also available, so the software modules can be tested over a software emulation of the hardware/RTOS platform. Minor corrections may be done directly in the generated source code, but major revisions should be done in the system model, so the partitioning can be re-done to ensure better results.

However, in most of the cases there is a need for customization in the underlying platform. This customization is defined by the hardware specification taken from the partitioned system description. It is usually HDL code, which should be simulated together with the software modules and its underlying platform. This procedure is called co-simulation. Again, minor corrections can be done directly in the HDL code, but if major corrections are necessary, it should be done in the system specification. Once the co-simulation shows the desired results, the synthesis of the hardware modules can start, as well as the synthesis of the communication structure that allows the interoperation of the hardware modules and the platform that runs the software part. Such synthesis is by itself very complex and will be described in details in the following subsection. Once the customization of the underlying platform is done, it is necessary to ensure that the software modules would be able to run optimally over it. New drivers must be implemented, to make the bridge between the software modules and the customized hardware, and - if the software processing hardware was also customized - new compilers must be generated.

2.2.4 Hardware Synthesis

The synthesis of the hardware modules and the communication circuitry is a very complex task by itself. After the system partition and communication generation, those modules are described in a high level of abstraction using a HDL. In order to translate such abstract description into actual hardware, a set of model transformations must be done. Such process is based on techniques developed over more than three decades of research, many of them covered in [DEM94]. It can be subdivided in sub-processes, regarding the abstraction layers of the initial and final design models. From the partitioned system model described in previous subsections, three steps are usually taken in order to reach a physical implementation of the system: behavioral synthesis, logic synthesis and physical synthesis.

In the behavioral synthesis, a high level model of the hardware part decomposed in a three sub-models:

a sequence graph, which defines the operations that must be performed by the circuitry, as well as the order that the operations should be executed;

a set of functional resources - usually a library of functional blocks - which are available for the implementation of the circuitry;

set of design constraints, which specify limits - for size, performance, power consumption, etc. - that should be respected by the final implementation.

The behavioral synthesis comprehends three stages. In the first stage, each operation on the sequence graph is scheduled, respecting the dependencies among them. Once the schedule is done, each operation must be assigned to a functional block. To minimize area, each functional block must perform several non-concurrent operations. So, in the second stage the resource sharing is optimized so that a minimum number of functional blocks can be found, still respecting the schedule previously done. Finally, the third stage - resource allocation - can be done, by explicitly assigning each operation to a functional block.

Following the synthesis flow, the next transformation - called logic synthesis - has as main goal the generation of a logic description of the circuit. The logic description - a net of logic gates, which are modeled as a set of Boolean equations - is necessary for the physical synthesis later on. Furthermore, several techniques can be applied during the logic synthesis in order to reduce the complexity of the final circuit, by reducing area and power consumption or even easing the testability.

Finally, the physical synthesis has the responsibility on the generation of the physical implementation of the circuit. Usually, this is done by mapping logic blocks - resulting from the logic synthesis - into pre-defined physical constructions, such as layout cells in an ASIC or configurable logic blocks in an FPGA. In the ASIC case, the standard cell approach is the most widely used. The layout cells are usually grouped in a library, possibly with alternatives for each cell - tailored for smaller area, higher performance, lower power consumption, etc. The libraries are closely related to the circuit fabrication process, so after this stage it is not possible to change the circuit fabrication technology. In the FPGA case, the mapping is done regarding the logic implementation of each configurable block

After the technology mapping, the relative position of the layout cells or logic blocks is defined. The connections among them - and the external world - are generated, following the connection between the blocks in the logic netlist in a procedure called Place-and-Route. Complex algorithms are used in this stage, in order to minimize the number and the length of the connections, because such factors affect significantly the circuit performance. Once the cells or blocks are placed and routed, final verification tasks are performed and the circuit can be implemented.

2.3 Design Automation Tools

As shown in the previous subsection, the design of integrated systems is complex, requiring a great amount of automation in order to be feasible. The automation of the design tasks is performed by specialized software running over general purpose computers. In some specific design automation tasks, however, specialized hardware can be required, such as high-performance computers used for simulation or configurable platforms used in emulation. But in most of the design flow, the design automation comprehends the creation and verification of design models, as well as the transformation of high level design models into optimized equivalent lower level design models. In order to assist the designer on each of the steps, a variety of design tools is needed. According to their functionality, the tools can be divided in the following groups: design entry and visualization, simulation, synthesis and verification. Further details and examples within each group can be found in [IND2002] and [REI2000a].

2.4 CAD Frameworks: the Classical Concept

A CAD Framework is a software environment which aims to support CAD tool developers, CAD administrators/integrators and designers [BAR92]. It provides automatic execution to some of the time consuming tasks performed by each of the three types of users, reducing the complexity of the design and increasing the productivity. So, a CAD Framework should provide mechanisms to support tool

development, tool integration and intercommunication, as well as to allow a simple and flexible usage of those tools.

Figure 2.3 shows the classic structure of a CAD Framework as proposed by [BAR92]. As one can easily notice, the system comprehends a number of abstraction layers built over the operating system of the designer's workstation. It was designed to hide from the designer the underlying complexity of the design automation software - only the tool developers and administrators would have access to the lower layers.

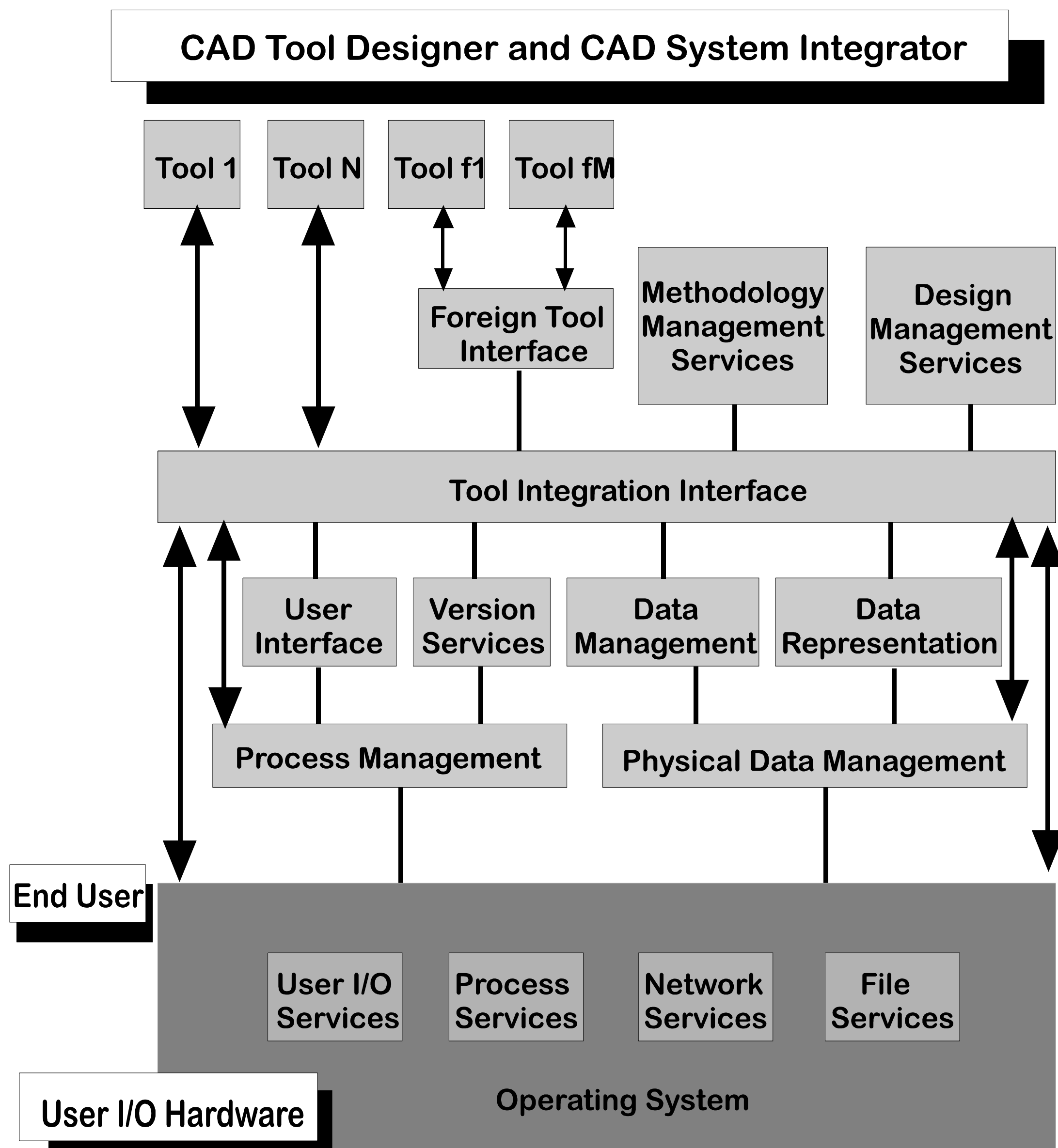


FIGURA 2.3 - CAD Framework architecture [BAR92]

In the following subsections, the architecture of a CAD Framework is analyzed in details. The functionality of each module is reviewed and the most significant advances obtained from the usage of such module is highlighted.

2.4.1 Operating System Services

The functionality of the CAD Framework is based on the operating system foundations. Among the services of the operating system which the Framework relies on are:

File Services, for data organization and management;

Process Services, for concurrent multi-program execution;

Network Services, for communication with processes and systems executed in different workstations;

User I/O Services, for the communication with the user and other peripheral devices.

It is not reasonable to expect that every operating system could be able to provide the same services, so an interface layer between the Framework and the operating system is needed. Such interface offers to the Framework a set of standard operations and maps it to the particular services offered by the host operating system. Basically, such operations involve physical data management and process management activities. Using such approach, the details of the particular operating system are hidden from the designers and tool developers, which should be able to deal with the Framework regardless of the operating system on which it is running.

2.4.2 Process Management Services

The processes which allow the execution multiple tools and data repositories concurrently in a single machine - and possibly in multiple computers in a network - should be managed properly within the Framework. Modern operating systems already offer some of the facilities needed for such management: network file systems, standard resource locators, support for concurrent process, etc. However, some services should be customized to the needs of a CAD Framework, for instance a load balancing system, in order to distribute among the machines of the network the processing load needed by particularly costly tasks [SCU95].

2.4.3 Data Representation and Management

The data representation and management services play a critical role on supporting design tool integration. Tools can only communicate if they share a common dialect - the way the design is represented - and a common communication protocol - the way they exchange design representation.

Due to the low complexity and small amount of the design data, the first generation of CAD Frameworks didn't have specific services for data representation and management [BAR92]. As the complexity of the systems to be designed grew, binary and textual data structures were created - often specific to a particular tool - in order to represent specific steps on the design flow, such as layout descriptions and logic schematics. Since many design tools were implemented separately by different teams or vendors, several data formats were created. To grant its interoperability, data format conversion tools were widely used and every design environment integrating a number of design tools was already distributed with a set of translators.

With the increase of the number of tools needed in the design flow, as well as the evolution of the data formats, resulting in several versions for each one, the implementation and maintenance cost of a set of translators become to high. The first solution to be proposed was the adoption of standardized exchange formats, which should be understood by every tool - each tool would have its own format translator internally, performing the conversion of the standard format into its own internal data representation. Languages such as CIF (Caltech Intermediate Format) [SHE93], EDIF (Electronic Design Interchange Format) [ELE2000] or even VHDL and Verilog are examples of widely used tool exchange formats.

Later on, more complex issues arose and the data management for concurrent designers became one of the main problems to be solved within the design automation Frameworks. Version management, active references from every design entity to each one of its instances (if the entity is edited, how to propagate the change to the instances), data consistency control for forced interruptions in the design process, access control for concurrent edition over a particular design entity are among the issues which were being researched more recently in this field.

An evolution can also be noticed when it comes to the data storage facilities. In the former Frameworks the storage was done completely over file systems. When the data management issues became critical, database technology was widely used as foundation for the storage systems within CAD Frameworks. As CAD applications require support for storage and retrieval of complex data objects and its relationships, many custom databases and data models were implemented [WAG94].

2.4.4 Design and Methodology Management Services

Design and methodology management tools are often called meta-tools, because they don't deal directly with design data itself but support the designer interaction with the design data and tools. The multiple tools needed by the designer during the design process are often organized in a so-called design flow. The tools that manage the design flow of an integrated circuit are responsible for the correct sequence of steps taken by the designer while going from the initial specification to the final implementation. The basic approach was based on sequences of automatic tool invocation, which were supposed to support each of the tasks performed by the designer. Besides that, the design flow management should also take care of the storage of the different visions of the design data, produced and consumed by each tool.

One of the first problems to arise when design flow management systems were introduced was the need for flexibility on the flow automation. Such flexibility is needed because the design steps and tools are constantly being updated, due to the increase of the complexity of the designs and, as consequence, the improvement of the CAD techniques. The design flow modeling should be as generic as possible [KWE95], so it can be adapted easily to face the evolution of the design methodologies and tools.

The creation of generic design flows was often based on the association of design tasks and automation tools. Such approach saved the designers from the tool invocation and the data transfer from one tool to another - format conversions could be also be done automatically when needed. As advantages, the process of design would be straightforward and faster. Furthermore, it would be performed in a standardized way, making easier the communication and design data exchange among members of a design team (or even the replacement of one of them), once all of them would be following similar design procedures. In opposition to the benefits, a set of issues had to be solved or optimized by design flow management tool designers [JAC95, BOS95, WAG94, BRE95]:

- design methodology modeling, so the management of the design flow can follow pre-defined, well known methodologies and styles;
- multi-user design flow, when the design tasks are performed by several designers, so the interdependencies among tasks should be handled properly;
- facilities for the storage of milestones, from which the design can be continued in multiple flows, allowing alternative implementations for the same design, for the sake of comparison;
- design metrics evaluation, in order to support the analysis of the project status, productivity evaluation, quality of the design, etc.

The problems which are dealt by process management tools in integrated systems design are generic, shared by many CAD environments from other engineering disciplines. In spite of that, few technological advances have been shared by them, and process management services have been developed individually for each domain.

2.4.5 Tool Integration and Encapsulation Services

Design tools can be incorporated to the design environment in several ways. Usually, we can classify them all in two groups: encapsulation and integration. The main difference is the degree of exposure of the tool internals to the Framework. In the encapsulation, the Framework has no access to the tool functionality, so it communicates with it only by data exchange. This approach is also called black-box integration or foreign tool interface. In the other hand, the integration of a tool requires direct access by the Framework to its internal structures - i.e. function calls, APIs, etc. - and is also called white-box integration. While the encapsulation can be done in nearly any type of tool, the integration assumes that the need for communication structures was predicted during the implementation of the tool, or that the source code is available for the implementation of such structures.

The management of the integration and encapsulation of tools comprehends the characterization and the control of such tools. For a small set of tools, the designer can manage the characterization herself, but for complex Frameworks, the amount of information to be managed is very large:

- tool name;
- tool version;
- physical storage of the tools executable and configuration files;
- online documentation;
- initialization procedure;
- runtime environment or shell;
- computational resources needed;
- input and output data formats;
- data repository configuration.

If those features are available to the design environment, it can automate the execution and data exchange for every incorporated tool, providing the designer with a single interface to control them all in simple manner. So, both integrated and encapsulated tools can be accessed transparently by the designer.

2.4.6 Data Versioning Services

A design data versioning service can be considered as specialization of the data management service, because it deals with the management of multiple sets of design data produced as several alternatives for a design transformation. On the other hand, it can be considered as a supporting technology to the design management services, because it supports the design team through its navigation over the design solution space.

The main functionality expected from a versioning service include:

- the maintenance of the multiple views of a design module, generated as the module is synthesized over the several abstraction layers used in the design process;
- the maintenance of multiple alternative solutions for a particular design problem, postponing for a later time the decision on which one would be implemented in the final design;
- the possibility to navigate backwards in the design history, so a previous state of the design can be restored and/or analyzed – such feature is essential when a wrong design decision was performed, or when the access to the situation from which the current design was derived is needed.

Several versioning strategies can be found in the literature. Some of them organize the versions in a linear fashion, allowing only multiple alternatives on the most recent versions. Some approaches are more powerful, allowing multiple alternatives for all of the versions of the design by modeling the version history as a tree or acyclic graph. Comprehensive reviews on the subject can be found in [KAT86, KAT91] and [WAG94].

2.4.7 User Interface Services

The study and development of interfaces between user and computer applications have been done since the early days of computing, because the user productivity, satisfaction and efficiency often depend on the quality of such interface. However, the evolution of the services of interface between designers and Frameworks was much slower than other Framework services describer earlier in this text, such as data representation and management. This situation is due to the limitations of the visualization devices available at the time of the introduction of the CAD Frameworks. In the beginning, the graphical capabilities of the displays were very limited, so the design process was almost completely done without direct manipulation of the design data. With the availability of graphical displays, graphic manipulation of the design data was done by specialized personnel, working only in

this particular task, because the costs to provide graphical workstations for the engineering personnel was too high.

Once the costs were bearable, and the techniques for implementing graphical user interfaces were refined - pointer devices, windows, menus, buttons, etc. - the designers started to work directly on the design.

Initially, the ability to interact graphically with the design data was mainly used to physical layout visualization and edition. Efficient data structures and algorithms were developed to allow the fast navigation and edition over very large sets of layout data [SHE93, TRI90]. As the levels of abstraction in the design activity were raised, the design visualization through interconnected block diagrams was widely used, both for logic-level schematics and for structural design based in HDLs.

Besides making easier the manipulation of the design data, the graphical interfaces helped also in design management: documentation, project management, communication among designers, etc. Furthermore, the use of an unified GUI "look-and-feel" within the Framework can also contribute on the sense of tool integration, because all the design tools would use similar UI components for similar commands.

3 Previous Work

3.1 Introduction

Several approaches already addressed the issue of multi-user distributed design of integrated systems. Most of them follow partial or totally the CAD Framework architecture presented in the previous section. Some of them even offered some kind of support to the collaboration among designers. The following subsections review the approaches which were more relevant and influential to the work presented in this thesis. Each subsection is introduced with the techniques the reviewed system offers as a support to the distributed, multi-user design of integrated systems. In the last subsection, a comparison among all the reviewed approaches is presented.

3.2 NELSIS

The NELSIS Framework [VAD90] provides support to distributed, multi-user design by implementing reusable methodology definitions, data versioning and supporting design data sharing among the designers. Its version mechanism is relatively simple, allowing several versions for a single design block. Each version has a status (working, actual, backup or derived), so a configuration mechanism can be implemented. NELSIS allows data to be shared between designers, but the concurrency control is based on the versioning support: each time the system detects a conflict between designers accessing the same design block, new versions for that block are created thus separating the conflicting code bases.

The methodology management services in the NELSIS Framework are performed by a methodology manager [TEN91]. It is responsible for creating flow-maps, which are sequences of interconnected activities. Each activity abstracts the (partial) functionality of an automation tool, as well as controls its execution parameters. Each activity has ports, which denote the data received and generated by the tool. Each data block has a defined type. The input ports are optional, and the output ports can be divided in modification and extension ports. By extension is meant that the produced data is stored in an existing design object. In the case of modification a new design object is created for storing the produced data. So, the data dependencies between activities are modeled through the interconnection of the activity ports. Figure 3.2 shows an example of a flow-map for layout design. The activities comprehend the layout edition, expansion, check, extraction and simulation. The activity ports are represented by diamonds. Optional input ports are represented as a filled circle, and modification output ports are represented as filled squares.

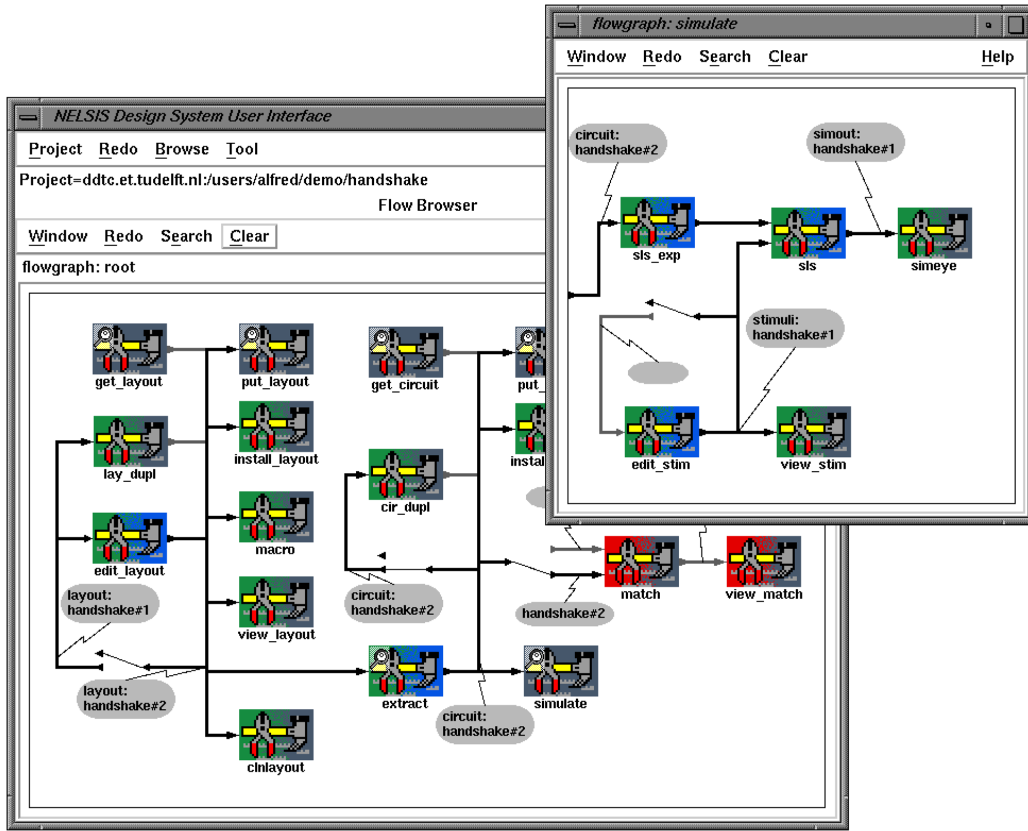


FIGURE 3.1 – NELSIS Design Flow Browser

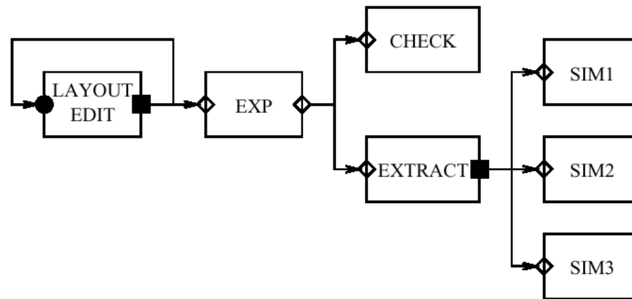


FIGURE 3.2 - A flow-map example [TEN91]

Hierarchical description of activities is also supported by the Nelsis Framework. The hierarchy can denote either a set of alternatives or a sequence of tasks. Figure 3.3 shows the same flow depicted in figure 3.2, but using hierarchical composition.

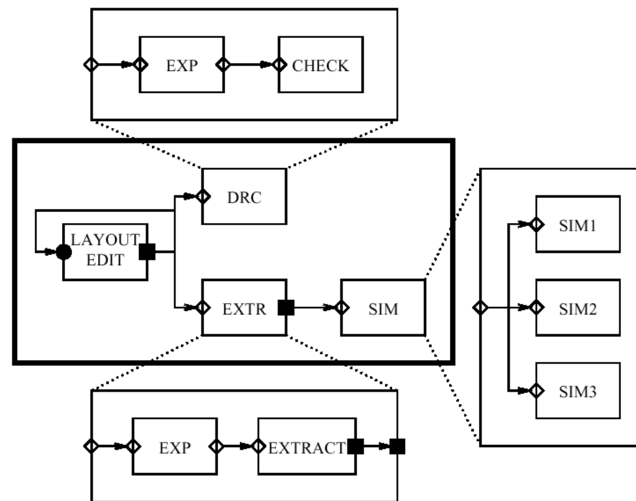


FIGURE 3.3 - An hierarchical flow-map example [TEN91]

3.3 Version Server

The Version Server provides support to distributed, multi-user design by implementing design data versioning. Proposed by Katz et al. in [KAT86], the Version Server is a database scheme, as well as its operational model, for generic design data. Basically, this approach proposes the inclusion of metadata within the design database, so special relationships among the data blocks can be modeled. Targeting multiple application domains, the metadata is neutral regarding the content of each data block, so the modularization of the design - as well as the granularity of the modularization - is let to the specific design tools, languages and its underlying modeling constructs.

In Version Server, three structural relationships were proposed: version, configuration and equivalence. The three relationships are described as three orthogonal planes. The version plane comprehends the version history of the data blocks. A tree-organized data structure is used to implement this scheme, contemplating both alternative and derivative versions for each design block. The configuration plane was responsible for the composition of several data instances, in order to form hierarchical design blocks. The third plane - equivalence - relates equivalent data blocks, specially in the case that they have different configurations and/or representations.

Besides the metadata model, the Version Server approach also proposes an operational model, based in the concept of workspaces. The server defines workspaces which can be semi-public, private or archive. The first is used to store and share incomplete and partially verified design blocks. The second allows access by a single designer, so it is mainly used for refinement. The third include validated instances, organized according to the three planes described before. A

transactional check-in/check-out mechanism is also proposed, in order to grant the consistency when moving data blocks from one workspace to another.

3.4 STAR

The STAR Framework [WAG94] provides support to distributed, multi-user design by implementing design data versioning. It is an extension of the GARDEN Framework [WAG91] and relies on some of the concepts shown previously in the Version Server, Oct [HAR86] and Damascus [MÜL88] Frameworks. It is designed to support the three dimensions of design evolution in such a flexible way that it is possible to incorporate consistently the successive design refinements into the design model. So, all the design decisions taken successively in every design step and every abstraction level are inter-related.

The STAR versioning scheme, shown in Figure 3.4, has the design object as the tree root. This object can have any number of viewgroups and views. The viewgroups – which are composite objects, aimed to provide n-dimensional hierarchy – can also have any number of viewgroups and views. Each view can have many viewstates, which store the actual design data for the design object. It is important to notice that the interface scheme of the design block - the set of ports used for connection with the external world – can be distributed all long the tree branch, because an inheritance mechanism is available within the module, so that the interface in a particular viewstate inherits the interface scheme from all its parent nodes. The inheritance of interface schemes is mandatory. Other attributes can also be inherited, but in this case the inheritance is optional.

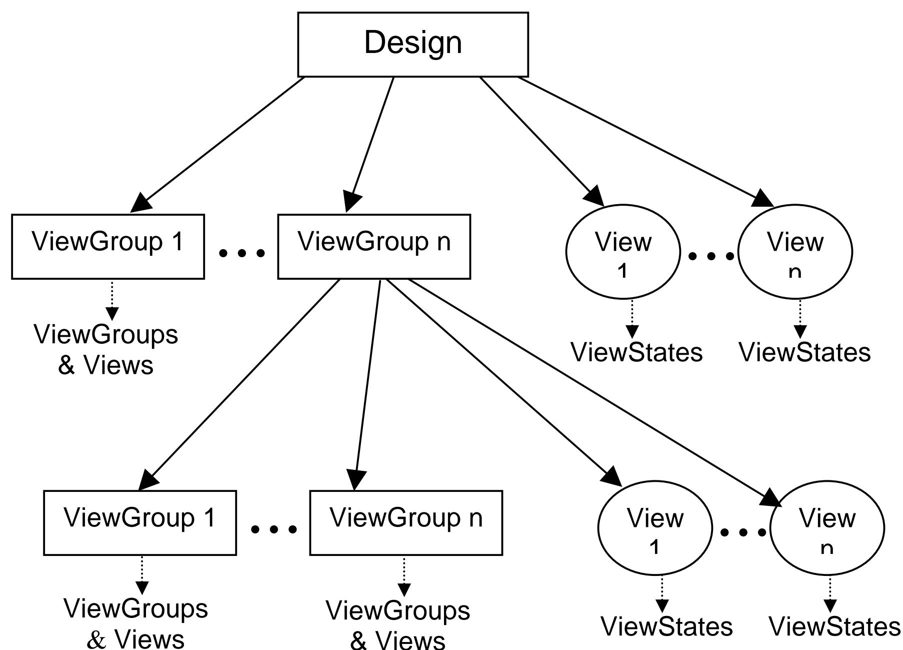


FIGURE 3.4 – Versioning in the STAR Framework [WAG94]

3.5 Ulysses and Odyssey

Ulysses and Odyssey are Frameworks which provide support to distributed, multi-user design by implementing reusable design methodologies. Relying on concepts developed in the fields of Artificial Intelligence and Expert Systems, the Ulysses Framework [BUS89] used a a knowledge base to support task execution and methodology management. Such knowledge base was intended to behave as an intelligent assistant to the designer, providing information about how to reach the design goals by supporting the scheduling of design tasks and offering details on CAD tool operation. Such approach intended to abstract from the designer the workflow model, in opposition to the approaches where the designer explicitly defines the workflow model.

The Odyssey Design Environment [BRO92] also offers such guidance through its Minerva module. It intends to support design planning by offering to the designer the possibility to state the design constraints in a so-called problem level. At this level, the designer can carry out design directly in terms of statements such us "synthesize an operational amplifier to meet a set of specifications" or "verify the performance of an ALU", rather than choosing specific tasks to achieve the desired goals. The plans created by the Minerva module are then modeled as a workflow by the Hercules and Cyclops modules. In opposition to Ulysses, Odyssey provides full support to user-defined workflow models [BRO92a].

The workflow model supported by Odyssey is a tree-like flow. For each desired result to be achieved - a circuit simulation or synthesis step, for example - both the design data and the automation tools needed for the result achievement should be included in the flow model. Once the model is ready, the workflow can be executed by doing instantiation of the automation tools and versioning of the design data. Such instantiation and versioning activities are included automatically in the flow model during the workflow execution. The completed flow can be stored and even re-executed if needed.

3.6 WELD

The WELD system [CHA98] provides support to distributed, multi-user design by implementing reusable design methodologies. It aims to provide a reliable, scalable connection and communication mechanisms for distributed users, tools and services. It proposes a three-tier architecture (Figure 3.5), consisting of:

remote servers, to provide access to either command-line tools encapsulated by server wrappers or tools with built-in support for socket connections and WELD communication protocols;

network services, such as distributed data manager, proxies and registry services, allowing the to incorporation of infrastructure components on demand;

clients applications, which use the WELD infrastructure to access network resources. Clients are either Java browser clients, or generic clients, developed in socket-enabled languages such as C, C++, perl, etc. using WELD protocols.

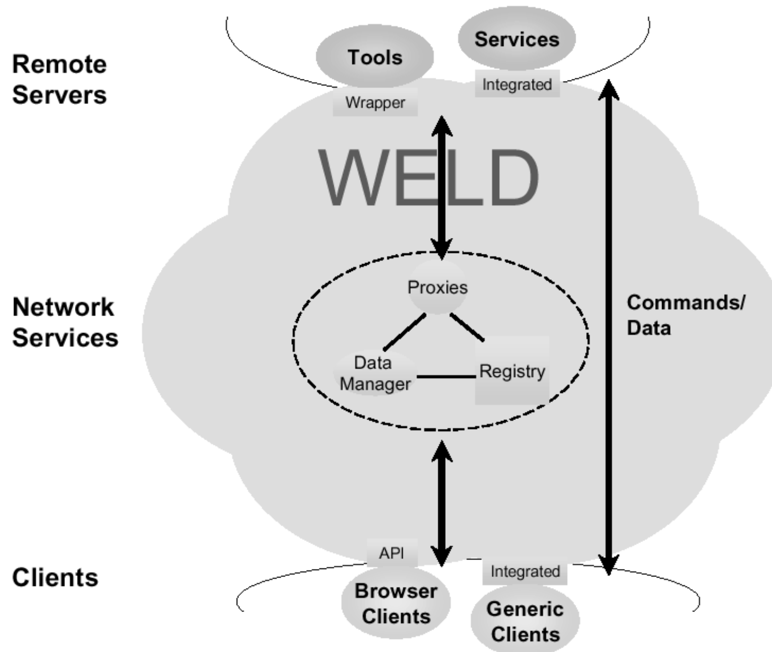


FIGURE 3.5 - WELD Architecture [CHA98]

An interesting feature should be noticed: the clients and the resources they access are loosely coupled because of the mid-layer of the WELD architecture. For instance, each time a client executes a particular task it may check on the registry for the network location of the service. By doing so, truly transparent distribution of tasks can be implemented, because the client can perform the same task in different servers without noticing. Furthermore, task execution servers can be added and removed without any noticeable effect to the clients.

However, this approach has also some side effects. Although command-line tools can be easily encapsulated on the remote server by using wrappers, other tools need to be re-written to conform to WELD communication protocols. The support for collaboration is also limited, once it does not provide a synchronous shared environment.

3.7 OmniFlow

The OmniFlow System provides support to distributed, multi-user design by implementing reusable design methodologies. Developed in the Collaborative Benchmarking Laboratory of the North Carolina State University, OmniFlow [LAV2000] [BRG2001] merges several engineering techniques - namely markup languages, hardware description languages and structured programming - to built a scalable and flexible workflow system.

The workflow model is based on the concept of markup languages, which became mainstream due to the success of HTML as the main language on WWW document construction. OmniFlow uses XML to capture the decomposition of the entire flow into a hierarchy of tasks, each of them associated to a software component. An XML Schema - named cdtML - was defined to allow the validation of workflow models which are to be processed by OmniFlow. Based on the scheme, a workflow model can be parsed and validated, and a GUI can be dynamically created by rendering the XML description of the workflow, so the user can view, edit and execute the workflow. Figure 3.6 shows a snapshot of a OmniFlow GUI. Within the GUI, the user can use structured programming constructs to control sequences of task synchronization, execution, repetition and abortion.

In order to attach the software components to the workflow system, as well as control the correct execution, [LAV2000] proposed a scheme based on the concepts found in HDLs: finite state machines. So, each task instance is controlled by a special structure which contains a finite state machine ([LAV2000] proposed the use of a Finite-State-Machine with a Datapath, referred as FSMD), a Control-Join (CJ), a DataMultiplexor (DM), a Control Fork (CF) and finally the actual software component, which can be attached either as a black-box or white box. The latter is the proposed construct to model hierarchies of tasks. This architecture is depicted in Figure 3.7. The FSMD, CJ, CF and DM jointly work on the following tasks:

receive data from previous tasks;

forward processed data to subsequent tasks;

synchronize the status of predecessor tasks and evaluate workflow/user defined conditions before invoking the current task instance;

validate the processed data according to user-defined constraints.

So, for each encapsulated task, the following operations are performed: (1) evaluate ControlJoin, (2) enable task, (3) execute component and (4) evaluate ControlFork. Operations (1) and (4) can halt the execution of the task if the pre or pos execution conditions are not met. Operation (3) depends on the type of the encapsulated software component: if it is a black-box, it is executed directly; but if it is a white box componend, it should be expanded and each of the child tasks should be processed according to the same set of operations.

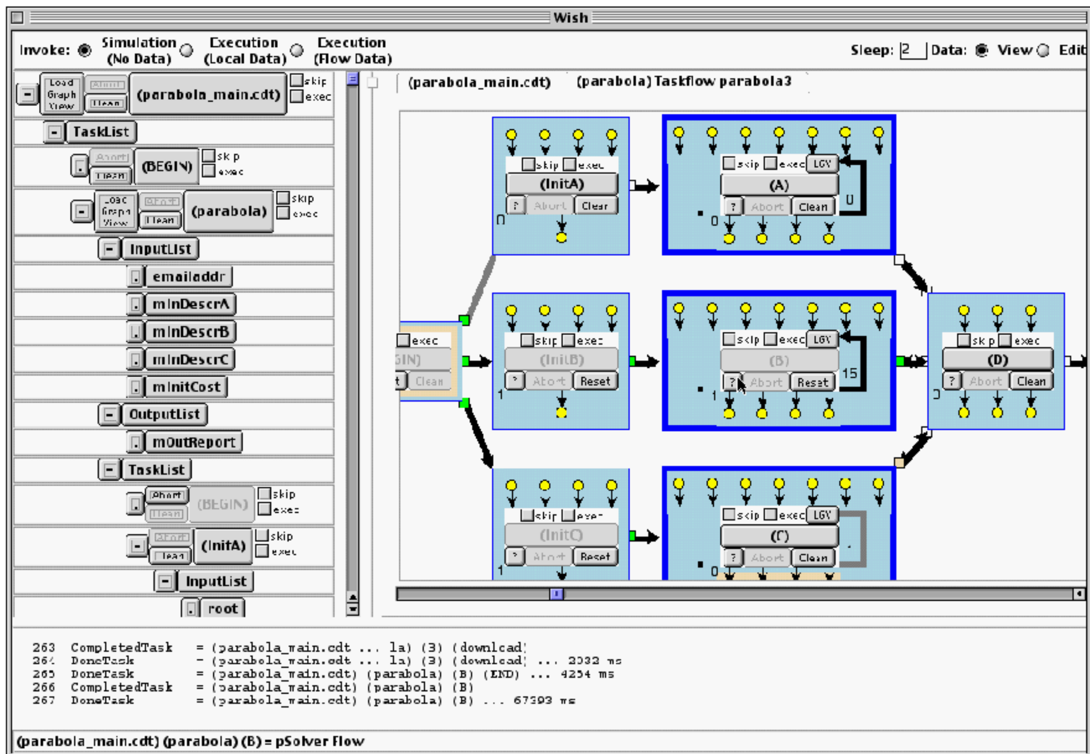


FIGURE 3.6 - OmniFlow Graphical User Interface

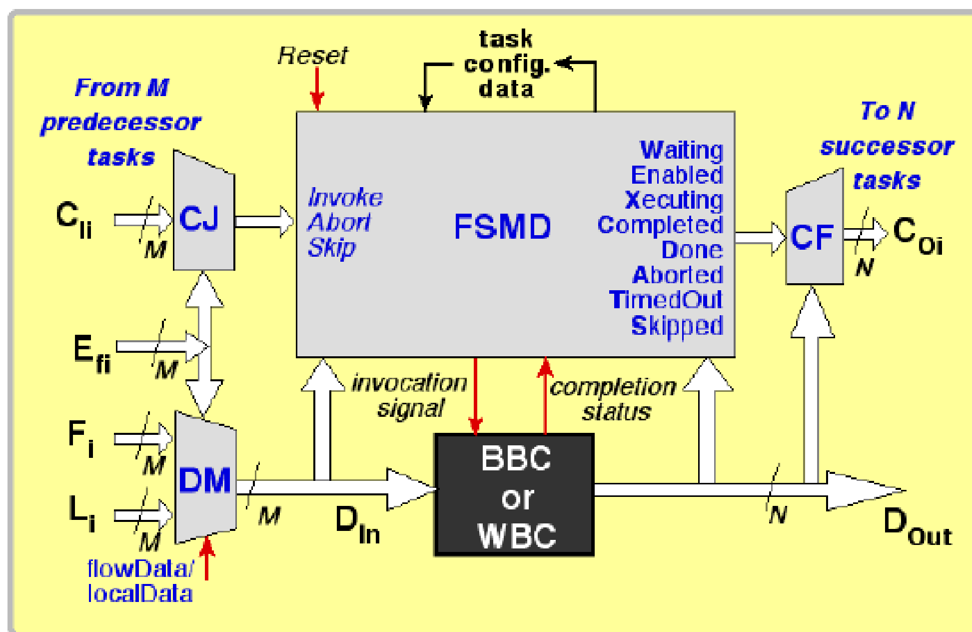


FIGURE 3.7 -OmniFlow Task Instance Architecture [BRG01]

The OmniFlow task instance architecture was reported to support all the workflow patterns reported on [VAE2000], showing its flexibility and expressiveness regarding workflow constructs. Furthermore, the authors claimed to

have modeled workflows with more than 9000 tasks - including a longest path of 1600 tasks - which demonstrates the system's scalability.

3.8 ASTAI(R)

The ASTAI(R) system provides support to distributed, multi-user design by implementing reusable design methodologies. Developed by the C-LAB research center in Paderborn, it provides distributed, multi user workflow management tailored to heterogeneous networks. It is a general purpose workflow management suite, but it was already used in electronic design automation applications [CLA2001].

The concepts embedded on the ASTAI(R) implementation are not state-of-the-art, but its production-quality distribution makes it a well documented, stable solution for EDA workflow modeling. It also integrates a version management module – the RCS system - as it allows automatic creation of data to allow undo/rollback operations on each workflow task. Furthermore, the versioning can be used to explicitly keep track on the evolution of any particular data object. An interesting feature resulting from the integration of a workflow system and a versioning system is that the evolution of the workflow model itself can be also managed by the versioning system, so a tree of versions of the workflow can be maintained.

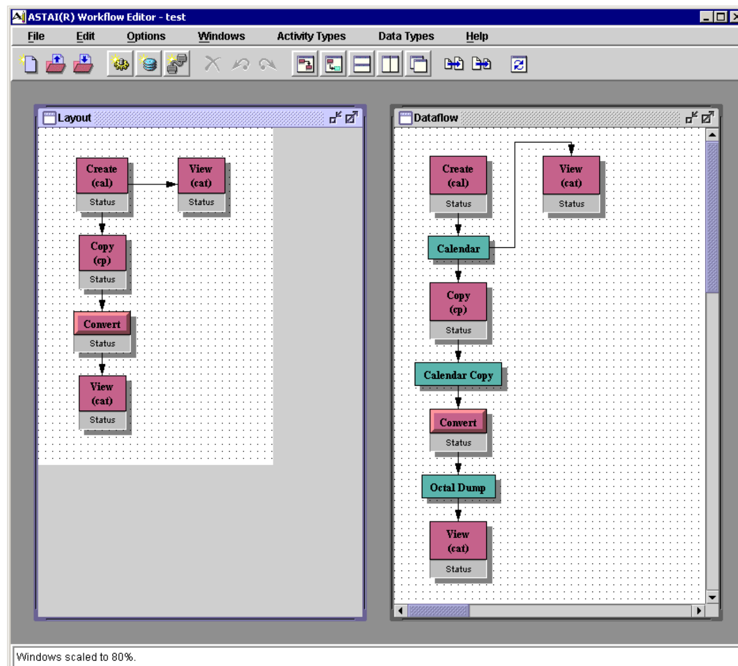


FIGURE 3.8 - ASTAI(R) Workflow Editor

3.9 Moscito

The MOSCITO Framework [SCH2002] was developed to support the distributed access to test generation tools. It provides facilities for the encapsulation of design tools and adaptation of the tool-specific control and data input/output to its internal formats. The encapsulation is done using the MOSCITO agents, which are interfaces between the tools and the MOSCITO kernel. Each agent must have a configuration file, defining the particular functionality of the tool it encapsulates. So, the kernel can invoke and configure the different tools through the agents in a standard way.

Another facility provided by the Framework is the workflow modeling interface. It allows user-created flows, as well as provides a set of pre-defined, often used workflow patterns. Once modeled, the flow is mapped into a chain of agents communicating via kernel. The Framework also provides facilities for visualization of messages sent by the executing tools, as well as support for viewers of known data types.

While not contributing to the state of the art on workflow systems - all of the features presented in MOSCITO were already implemented elsewhere - the software architecture was made simple and extensible, so its reuse in other application domains is probably feasible. The platform-independency granted by the use of the Java technology also contributes to its reusability potential.

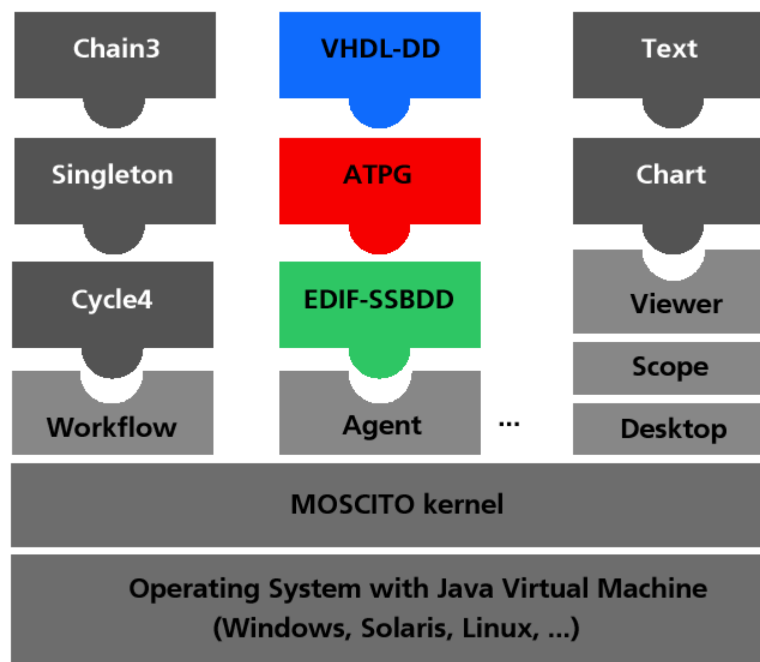


FIGURE 3.8 - MOSCITO Software Architecture [SCH02]

3.10 PPP

The PPP system supports the distributed, multi-user design by providing an abstraction to the distribution of design automation resources over the network. It was proposed by [BEN96] and presented an approach for the integration of tools running in different servers, accessible through a single user interface. Such approach is based on the remote tool execution over the WWW. In PPP, there are no tools installed in the client machine, so all the data processing is done on the server side.

The design cycles in PPP are initialized by the designer, which requests to the web server the desired tool. The server replies with an hyperdocument, where the designer can input the necessary parameters and design data for the tool execution. The data is then sent to the server, which executes the tool and send the results back, also formatted as a hyperdocument.

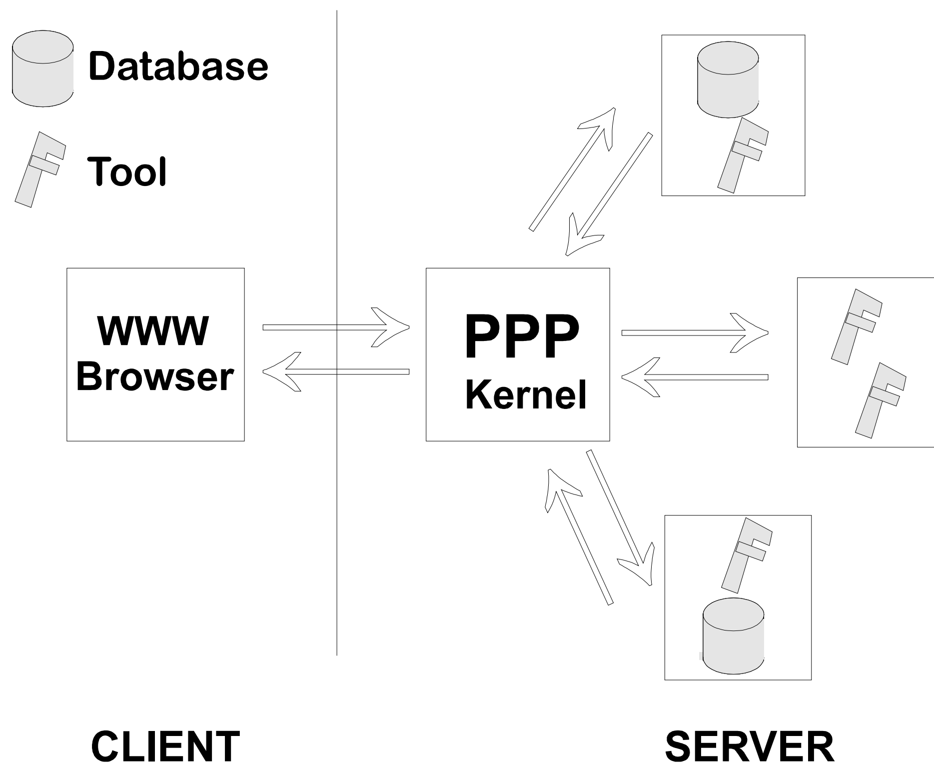


FIGURE 3.9 - Client-server architecture on PPP [BEN96]

Since the results are displayed in a web browser, some data representation formats which are used in regular CAD tools - such as schematics, waveforms, layout masks, etc. - may not be supported. To allow such data to be visualized, the PPP system includes a conversion tool, which maps the unsupported formats into data which can be properly handled by the web browser.

After the analysis of the results, the designer can repeat the procedure as many times as needed.

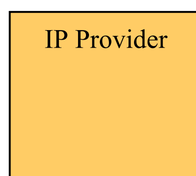
3.11 JavaCAD

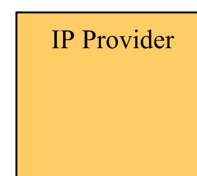
The JavaCAD system [DAL2000] supports the distributed, multi-user design by providing an abstraction to the distribution of design automation resources over the network. It used Java RMI to implement a distributed system for simulation of integrated systems composed by Intellectual Property (IP) cores. Java RMI is an important extension of the Java platform tailored to the development of platform independent distributed systems. It allows objects running on different machines in a network to communicate. The mechanism is build over Internet protocols, so it can work in a heterogeneous network (composed by different kinds of computers).

As shown in Figure 3.10, JavaCAD provides infrastructure for the distributed simulation of an integrated system, interconnecting the system designer and the providers of IP cores. This approach has the following advantages:

- the IP cores can be evaluated before licensing, because the designer can simulate his design together with the IP core without purchasing or even copying the IP content to his computer;
- the intellectual property of the provider is not disclosed during the evaluation procedure, because the evaluating designer can access only the IP core functionality, not its implementation.

The implementation of the JavaCAD approach is based on the use of proxy objects, which are installed in the designer's machine. During the simulation, such proxy objects - implemented using Java RMI - receive all the stimuli that the actual IP core should receive and forward through the network such stimuli to the provider's server, where the IP content actually resides. The stimuli processing are done and the results are sent back to the proxy object, so it can feedback the system under simulation.





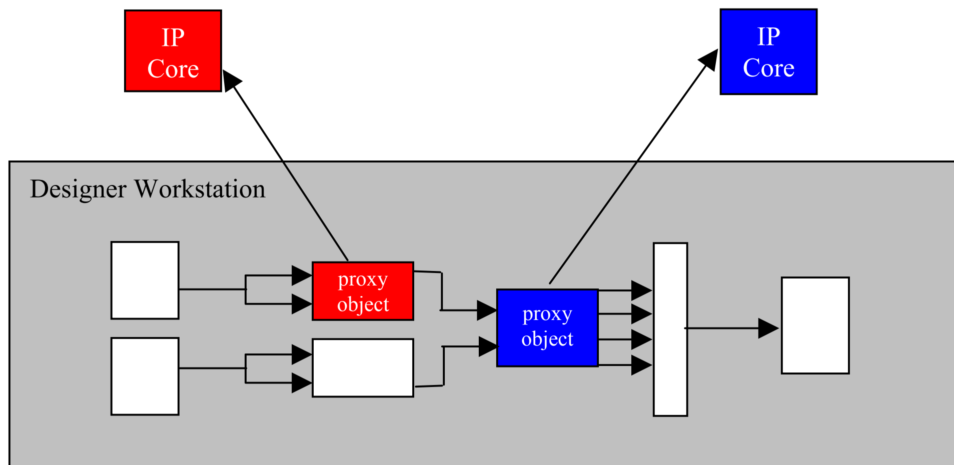


FIGURE 3.10 - Platform independent IP simulation

3.12 Ptolemy II

Ptolemy II [LEE2001] is a set of packages supporting the modeling and design of heterogeneous embedded systems. One of its major goals behind it is the possibility to design embedded software together with the systems within which it is embedded. Its software architecture is based on the concepts of entities, relationships, actors and domains. Its kernel package supports clustered hierarchical graphs, which are collections of entities and relations between those entities. Its actor package extends the kernel so that entities have functionality and can communicate via the relations. Its domains extend the actor package by imposing models of computation on the interaction between entities.

Ptolemy II does not specifically support the distributed, multi-user design of integrated systems, but its platform-independent architecture and its extensible set of modeling constructs influenced the development of the work presented in this thesis. The object-oriented framework which forms the kernel of the system provides a well-defined foundation for the extension of its modeling constructs. Such extensions can be reused by other designers, allowing for a code-level collaboration among designers and tool developers. Figure 3.11 depicts the kernel of the Ptolemy II system in an UML class diagram.

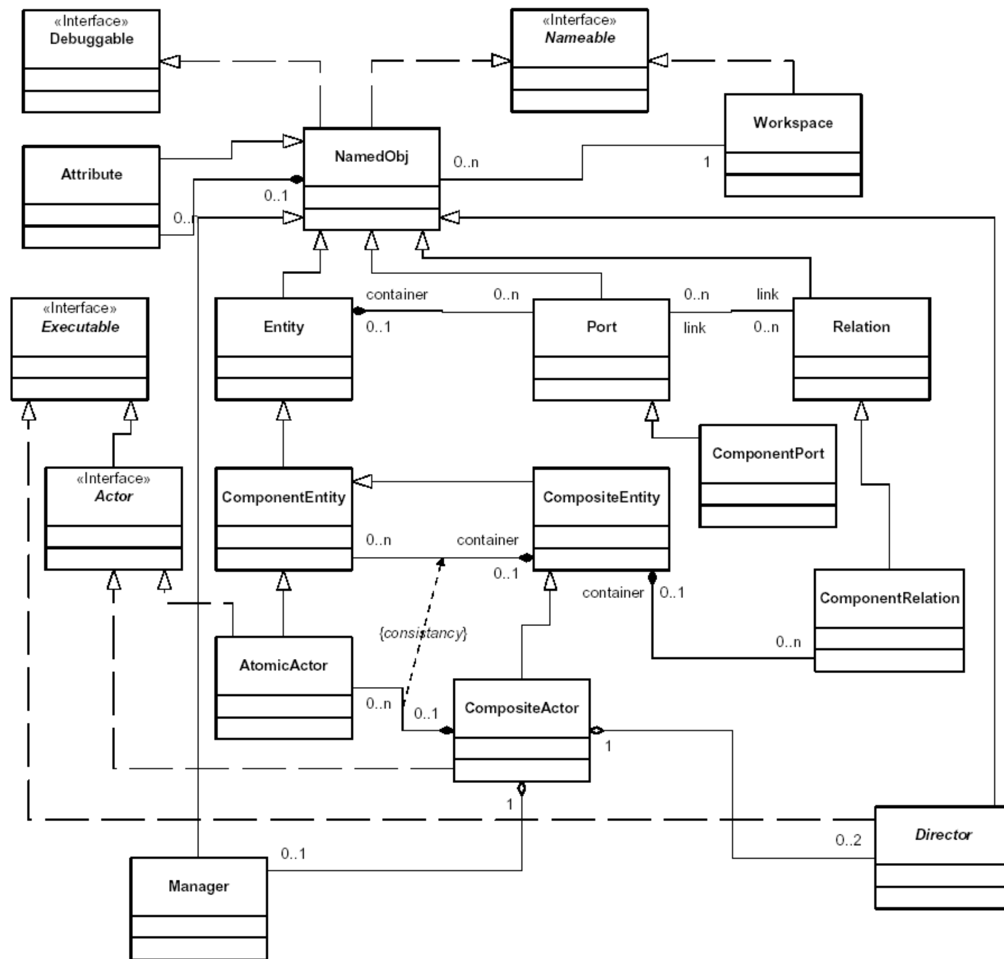


FIGURE 3.11 – Ptolemy II Kernel class diagram (partial) [LEE01]

3.13 Cave

The Cave Project [IND98] is a research initiative aiming to make possible a user-transparent distribution of CAD resources over the World Wide Web. It can be divided in two parts. The first one, a framework of reusable software, available to design automation tool developers, allows an easier way to produce Internet-enabled design tools and model design data. The second one, a web based design environment prototype, validates the framework, and can be used for IC design and education. The original architecture of the Cave Project is based on the distribution of the design resources between client and server sides of the network, as well as on the interfacing of those tools using hyperdocuments. In order to define the design automation tools distribution over the network, the tools are divided in two groups, regarding the level of interaction of the designer with each tool. Interactive tools are attached to hyperdocuments and run inside a web browser on the client side, while non-interactive tools are executed on the server side, according to inputs from the designer on a HTML form.

Group 1 - High level of interaction: this integration architecture is easily understood if related to the white box kind of integration. Belong to this group the tools with intense work of the designer over graphical interfaces, such as schematic editors and layout editors. These tools must be written (or re-written) using platform independent solutions, such as Java programming language, and be attached to a hyperdocument. The execution procedure is described below and is illustrated on Figure 3.12:

when the designer browser requests the tool hyperdocument through its URL, the server sends the hyperdocument with the tool attached to it;

the client receives the application and executes it;

the project data can be stored in the client or in the server storage systems (in the later, it is necessary to open another network connection).

Group 2 - Low level of interaction: this integration architecture is easily understood if related to the black box kind of integration. Belong to this group the tools in which the user interface is based in data providing and analysis, form filling, simple choices over checkboxes and so on. Tools such as electrical simulators, rule checkers and automatic layout generators, which require only passing a circuit description file and some parameters, are typical examples of the low level of interaction tools. These tools run at a server machine and only exchange data with the client, using the Common Gateway Interface or Servlets.

The execution procedure is described below and is illustrated on Figure 3.12:

when the user's browser requests the tool hyperdocument through its URL, the server sends a HTML form, with the input fields related to the parameters required by the tool;

the user fills the form and sends it to the server;

the server starts the program, feeding it with the data provided by the user via form;

after running the program, the server can send the results to the client and/or store locally.

To avoid keeping network connections alive for a long time, the Group 2 integration architecture must provide methods to handle tools with long processing time. These methods may deal with push techniques. So, the server machine has to keep track of the client, while it is processing the data. When the job is finished, it opens a new connection to send the results. To keep track of the client, the server machine opens short time connections, by which it sends the current status of the job and receives acknowledgement from the client.

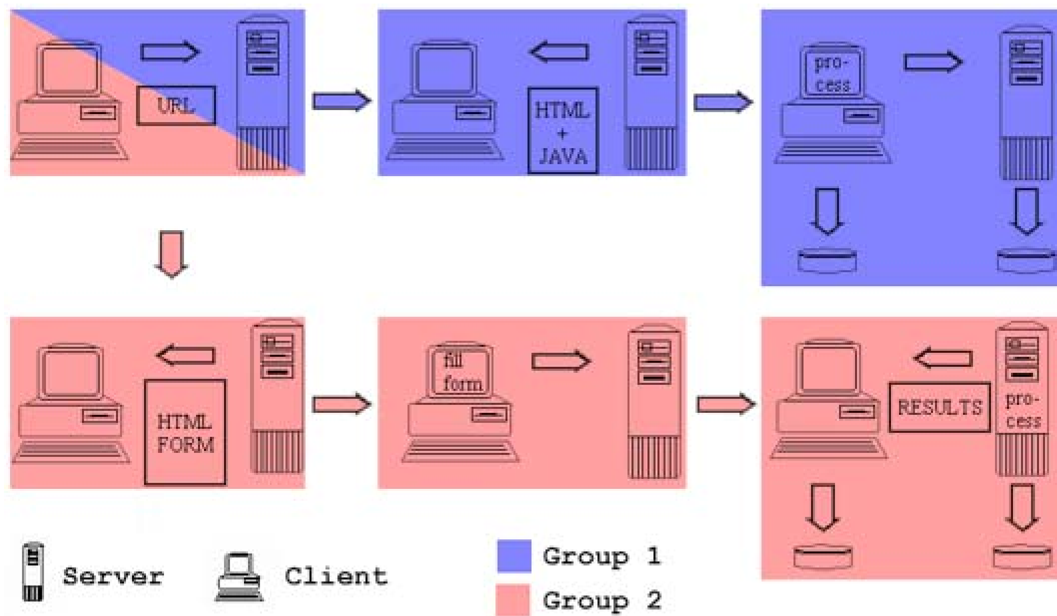


FIGURE 3.12 - Information Flow on Cave System

While achieving valuable results, as shown in [IND98a] and [FRA2000], such architecture was discontinued, as shown in [IND2000], and a new model based on object-oriented concepts is currently the underlying technology on the Cave Framework.

3.14 Comparison of reviewed approaches

A comparison between the reviewed systems is presented in Table 3.1. The marks determine which features are implemented by each system. The comparison terms are the following:

methodology management support – whether the system allows the modeling of tool sequences using workflow concepts;

design data versioning – whether the system supports the storage of multiple versions for a given design data block;

abstraction of CAD resources distribution – whether the system supports the abstraction of the network location of distributed design automation resources;

platform independence – whether the system can be ported to other operating systems without significant changes in its core implementation;

extensible design data modeling constructs – whether the system provides a design data model that can be customized after the system deployment.

TABLE 3.1 – Comparison between CAD systems supporting distributed, multi-user design of integrated systems

Tool	Supports methodology management using workflow techniques	Supports design data versioning	Abstracts the complexity of the distribution of CAD resources over networks	Platform independent	Provides extensible design data modeling constructs
Nelsis	X	X			
Version Server		X			
STAR		X			
Ulysses & Odyssey	X	X			
WELD	X		X	X	
OmniFlow	X		X	X	
ASTAI(R)	X	X	X		
Moscito	X		X	X	
PPP			X	X	
JavaCAD			X	X	
Ptolemy II				X	X
Cave			X	X	

4 Cave2 Foundations

4.1 Introduction

This section covers some of the techniques used to extend the concept of CAD Frameworks presented in Section 2, in order to accomplish the requirements stated in subsection 1.1. The need for those techniques is also covered within this section, as the shortcomings of the original Cave framework are detailed. The outline of the Cave2 architecture, which is the foundation of this thesis work, will be presented at the end of the Section.

4.2 Object Orientation

Object-oriented techniques were proposed in the 1980's within the software engineering and programming language communities [PRS96]. Such techniques advocated on software reuse through the application of information-hiding concepts, so a software system can be developed as a set of self-contained modules interacting among themselves by message passing. Actually, the main concepts which define the object-oriented paradigm - classes and objects, inheritance, encapsulation, polymorphism and dynamic binding – were developed since the late 1960's [HOL94]. However, it was during the last decade of the 20th century that it started to boost general purpose software development productivity [PRS96]. This delay may be understood if the following facts are taken into account:

general-purpose software development tasks were relatively simple at the time of the introduction of the OO paradigm, mainly due to the simplicity of the hardware resources;

software maintenance – which greatly takes advantage on the OO features – wasn't a critical task, once the team who built the software usually was the one to maintain it.

In the second half of the 1980's, a shortage of software developers was reported due to the high demand of application software, mainly because popularization of PCs. In order to increase the productivity, the object-oriented paradigm was taken of the shelf and started to be introduced in the software industry environment. This introduction was done based on several methodologies, proposed by several research groups [PRS94, JAC92, BOO91, RUM91]. Later, the most important methodologies were adapted and put together under the name of Unified Modeling Language (UML), standardized by the Object Management Group (OMG) [KOB99].

Within the CAD Frameworks community, there were two main reasons for the adoption of object-oriented techniques. The first motivation, as stated in [GUP89], is to accelerate the development process of CAD tools and ensure the satisfaction of the users. Using object-oriented techniques, it is easier to do incremental software development, so an early prototype can be provided to the future users and the feedback about it can be obtained during development time. Furthermore, the object oriented CAD systems are easier to maintain and upgrade, because each modification affects only some small, self-contained modules of the system. While the first motivation is also true for every object-oriented system, the second one is particularly related to the complex data models needed by CAD systems. In systems based in simple data models, sometimes the overhead due to the usage of object-oriented techniques exceeds the granted advantages, but when it comes to data models with complex relationships, the object-oriented data modeling and maintenance is much more effective.

In [HEI87], some object-oriented modeling constructs are reviewed regarding their applicability to CAD data:

relationships - object oriented data models are able to express a wide range of relationships between data blocks. Relationships such as IS-A (relating object sub-types to their super-types), COMPONENT-OF (denoting aggregation of objects) and INSTANCE-OF (relating an instance to its type) can be found natively in some object oriented languages. Furthermore, user-defined relationships can be implemented in order to fulfill CAD-related needs, such as VERSION-OF and DERIVED-FROM;

customizable constraints - record-oriented data models usually have very strict rules to ensure consistency, which has as consequence the rejection of any transaction that is not complying with the rules. In CAD data models, such policy may be sometimes inadequate, because in many cases the system should annotate the modifications and notify the users, rather than rejecting the transactions. By using customizable consistency rules embedded in the object model, particular procedures can be used under different conditions, and particular actions can be taken based on the kind of failure or restriction violation;

complex data types - objects can be modeled after complex entities from the application domain. Such entities are well defined by the encapsulation of their state and behavior. In record-oriented data models, such entities would be broken in many parts, relying on aggregation relationships to ensure their integrity, incoming into higher design and maintenance costs on the data model;

abstraction - the abstraction mechanism relies on the information hiding concept: an external view of the objects are provided, but their internal details are not available to the external world. This approach encourages the decomposition of the modeling problems into independent sub-problems. Furthermore, it allows the management of

multiple views of the design data, because some designers may want to deal only with higher abstraction levels while others would have to understand implementation details, and with an object-oriented model the design of such structure is straightforward.

Several EDA research groups started to turn to object orientation since the late 1980's to better organize the development of CAD Frameworks and many advances were reported in the most important design automation conferences. Notable contributions were achieved by the NELSIS group, in the topics of object oriented data models [VAD88], concurrent access to object databases and versioning [WID88].

In methodology support, the advances were achieved by Cadweld group [DAN89], which extended the Ulysses Framework [BUS89] by modeling design tools as objects in a design flow.

Many HDLs were also extended/created to support object-oriented constructs. While the analysis of OO HDLs is outside of the scope of this study, some CAD Frameworks supported such languages, as can be seen in [CHU90], intending to achieve the same level of code reuse and reliability as reported by software developers.

4.2.1 Object Oriented Frameworks

In the modern software engineering domain, a framework is an architecture to build extensible and reusable object-oriented software systems. According to Johnson and Foote [JOH88], a framework is a set of classes that embodies an abstract design for solutions to a family of related problems, and supports reuses at a larger granularity than classes. So a framework defines an efficient, proven software architecture to solve design problems in a particular application domain. The framework defines the global structure of the application, its division in classes and objects, the key responsibilities of each part, how the classes and objects cooperate and how the control sequence is implemented.

It is important to notice that Johnson and Foote, as well as many other theorists in object-oriented design, advocate that the strength of a framework lies on its abstract nature. This means that the architecture of the object-oriented software system should be expressed in terms of abstract classes (classes which cannot generate objects). From this set of abstract classes, several implementations can be derived, by creating concrete subclasses from the abstract ones. So, those classes would inherit the abstract behavior defined in the framework, as so would do the objects instantiated from them.

In [PRE94] such structural aspects are specially highlighted, and special attention is given to the framework usage through specialization in application

building. According to the author, a framework comprehends a set of building blocks, some of them ready to use, some unfinished. The global architecture is pre-defined, and the construction of a new application usually is done by adapting the framework components to specific needs by implementing variables and methods in the classes and subclasses of the framework.

Such scenario is depicted in Figure 4.1, where a framework is shown as a simplified UML class diagram. The framework extension is built by inheritance, and the handling of client requests can be done either by the framework core or its its extensions.

In the design automation field, such frameworks can provide foundations to the development of data models, as well as primitives for the construction of design tools. Differently from the classic concept of CAD Frameworks, such foundations are not executables or libraries, but abstract guidelines, which must be followed during the implementation of the design models and tools, so the interoperability is granted.

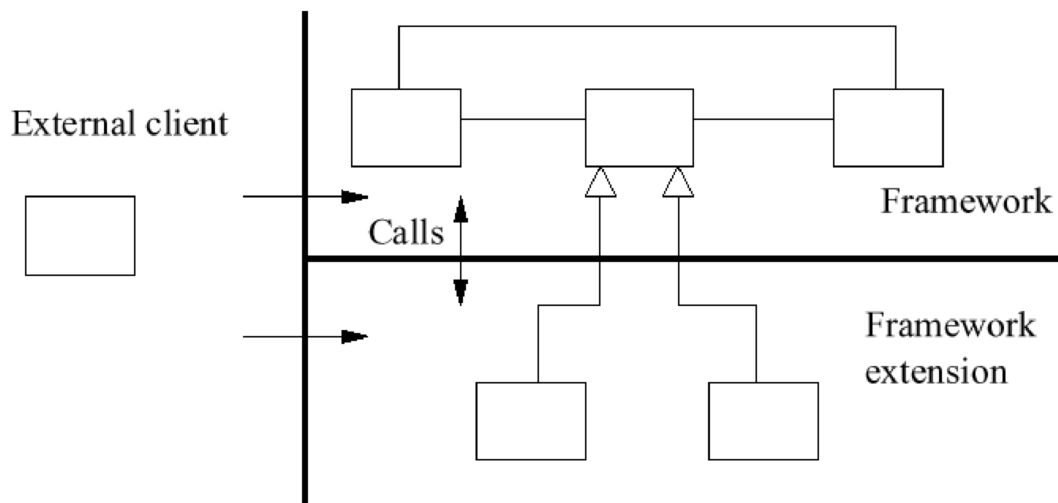


FIGURE 4.1 - Object-Oriented Frameworks

4.2.2 Design Patterns

Many of the architectural solutions used in the construction of a particular object-oriented framework can also be used in another one, even if they are tailored to completely different application domains. According to [GAM95], such solutions should be documented in a proper way, so their usage could be made simpler and their reuse stimulated. A design pattern is then a well-documented solution for a generic problem in software architecture. Such patterns are identified by

names and included in catalogs, so they can be searched and referenced easily during the development process of software systems.

The core of a design pattern includes [GAM95, GOL2002]:

pattern name;

motivation - problems it addresses;

known uses - application scenarios, examples;

structure:

identification of classes and instances involved in the pattern;

roles and responsibilities of each class/instance;

types of collaboration among instances;

expected advantages and costs due to the pattern usage.

We can take as an example the Observer pattern, which is one of the patterns included in the catalog by [GAM95]. It defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is widely used in graphical user interface toolkits which separate the presentational aspects of the user interface from the underlying application data. When the user changes the information in one of the presentations, the others should reflect the changes immediately, and vice versa. This behavior implies that each of the views are dependent on the data object and therefore should be notified of any change in its state. Figure 4.2 depicts such situation.

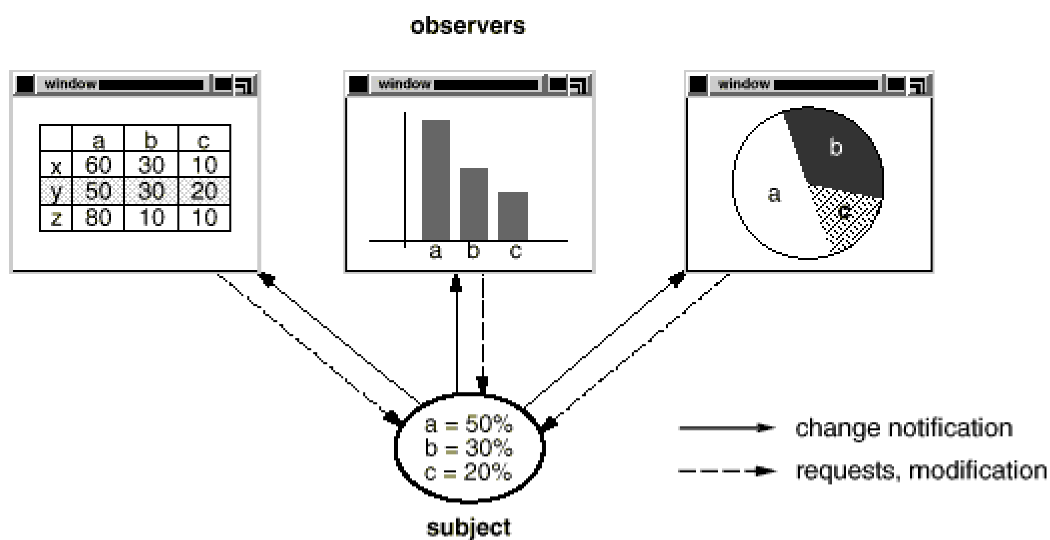


FIGURE 4.2 - Observers and Subject [GAM95]

The design pattern actually describes how to establish the relationships among the participating objects. The key objects in this pattern are subject and observer. A subject may have any number of dependent observers. All observers are notified whenever the subject undergoes a change in state. In response, each observer will query the subject to synchronize its state with the subject's state. Figure 4.3 shows the relationships in an UML class diagram.

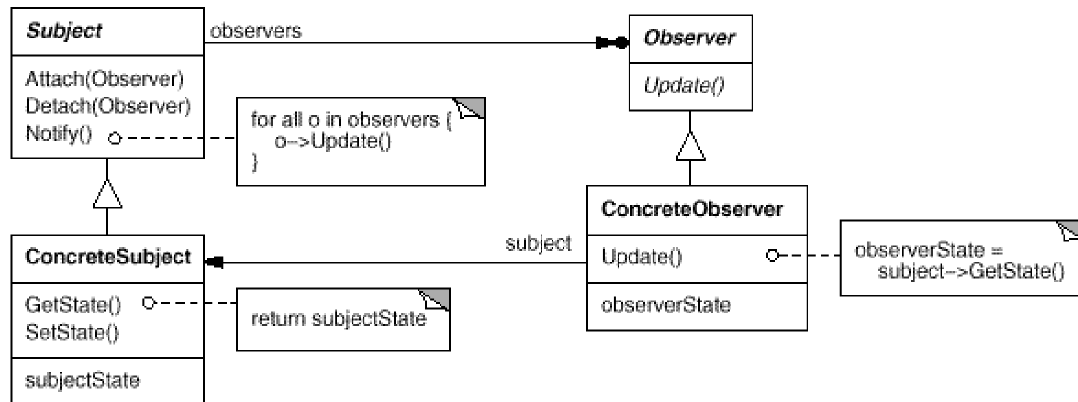


FIGURE 4.3 - UML representation of the Observer design pattern structure [GAM95]

The first and perhaps best-known example of the Observer pattern appears in the Smalltalk Model/View/Controller (MVC), the user interface framework in the Smalltalk environment [KRS88]. That framework advocates for the separation of the software functions: (1) that represent and store data (Model), (2) that allow the visualization of this data by the user (View) and (3) that capture the interaction of the user with both the data and its visualization (Controller). MVC's Model class plays the role of Subject, while View is the base class for observers. Some examples can also be found in CAD Frameworks as well. In [GIR87], the development of a design environment built over the Smalltalk MVC framework is reported. The separation between data and visualization was implemented, so different representation formats of the design data could be presented to the designer. The design environment prototype, named STEM, had views for displaying layout information, spice models, EDIF code, among others, for every cell model in the design library. A controller was also implemented to allow the edition of the cells. Diva [GIG2002], a framework for information visualization developed as support within WELD [NEW99] and Ptolemy II [LEE2001] design environments, also relies on the separation of model and views.

Likewise, many recurrent problems on data modeling within CAD Frameworks can be addressed by the application of design patterns. Within this work, wherever a design pattern is used, it will be referenced by its most common name, according to [GAM95].

4.3 Architectural Evolution - from hyperdocuments to OO

The original architecture of the Cave Project, covered in subsection 3.13, is based on the distribution of the design resources between client and server sides of the network. In order to define the design automation tools distribution over the network, the tools are divided in two groups, regarding the level of interaction of the designer with each tool. Interactive tools are attached to hyperdocuments and run inside a web browser on the client side, while non-interactive tools are executed on the server side, according to inputs from the designer on a web form.

That architecture – described originally in [IND97, IND98a] - has a strong bind with the World Wide Web concepts. It takes advantage on the hyperdocument-centric nature of the WWW, modeling the design flow as a chain of hyperdocuments. Each one of the hyperdocuments embeds the user interface of each of the design tools. So, the hyperdocument links connect the tools in the right order, allowing the user to navigate across the design tasks.

That architecture leads to a file-based design data storage and transfer: the data is stored in web servers together with the tools and the hyperdocuments that create the structure of the design flows. Databases can also be used for design data storage, but as unstructured data. The transmission is done by the WWW standard protocol, the HTTP.

Several advantages - described in [IND97] - are granted by the use of that architecture, such as easy implementation and maintenance of the design environment, platform neutrality, reduced cognitive overhead and possibility of remote use and collaborative design by distributed teams.

In despite of all the advantages granted by the Cave Project original architecture, modifications were necessary in order to fulfill the requirements presented in subsection 3.1. Some of the initial reasons are described below:

- some of the tools didn't fit into the proposed model, because they needed modules to be executed both in the client and server sides of the network, such as Cadena [IND99], C2VHDL [MAY2000], CaveJTAG [IND2000] and WTropic [FRA2000];
- the file-based design storage is not suitable for multi-user distributed designs, and requires intense use of format converters. Furthermore, it is not very efficient on supporting the design navigation;
- the support for collaboration is very hard to implement, since there is no defined data model and the interaction between designer and data are completely done within the tool. So, the implementation of a data-driven collaboration would not be possible, and a GUI-driven collaboration – such as [STE2001] - would probably be the only

option if synchronous collaboration is required. The asynchronous collaboration could be done by using workflow and versioning techniques, but no domain-specific optimizations would be possible;

the use of HTTP protocol, based on URLs, requires fixed references for every resource on the network. A more flexible architecture is needed, in order to allow the dynamic inclusion and exclusion of resources.

In order to address such shortcomings, a new design data model was planned as a substitution for the file-based data storage and transfer. Such data model substitution was already predicted in the Cave Project original proposal [IND98], and could be done over the existing infrastructure. Furthermore, a new architecture for the distribution of design automation resources is needed, in order to accommodate the new modules in a flexible way.

The prototyping methodology [PRS96] was chosen for the design of the new distribution architecture and of the new data model. Due to its iterative nature, the methodology can support the progressive refinement on the system requirements, as well as allow the experimentation of different technological approaches. Intermediate prototypes are not going to be explicitly described in this thesis, as further information about them can be found in [IND2000, BRI2002, SAW2002]. They were used to experiment the new concepts, thus acquiring more knowledge about the overall functionality of the system and about the potential to fulfill its requirements. The Cave2 system architecture was devised over the foundations of those intermediate prototypes, and it is described in the following subsection.

4.4 Cave2 Architecture

In order to create a distributed system architecture, it is important to define first the elements of such system, as well as each element's role in the overall system functionality. The prototyping cycles referenced in the previous subsection lead to the definition of following elements, which are also depicted in Figure 4.4:

data primitives for design representation: set of design data primitives, which are instantiated by the designer while interacting with the design tools. The data model which organize such set of primitives is described in details in Section 5.3;

building blocks for design tools: set of reusable software blocks, aimed to simplify the development of design automation tools, detailed in Section 5.2. Emphasis should be given to the data access modules dealing with the peculiarities of the data primitives mentioned in the previous item;

design data repository: storage resource, aimed to provide persistence to the data generated by the designers. Should be designed to support data-driven collaboration;

system communication channel: infrastructure supporting the data exchange among the modules of the distributed system. Should allow flexible inclusion, exclusion and location of design automation resources;

design cockpit: main user interface, it is responsible for user authentication, communication with the other modules over the communication channel, invocation of design tools and inter-user communication;

service integration interface: each external service is integrated using a pre-defined API. The access to the external service is done through procedure calls, and the parameters passed to the external service are modeled by public fields in the object. Such fields can be used for the automatic creation of the service's user interface by using reflection mechanisms. Combination of several services can be done through a flow editor, such as seen in [CLA2001, LAV2000], modeling task chains parameterized by the designer. The incorporation of such automatic flow editor is outside of the scope of this thesis and is let as future work. Current flow models were defined at source code level.

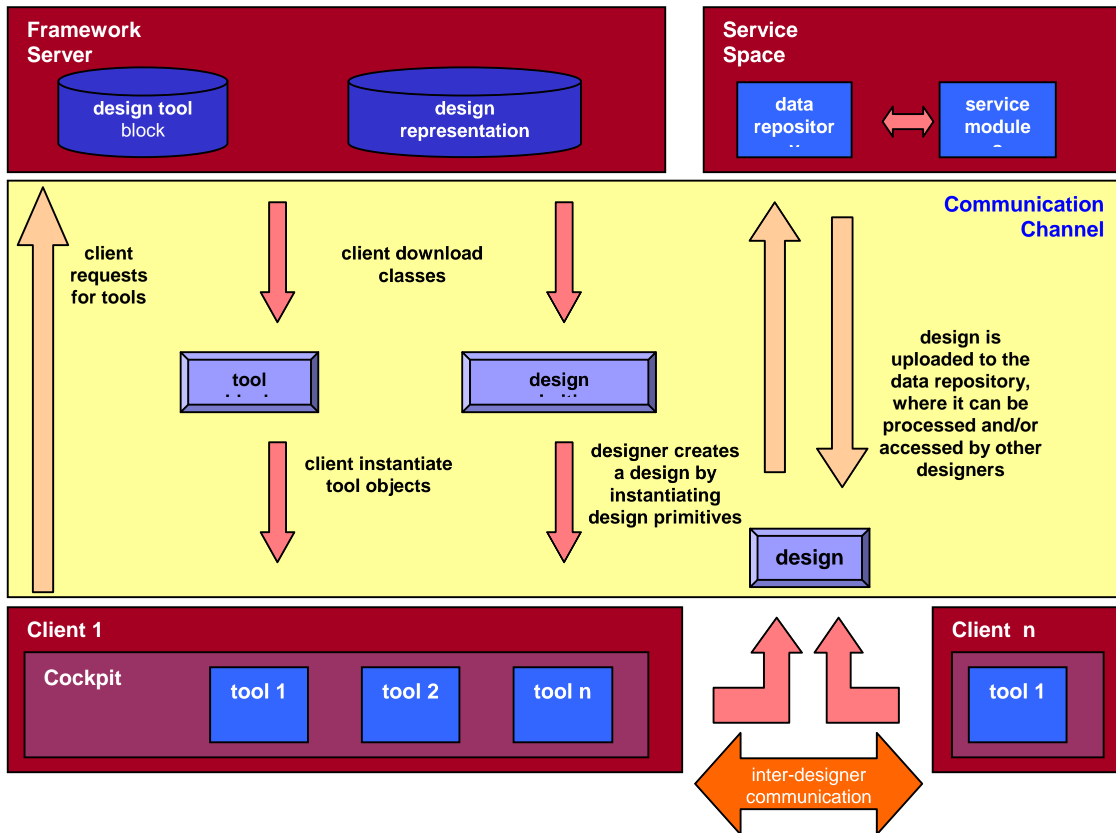


FIGURE 4.4 - Proposed architecture for resource distribution

Many decisions are left to the implementation step, and the detailed communication mechanisms are still to be defined, but the main functionality - the functions expected from a CAD Framework - can be already envisioned through the proposed architecture. The Service Space is responsible for the external tool interface, methodology management and data management. The Cockpit and the Design Tool Block Repository are responsible for tool management and user interfaces, while the Design Representation Primitive Repository is responsible for the data representation.

A typical usage scenario of the proposed architecture is shown in Figures 4.4 and 4.5. The designer downloads (or uses a locally installed copy) of the design cockpit, which searches and connects to an active instance of the Framework Server. It issues a request for tools and services. Such request can be based on a set of attributes, so the Framework Server refers to the client only those tools matching the criteria, otherwise all available tools are referred. Once the client downloads the reference to the desired tool or service, it queries the Framework Server for required or desired additional tool modules, which are dynamically linked to the actual running tools. As the designer creates or updates a design, new instances of design data primitives are requested to the Framework Server. Such set of interconnected instances are the actual design model. Designers can work over a model in a standalone mode, or in collaboration with other designers. In the first case, a design repository service is available from the Service Space (but optional, as the designer may want to store the models locally), and in the second case a collaborative service

is needed on top of the previously mentioned repository, to control the concurrent access to the design data. Other design automation tools and resources can be also available through the Service Space, and should be queried directly via design cockpit or accessed within other tools.

The proposed architecture is based in two core technologies: CAD Frameworks and OO frameworks. The first technology was reviewed in details in Section 2, while the second one was briefly covered in subsection 4.2. It is important here to reinforce the distinct meanings of the term *framework* within the fields of design automation and software engineering. We do so by exemplifying its use on the intended goals of the proposed work:

- in the design automation domain, a Framework is a software infrastructure that enables the interoperation of design tools. Such Frameworks should support the design data exchange among tools, data storage, project management, design edition and visualization, among other functions reviewed in [IND2002]. In this thesis, the complete Cave2 design environment can be considered as an example of such Framework, as it provides most of the services reviewed in Section 2. Its contributions and advantages over the previous approaches in this area are covered in following sections;
- in the software engineering point of view, a framework is an extensible set of inter-related objects. Such objects are usually domain-specific, but many of its relationships are domain-neutral and can be reused from other frameworks (such relationships are know as design patterns, reviewed in subsection 4.2.2). Such frameworks should be extended and used to build applications. In the proposed work, all the data primitives - as well as primitives for user interface and data manipulation - are designed and implemented as an object-oriented framework, described in Section 5.

The combination of the strengths of these two technologies is one of the major contributions of this thesis, as no other approach exploring such possibility is known.

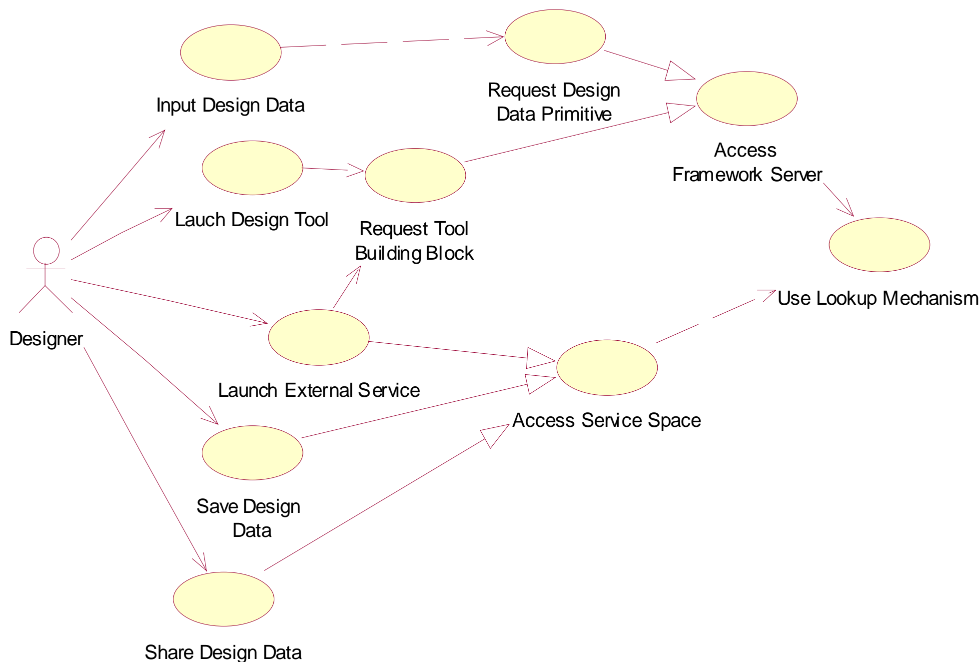


FIGURE 4.5 - UML use-case diagram modeling interaction between user and design environment

4.5 Java-based Approach

Initially an implementation decision, the use of Java technology influenced significantly the development of Cave2. Such influence can be considered harmful, as it denotes a dependency on a given technology and thus makes it harder the deployment to a platform which cannot implement the Java Virtual Machine. However, two points should be considered. The first one regards the portability of the Java Virtual Machine. It has been ported to most of current hardware/software platforms, numerous legacy platforms and it is likely to be also ported to platforms that may appear. Such wide availability of the JVM reduces significantly the odds of having difficulties to widely deploy Cave2.

The second point to be considered recalls the original definition of a CAD Framework [BAR92]: a CAD Framework provides an abstraction layer between the design tools and the underlying operating system. Such layer is responsible for a standard interface for the tools to access the computational infrastructure without using platform-specific resources, thus easing the portability among platforms. If we consider the Java platform a part of such layer, its co-existence with Cave2 is already justified, as it actually provides a significant share of the functionality a CAD Framework is expected to provide to its tools:

- process management - its concurrency library based on the java.lang.Thread class provides a complete support for implementing and managing concurrent processes;
- file and network services - the java.io package provides stream-based facilities for data input and output in such a way that most of the media-related complexities are abstracted from the developer (for instance, there are few differences when reading a file from the local file system or from a remote server);
- user I/O services – user intervention is modeled within an extensive event library, so user-system interactions are captured in a standard way;
- user interface – two user interface libraries are available, AWT and Swing. The first relies on native implementations of GUI elements, while the second has a fully independent implementation, granting a cross-platform look-and-feel to the applications;
- data modeling – Java was designed to implement all the major concepts of object orientation, so it provides modeling constructs which are suitable for the complex CAD data models;
- integration of legacy code – through the Java Native Interface, legacy code written in C/C++ language can be accessed directly by Java applications. Furthermore, its distributed architecture make it simple to encapsulate and integrate legacy applications as black-boxes regardless to their location in a network.

Taking into account the fact that few of the known CAD Framework implementations were able to provide such a transparency layer between the application domain and the operating system because of the complexity of such structure, the decision on relying on Java to do so seems to be appropriate even though it may reduce slightly the Framework deployment potential.

There is one more issue that should be noticed. The most common critic against the Java technology is its performance overhead. In order to be platform-independent, Java code is interpreted during runtime, causing it to run slower than its natively-compiled counterparts. Many techniques have been used to reduce such overhead, such as Just-In-Time compilation, runtime optimizations and integration of native code for critical computation. Thanks to those techniques, in many cases the Java code execution has a comparable performance with native code, while in most of cases the performance is 1.5 to 3 times worse, according to the application domain [WEN2001, BUL2001]. However, in some cases when memory management issues are not critical, Java performance was reported to be even better than native code [MAN98].

5 Framework Core

5.1 Introduction

The core of the Cave2 Framework is the Framework Server. It is the part of the architecture which hosts the object-oriented framework of design representation and tool primitives. As showed in Figure 4.5, the access procedure for the design representation primitives and for the design tool building blocks is the same. The Framework Server is responsible for the storage and access control for all the primitives. The Framework Server provides the primitives to the cockpit as the primitives are needed. When a designer invokes a particular tool, the cockpit asks the Framework Server for the primitive building blocks for that tool. From the primitives, the cockpit creates instances, connect them together and assembles the tool: GUI, data structures, event listeners, communication channels, etc. Once the tool is assembled, the designer can starts to interact with it. As he/she interacts, it creates a design model by instantiating design data primitives requested from the Framework Server.

According to the underlying infrastructure, there is more or less development work to implement this part of the system. A query mechanism is needed in the Framework Server, supported by a naming system to guarantee the existence of an unique index for each primitive. An API should also be defined to allow the proper communication between the Framework Server and the cockpit. The Framework Server must support concurrent access, in order to serve many cockpits simultaneously. We do not expect to provide maintenance on the primitive pools while the Framework Server is running, so a maintenance mode is not planned. The reason is that the base primitives which are already included in the object framework should not change, and the addition of new primitives does not require an explicit control, once they would be not in use before. The naming system should guarantee that the new entries are available through the query system, though.

As the Cave2 implementation relies on Java technology, all primitives can be modeled as Java Classes and a Java Virtual Machine [LIN97] is be provided as the underlying platform for the cockpit. The naming and the query mechanisms available in the JVM can be reused within Cave2, because the JVM follows a concept of name spaces to identify all the Classes according to their URLs, and uses HTTP servers or file system calls to query and retrieve the Classes when needed. Such spaces are identified by the *codebase* parameter of the JVM. The handling of concurrent accesses can also be taken for granted, as file systems and HTTP servers implement such functionality.

Regarding the composition of tools, there must be an assembly procedure definition for each one of the tools. In most of the cases, this composition should be done at code level - for instance, there will be a primitive which will be the responsible for assembling the others, so this primitive will be seen by the cockpit as

the tool itself. In the proposed implementation, we use the possibilities of dynamic Class loading and late binding [GOS96], in order to allow the tool assembly during runtime.

If more than one Framework Server is allowed, a communication API between them should be defined, as well as a policy for resource duplication and consistency check. For the proposed implementation, we decide to use unique Framework Servers. Future research will provide information about the needs for load distribution, multiple servers optimized for specific application domains, local caches for faster network access and redundancy for fault tolerance.

In the following subsections, the two major components of the Cave2 object-oriented framework are detailed. We hereby clarify that the term *framework* is used within the following subsections with the meaning of the object-oriented framework which is hosted by the Framework Server. When referring to the whole Cave2 CAD Framework, we will use its proper name to avoid confusion. Generic references to the CAD Framework concept will usually mention the complete name of the concept as well, always with the first letter capitalized. The terms *subclass*, *superclass* and *extension* refer to the inheritance concept from the object-oriented paradigm. Object-oriented types – Class and Interface names - are presented italicized. Furthermore, throughout the current Section all the references to such OO concepts – Classes and Interfaces - appear with the first letter capitalized.

5.2 Design tool primitives

As stated in [BAR92], a CAD Framework is supposed to provide support for the design tool developers, so new and existing tools can take advantage on the Framework infrastructure. Following such guideline, a part of the proposed CAD Framework is aimed to provide building blocks to make easier the construction and integration of design tools. As a collaboration-enabled CAD Framework, it is also necessary to simplify the inclusion of collaboration support in the integrated tools.

Because of its commitment with object orientation, as well as with Java technologies, the intended implementation will provide only facilities for the development over Java platform. Platform-dependent solutions should be either integrated as services (see Section 6) or integrated in the code level using the Java Native Interface (JNI). The building blocks which are part of the core framework include:

- abstract graphical user interface primitives which provide the support for dynamic linking of the design tool's GUI with the Cave2 cockpit;
- domain-neutral 2D graphical engine, for rendering diagrams, schematics, layouts, etc. It is built on top of the interface for

accessing the data repository and interact with the concurrency control service, but works as well in standalone mode;

domain-neutral text edition module. It is built on top of the interface for accessing the data repository and interact with the concurrency control service, but works as well in standalone mode. This work is outside of the scope of this thesis, but it is being performed concurrently with it [HER2001];

sample modules for querying the design repository, aimed to be used on the development of tools which need to extract particular subsets of the repository data;

parsers and export tools to allow the data exchange in file formats.

Following the guidelines of framework development, the design tool primitives are organized as an extensible set of packages, so addition to the elements mentioned above are possible and encouraged. The foundations to such extensions can be found within the *cave*, *cave.graphic*, *cave.text* and *cave.tool* packages of the framework (see Appendix 3 for detailed description on the framework packages). Some support to the GUI primitives are also available in packages *cave.awt*, *cave.util* and *cave.io*. Many of those facilities had their functionality implemented by Java Foundation Classes (JFC) library after their creation within the Cave packages for tool primitives. In many cases, the Cave primitives were substituted by JFC Classes, but in some cases the Cave primitives are still in use by tools implemented before the introduction of the equivalent functionality in the JFC libraries. The usage of such Classes in newly developed tools was only accepted when the functionality of the Cave implementation presents clear advantage over the JFC counterpart.

The *cave* package is the root of the complete framework. It includes the primitives for assembling the Cave2 cockpit – its GUI and the Classes supporting its connection to the Framework Server and Service Space. The most of the primitives for tool GUI construction are inside of the *cave.graphic* package. The *CaveGUI* Class is the abstract superclass of the Class every tool must implement to describe the procedure for the dynamic assembly of its GUI. Such abstract superclass ensures every new tool which is developed can be dynamically linked to the Cave cockpit – no further compilation or linking is needed, and the tool inclusion can even be done within an executing cockpit. The *CaveGUI* also provides the tool all the necessary functionality for operating in a windowed GUI, because it inherits from the JFC Class *JFrame*. Furthermore, *CaveGUI* also provides a reference to the Cave cockpit and to the Framework Server. The first reference is used to provide communication between the several tools that can be concurrently executing on top of a cockpit, granting the possibility of a common transfer area to implement copy-paste or similar functionality. The second reference is implemented as the communication channel depicted in Figure 4.4. It allows every tool to download design data primitives (described in the next subsection) and to query and contact the Service Space (described in Section 6).

While one can easily integrate a new tool into the Cave2 Framework by extending *CaveGUI*, the effort to build the tool itself is not changed by using that Class. To address such issue, a set of Classes were created to make the development of new tools on top of the Cave2 Framework easier. Such facilities were usually disregarded in many of the CAD Frameworks developed previously, and it is not difficult to imagine that the lack of facilities for tool development was one of the reasons such Frameworks had restricted adoption.

In order to decide which facilities should be implemented, we started the prototyping of typical tool GUIs, such as schematic viewers and editors, layout editors, simulation and synthesis tool interfaces, etc. Details on the implementation of such prototypes can be found in [BRN98, IND2000a, IND2001a, OST2001, BRI2001]. During the first implementation cycles, common functionality among the several tools was found. Such kind of functionality was then factored out from the tools and added to the framework as a reusable module.

An important example of such reusable modules is the *CaveGraphicEditor* Class, which incorporates and uses several other reusable modules to built a superclass for all graphic editors. The elements which are viewed by such editors- visual representation of design objects - should be subclasses of the abstract Class *CaveVisualObject*. By following the extension guidelines, one can easily build a graphic editor which already renders visually a set of objects using a basic rendering engine with user-driven panning, zooming, object resizing and positioning. In Figure 5.1, an UML Class diagram shows partially the structure of the Cave tool primitive packages. A more comprehensive diagram is shown in Appendix 2.

The additional Classes shown in Figure 5.1 can be divided in two groups:

- graphical engine group, which manages the visual objects and the visualization procedure. The Classes *CaveCanvas*, *Universe* and *CaveVisualObjectManager* belong to this group. *CaveVisualObjectManager* is the main interface to the graphical engine. It propagates the requests to the *CaveCanvas*, which manages the *Universe*. The *CaveCanvas* is actually the responsible for the rendering of the visual objects, so it is a GUI component itself - derived from the JFC Class *JPanel* - aggregated to the *CaveGraphicEditor* GUI. The *Universe* is a multidimensional data structure, which organize the visual objects in layers and manages a coordinate space for them. It allows for extensions that take advantage on the coordinate space and layers to provide an optimized access to the visual objects it stores, such as using a layer index or a quad-tree;
- event handling group, which manages the user interaction. The Classes *CaveGraphicEditorEventHandler,* *CavePrimitivePalette,*

SelectMode, CreateMode, GroupMode and *ConnectMode* belong to this group. The first Class handles the user interface components of the *CaveGraphicEditor*: menus, buttons, etc. The second one handles the selection of interaction modes, which are actually handled independently by the other mentioned Classes. For instance, when the designer uses the palette to set the creation mode, the palette activates an instance of CreateMode to handle the events generated by the user over the CaveCanvas, and so forth.

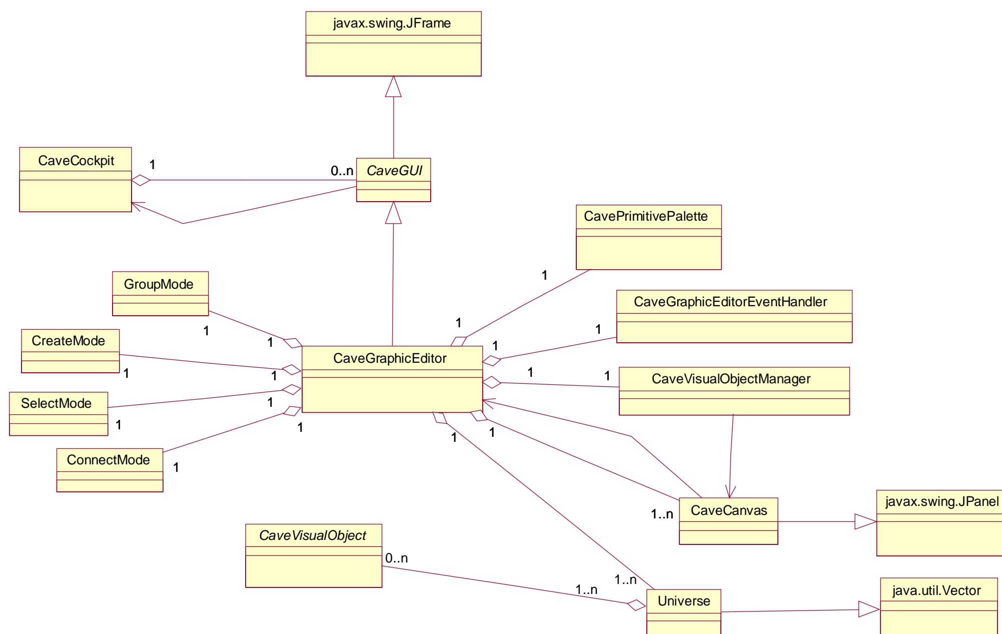


FIGURE 5.1 – UML Class diagram of GUI primitives (partial)

The foundations described above are based in a generic, abstract visual object. In the implementation of the actual tools, such object was extended into concrete Classes implementing actual representations of design objects. Also in this case, a set of common functionality was factored out from the prototype tools and included in the framework. Such structure is depicted in Figure 5.2. It connects to the diagram shown in Figure 5.1 through the *CaveVisualObject* and *Universe* Classes.

The first extension implemented within the framework is the support for metadata. This feature allows for the inclusion of additional information to each visual object – for instance name, comments, author, etc. Such support is also included in abstract Classes. The first concrete Class in the hierarchy of visual representations is the *CaveVisualBox*, which define the shape of the visual object as a rectangle. Such Class is extended to support connection ports (*CaveVisualBoxWithPorts*) and hierarchical compositions (*CaveVisualBlock*). The connection between objects is modeled by the Interface *ConnectionEnabled*, implemented by both *CaveVisualPort* and *CaveVisualConnection*. For more details about each of the Classes and Interfaces, refer to the Appendix 3.

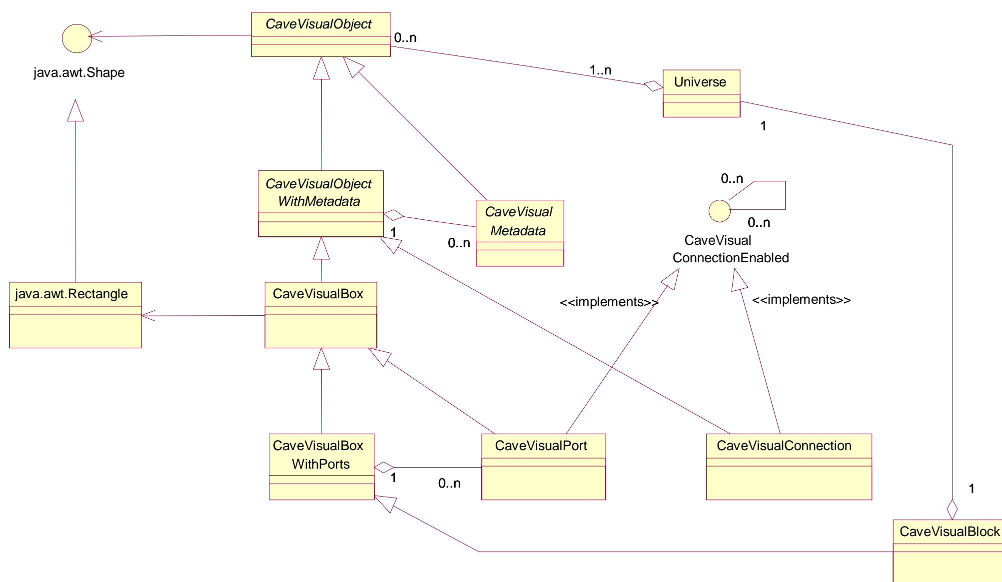


FIGURE 5.2 – UML Class diagram of visual primitives for design representation (partial)

Such foundations were used to build the late prototypes of most of the tools referenced earlier, namely Jase, Jale, Jale3D and Blade. The particular extensions within each one of the tools – the building blocks which are not reusable in other tools – are organized inside the *cave.tools* package, in a subpackage for each tool.

5.3 Design data primitives

One of the purposes of any CAD Framework is to support the designers on creating a design model. Such design model must be represented within the CAD Framework somehow. Such representation – often referred as design data model – is critical to the interoperability between tools – and as we will address later, between designers. Such representation is also critical to the applicability of the CAD Framework: it would only be applied to a design domain if its design data model is able to express the constructs which are peculiar to that particular domain.

Thus, to achieve a maximum of applicability and interoperability, the design data primitives included in the proposed framework are generic, aiming to a minimum common denominator to the variety of modeling constructs used within the EDA field. Initially, we restricted the supported modeling constructs to the following:

modularity – the data model should support the definition of independent, self-contained design modules, so that the concurrent access to different modules can be done transparently. For instance, two designers must be able to work in different pieces of the design

simultaneously, or one designer may like to further develop one block while simulating another. In order to make it possible, the data model should provide well defined primitive objects, with well defined interfaces, so all the design data modules would be derived from those;

hierarchy - the data model should support the description of tree-like hierarchical structures, which are necessary for every type of design relying in *part-whole* decompositions (Figure 5.3). For example, a 4-bit full-adder block can be a composition four single-bit-full-adder blocks, each of them composed by of NAND gates. In [RUB94], the hierarchy is described as a feature present in complex objects, as opposed to primitive ones which contents and functionality can be considered common sense, for instance again the NAND gate;

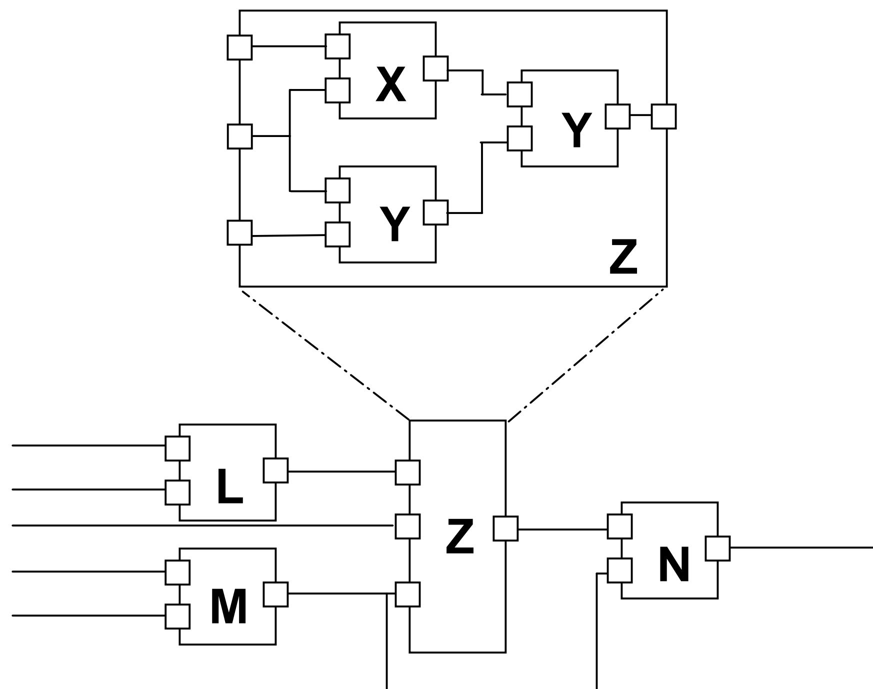


FIGURE 5.3 - Example of hierarchical construct

instances - within a design model, the instances of a particular design data module are considered as new blocks which mimic the structure and behavior of the original one. For example, a module which calculates Hamming distance can be used in many different implementations of a Viterbi decoder algorithm. It makes no sense to replicate it in every implementation, so its description is kept unique, and an instance is used everywhere it is needed. The main advantage of this modeling construct is to avoid data redundancy, so no storage resources are wasted, and the updates are done through a single point. The concept of design libraries also rely on instances, because the actual design data is stored on the library, and only instances of it are used on the design domains. The implementation of the *instance-of*

relationship is often integrated with hierarchy, and sometimes both relationships are considered as one. We differentiate them because their purpose is distinct. While hierarchy is used to define that a particular block is composed of parts, instances are used to denote that a particular block is an exact replication of another, which may be hierarchical or not;

communication - the connection between design modules should be also supported by the data model. A connection represents the possibility of communication between two modules via any medium. The nature of such communication will depend on the abstraction level of the blocks and on design domain specific information. Often, design modules present specific communication structures – ports, pins or similar structures – which should be supported by the data model as well.

A classic data model implementing such constructs is the *5-box* model referenced in [WAG94]. In such model, depicted in Figure 5.4, a cell presents pins and define instances, which also present pins. A cell can be hierarchically composed by instances and the connections between them. Connections can not only connect instance pins among themselves, but to the pins of the container cell.

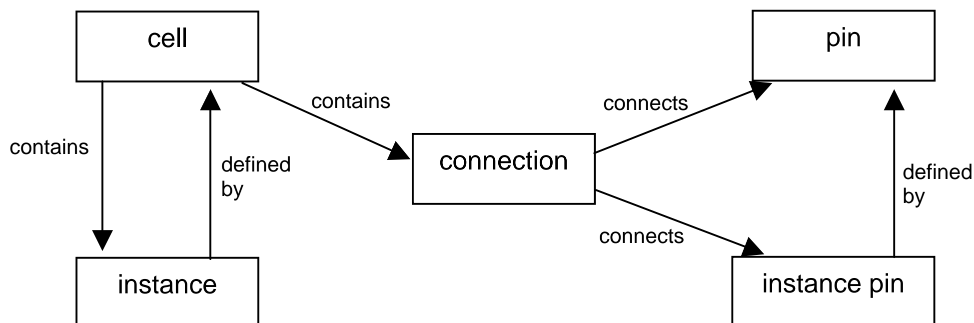


FIGURE 5.4 – 5-box Data Representation Model[WAG94]

As stated in [WAG94], such model is generic in the sense it does not assign any semantics to its elements. Cells, instances, pins and connections are black boxes which can be used to store any domain-specific entity. The data model only manages the relationships between them.

However, in an object-oriented framework such model can be used as an abstract foundation for domain-specific extensions. Our implementation takes advantage on such possibility and provides already some extensions, which were used as case studies on the extensibility of the framework. Such extensions were used in some of the design tool prototypes referenced before, and incorporate the following constructs to the basic model. We introduce one of those extensions below, while

others are introduced when the support for collaboration is introduced in Section 7, and on the case studies shown in Section 8.

The first extension to be introduced to the fundamental data model is aimed to support the inheritance construct. The data model should support the factoring of common attributes of design modules through *is-a* relationships. This modeling construct was introduced as part of the object-oriented paradigm in programming languages [PRS94], and can be very useful on the refinement and specialization of design blocks. Inheritance allows the designer to model extensions – or specializations – of a design block by adding functionality without modifying its specification. The block specialization inherits all the functionality of its parent block, and differentiates from the parent by its own added functionality. Special cases of inheritance allow the substitution of the parent's functionality by overriding them explicitly on the specialization.

For instance, a given functional unit FU1 performs basic logic operations, addition and subtraction, while another functional unit, performing the same operations plus multiplication and division, is called FU2. By using inheritance, the designer can state that FU2 is a FU1 – it inherits all its functionality – and then implement only the remaining operations. Another example of the application of the inheritance relationship is when the designer factors out common functionality from several design blocks, creating a common parent for all of them.

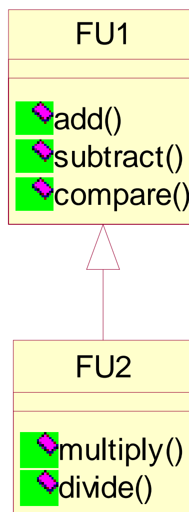


FIGURE 5.5 - Example of inheritance construct

In Figure 5.6, an UML Class diagram is presented, depicting the proposal design data model. Such model already includes the extensions supporting inheritance.

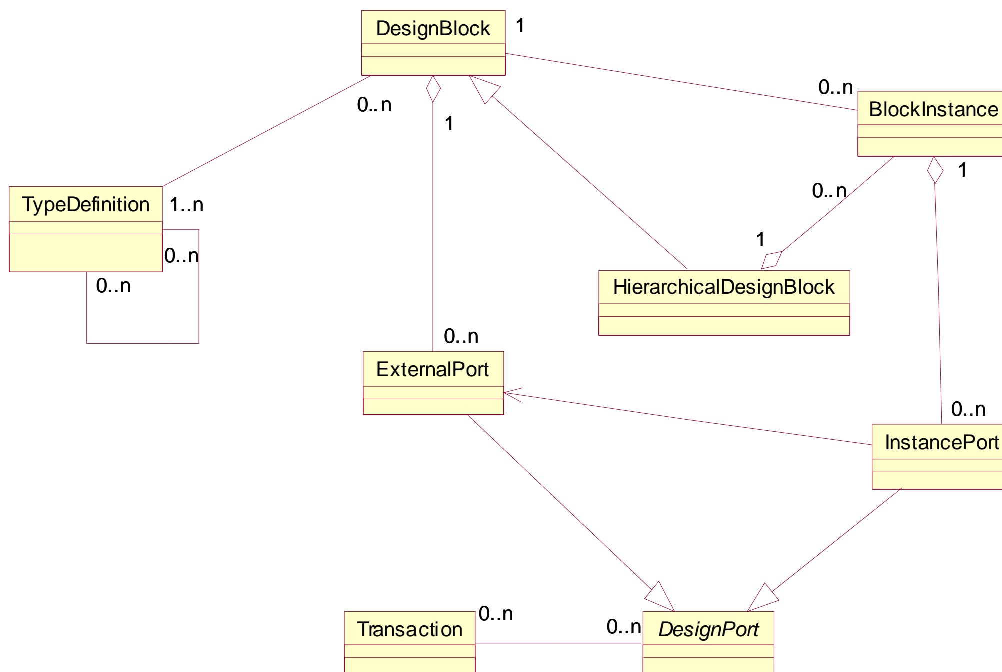


FIGURE 5.6 – UML Class diagram of the proposed design data model

The implementation of the 5-box model was straightforward. The names were changed in order to adapt to a more general nomenclature scheme adopted in the overall framework. Cell became *DesignBlock*, pin became *ExternalPort*, instance became *BlockInstance*, instance pin became *InstancePort* and connection became *Transaction*.

Some additional entities were included. The *HierarchicalDesignBlock* Class was included in order to model the hierarchical composition of blocks using an implementation of the Composite design pattern [GAM95]. In such implementation, we model a hierarchical block as a composition of instances of other blocks, that may or may not hierarchical, allowing for hierarchical compositions of any depth. The concept of instances favors the reusability of blocks over the design description - one block can be instantiated several times and each instances will reflect any update done to the block definition.

In order to allow loose coupling between instances and their respective blocks, we use a Flyweight pattern [GAM95], which advocates for the separation of intrinsic and extrinsic characteristics of an entity. Its definitions on extrinsic and intrinsic attributes matched well with the relationship between the block – which handles the instrinsic attributes - and its instances – each one with its own extrinsic attributes. The concept of the flyweight pool can be seen as a library of design blocks. With the help of the flyweight pool, each hierarchical level can be manageable independently, so that designers can work concurrently on them.

Another additional entity is the abstract *DesignPort* Class, which factors out common functionality from *ExternalPort* and *InstancePort* regarding their communication behavior through transactions. Such approach allows the modeling of transactions that span different hierarchical levels.

The implementation of the inheritance construct, shown by the classes and relationships in the left side of the diagram, was very simple as well. For every *DesignBlock*, a *TypeDefinition* instance is associated. Such class has a self-association to denote the inheritance relationship between types. Notice that the model supports also multiple inheritance, as the multiplicity of the self-association is many to many.

The classes presented in Figure 5.6 are included in the *cave.design* package of the framework.

6 Supporting Distributed Design

6.1 Introduction

This section covers the devised techniques to support the distributed design of integrated systems. As a definition for distributed design, we assume that the following statements can – and probably will – be true:

designers are not co-located but geographically dispersed, so all the interaction between them and the design automation environment – and also among themselves – must be done over a computer network;

design tools are not installed in every computer in the network, so the design automation environment should make them available to remote users in a consistent and efficient manner;

design data is stored in many computers over the network – in some cases redundantly - so the design automation environment should take such situation into account when retrieving and storing data.

Most of the previous approaches reviewed in Section 3 addressed such issues by adapting and applying the client-server model. Such approach defines roles to be played by service providers and service consumers, as well as their communication patterns. Such approach is widely adopted for its simplicity, and it is currently the underlying technology for the Internet network.

However, some of the limitations of the basic client-server model – which are particularly significant in the case of the HTTP protocol, which was used in the original Cave architecture - restricted its applicability on the support for distributed collaborative design. In many cases, the distinction between client and server is not clear as there are design automation resources that may act either as a client or a server. Some examples of such tools developed on top of the Cave Framework are referenced in section 4.3.

Another limitation is its lack of flexibility to freely add and remove servers in a network: (1) clients are by default unaware of server additions, and there is usually no mechanism to give them such awareness; (2) clients are also unaware of server removals, and they notice the removal only when they experience repetitive problems on trying to connect such servers. This happens because the client-server communication is always initiated by the client, and there is no default mechanism for the network elements to inform the clients about changes.

To address such issues, the Cave2 architecture includes a layer between the client and server side. Such layer – often called middleware -

intermediates the communication between network elements, thus blurring the differences between clients and services: every element in the network can be both client and server. To do so, every element declares a communication interface to the middle layer, so it can connect to whatever module which implements a compatible communication interface. Such approach allows a complex task to be distributed in a variety of ways over the nodes of a network, as shown in Figure 6.1. The level of transparency of such distributed processing can be really assessed in the case depicted in Figure 6.2, where a network node *A* issues a service requests to another node *B*, which triggers the execution of sub-tasks in many nodes, and one of them can even be the initial service requester *A*. While *A* believes it is requesting B a service, such service is being done in a distributed way, *B* is only a *Façade* and part of the service is being actually performed by *A* itself.

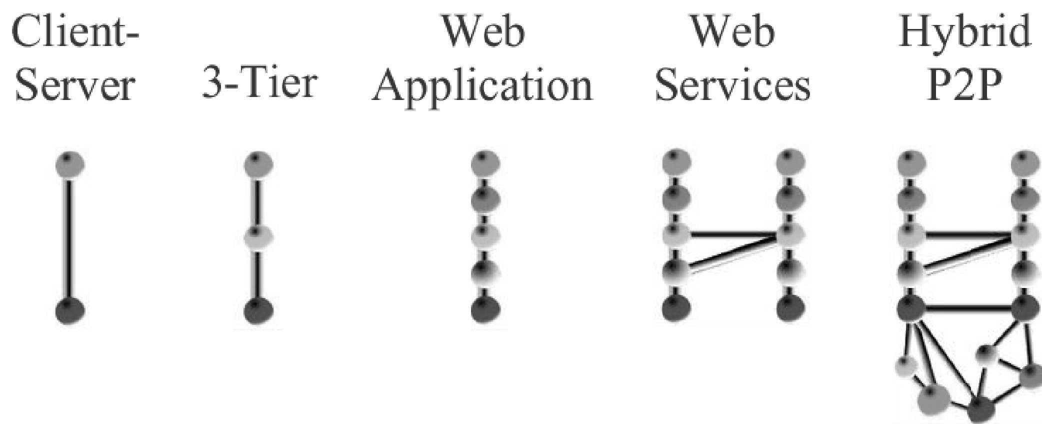


FIGURE 6.1 – Evolution of distributed systems [SHI02]

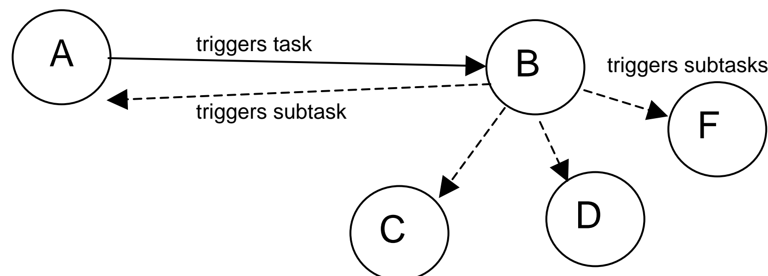


FIGURE 6.2 – Example on task distribution

To address the lack of flexibility to add and remove elements in the network, we use the middle layer to register such events and to notify other elements that may be interested on those events. A query mechanism is also needed, as a network element may need to dynamically request a service provider to perform a particular computation. Such query must be flexible enough to allow several criteria for the search, for instance regarding the service's input parameters, name, underlying platform or even its physical location.

Our approach uses a Jini-based implementation for such middle layer, detailed on the next subsection. Following, we describe the Service Space, which is built on top of the middle layer, and the Framework Server, which is where our object-oriented framework lies.

6.2 Resource Distribution Architecture

Several design decisions were taken in order to implement the underlying architecture for resource distribution in the Cave2 Framework. The first of them regarded the underlying network infrastructure over which the Framework resources should communicate. As TCP/IP networks are nowadays the de facto standard for the intercommunication of computer systems, and as this was the protocol used in the original Cave implementation, this was not much of a choice. By using TCP/IP as underlying infrastructure, the deployment of Framework resources can be done over any Internet-like network.

However, the WWW service - which was the basis for the original Cave implementation - is strongly influenced by the client-server model and, as reviewed previously in this text, present some shortcomings when applied to the distribution of design automation resources. In a distributed CAD Framework, we expect to have access to CAD resources no matter where they are located or which kind of platform they run. We also expect to allow a CAD resource to be included, moved or excluded dynamically within the distributed system, aiming to more flexibility and fault tolerance. Such expectations add to the need for a multi-layered approach, discussed in the previous subsection. There are some middleware solutions which can fulfill those needs, such as CORBA [OBJ2002], Jini [ARO99] and UPnP [MIC2000]. All of them work over TCP/IP networks, and share the common architecture showed on Figure 6.3. In principle, any of them could be used to support our approach.

Our implementation uses Jini, because it is Java-based, there are freely available development tools and most of the facilities for service lookup, discovery and join are already implemented and freely available. Jini also includes a programming model - built over the Java language framework - covering leases, events and transactions. The remote method invocation infrastructure also rely on Java language features, specifically on the JavaRMI package. Such programming model uses local proxies to reference remote objects, so in the application domain all

method calls seem to be local, reducing the overhead usually associated to dealing with remote subsystems. This is a particularly important feature to overcome some of the limitations of the original Cave architecture, as we will describe ahead.

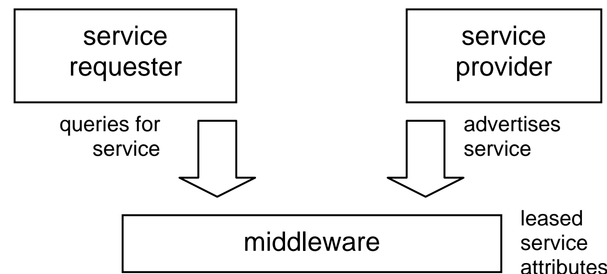


FIGURE 6.3 – Middleware architecture

The procedure a client goes through in order to access a CAD resource is briefly described in the UML sequence diagram in Figure 6.4. First, the CAD resources must advertise the services they provide to the other network elements. This procedure is managed by a middleware entity called lookup server. To locate an available lookup server, the CAD resource uses the *Discovery* protocol, which is particular to the middleware architecture being used. In our case, we use the Jini discovery services: if the network address of a lookup server is known, the CAD resource can establish a direct connection with it; otherwise, an UDP multicast request can be sent, which would be replied by the lookup servers reached by the request.

Once the connection is established, the CAD resource starts negotiating the publishing of its services. Such protocol is called *Join* and requires from the CAD resource some information which is relevant to the lookup server's functionality:

attributes describing the service to be made available, so potential clients can check if the service match their criteria;

service interface, so all clients intending to use such services must comply with;

expected life cycle of the service, so the lookup can publish its service during a limited period of time only, avoiding allocation of unavailable services by the clients.

Once the lookup server receives from the CAD resource such information, it is ready to include it in its registry. It assigns the CAD resource an unique ID – which is used to keep track of the services even if they leave the lookup and join later on – and grants it a lease for the service registration. Such lease is based on the expected life cycle of the service, and the it is part of the lookup functionality to enforce the removal of services that do not renew their leases before the expiration.

In the Jini implementation, the lookup server has a peculiarity which favors our Java-based approach. In such case, the service interface of the CAD resources is stored by the lookup server as a proxy object for that resource. Such models allow several possible implementations, which will be described ahead. Before, we describe the lookup procedure from the client point of view.

The *Discovery* protocol for the client is the same as described for the service providers. Once the connection with the lookup server is established, the client should perform then the service lookup. Such lookup is based on a key which provides an identification of the desired CAD resource. Such identification can be an unique name, when aiming to locate a specific resource, or a generic description, which would locate all available resources matching it. In the second case, the search criteria depends on the service attributes declared by the CAD resources when registering their services during the *Join* protocol. By using Jini, there is the possibility of using a Java object as a key, so even more sophisticated and optimized matching schemes can be implemented.

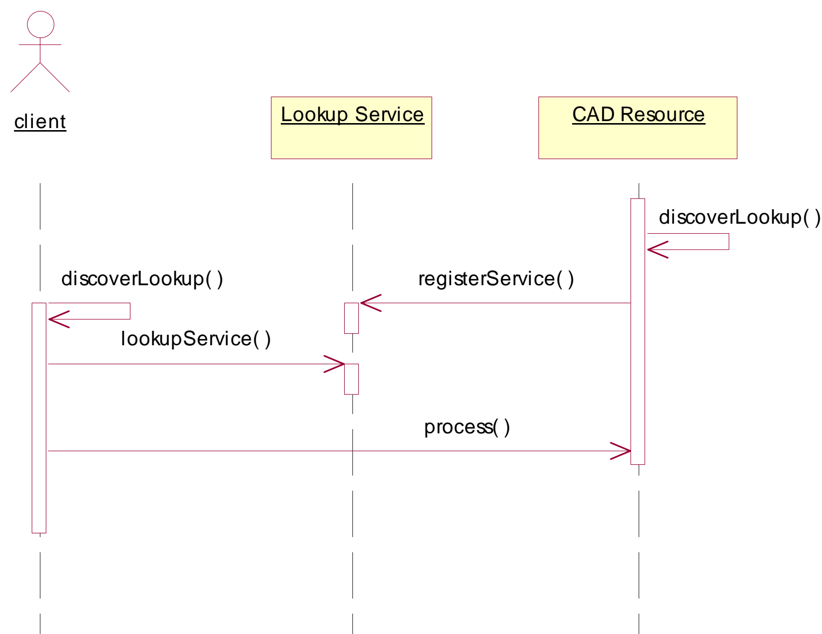


FIGURE 6.4 – Resource lookup protocol

When one or more services match the lookup query submitted by the client, a reference to each of them is returned by the lookup server, so the client can contact them. Along with the reference, a specification of the service interface. Such interface specifies how the client can access the functionality of the service provider, as well as the functionality the service provider expects from the client, if any (as explained earlier in the text, in this approach all the parties are usually both client and server).

As mentioned earlier, in the approach implemented in the Cave2 Framework the clients rely on Java object proxies provided by the Jini lookup server as both a reference and an interface specification for remote CAD resources. Every time a client looks up for services, a proxy object for each one matching the query is returned by the lookup server. Such proxy objects can be used by the clients as local objects, but their interface can hide several possible implementations:

the proxy object encapsulates code which is executed locally in the client machine. The service provider is only responsible for providing the right service implementation, but it does not participate in the service execution;

the proxy object performs some local computation in the client machine, but it contacts the service provider for performing specific tasks (i.e. computational intensive tasks, database access, platform specific tasks or location specific computations such as reading a value from a sensor). The service provider is responsible for providing the service implementation and specific computation resources, but the control flow is executed in the client machine;

the proxy object is only a pipe, and forward all the requests to the service provider. In this case, which is adequate for clients with restricted computational resources, the complete service is provided remotely.

Such possibilities overcome some of the problems found in the original Cave architecture (see subsection 4.3). While the original approach was able to provide transparency on the server-side distribution of CAD resources, the current approach allows for a complete transparency, no matter if resources execute locally, remotely or both. The client only needs to reckon with the service interface.

As a summary, we can state that by relying on the present approach a client can access a CAD resource in a completely transparent way. Its location - and possibly the location of the lookup server - are dynamically obtained during runtime and are transparent for the client developer. The internal functionality of the CAD resource is also hidden, since the client access it through a proxy object, so the service can actually run in the local machine, in the remote machine or in both. The interface of such proxy object - the API calls it supports - are the only information the client requires at development time. In Table 6.1, we review the approaches which supported abstraction of resource distribution from Table 3.1, and this time we differentiate between server-side abstraction and complete abstraction. We include the Cave2 approach in the comparison, in order to show that it overcome most of the limitations of previous approaches in this regard. The WELD system proposed a similar architecture based on a generic middleware solution [CHA98], so it should theoretically support the same level of transparency supported by Cave2. However, due to the lack of reports or availability of an implementation of WELD, such assumptions could not be checked within the frame of this thesis.

TABLE 6.1 – Comparison between CAD systems supporting abstraction of the CAD resource distribution

Tool	Abstracts the server-side distribution of CAD resources	Abstracts completely the distribution of CAD resources
WELD	X	X
OmniFlow	X	
ASTAI(R)	X	
Moscito	X	
PPP	X	
JavaCAD	X	
Cave	X	
Cave2	X	X

6.3 Service Space

The service space is a core component within the architecture presented in subsection 4.4, supporting a variety of tasks:

- integration of external design tools;
- providing runtime environment for internal Framework services, such as repositories, concurrency control, prototyping and authentication services;
- providing service lookup and publish infrastructure, in order to allow dynamic plug-and-play service inclusion.

As described in the previous subsections, many implementation possibilities are available, such as CORBA, SOAP-based webservices, Jini, besides other approaches which are not related to the object-orientation paradigm. Our choice was based on the features of the Jini technology because of the already mentioned reasons.

Built over the communication channel, the Service Space is accessible through the lookup infrastructure. Services plugged into the system use the *Join* interface to notify about their network location and service properties. Clients use the *Lookup* interface to search for the services they intend to use. Such services include external design tools as well as internal services which are part of the CAD Framework, such as the authentication mechanisms, several concurrency control modules (transactions, locking, etc.), the interface to the data repository and the prototyping service. Both external and internal services can be included dynamically, contributing to the scalability of this solution. An overview of such solution is depicted in Figure 6.5.

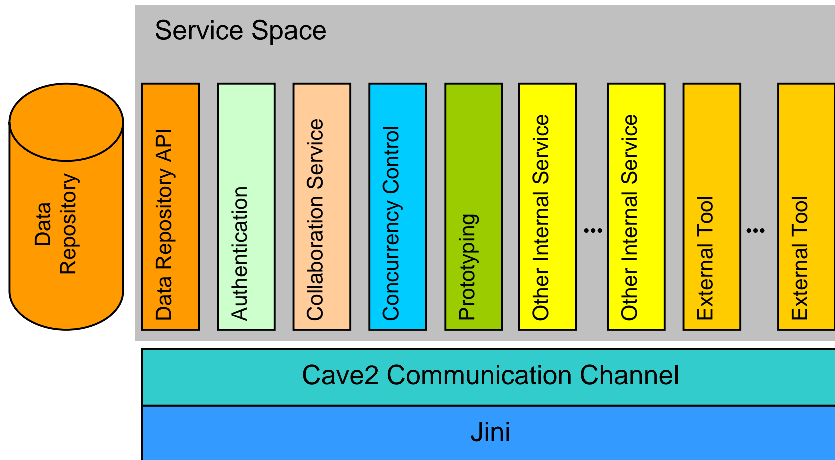


FIGURE 6.5 - Service space architecture

As mentioned in the previous section, the proxy-based approach for the service integration within the Service Space contributes for the resource distribution transparency. Every client dealing with a given service should interact only with a local service proxy, which would propagate data and control instructions to the actual implementation of the service when necessary. This approach is particularly useful for simplifying tool integration and workflow modeling issues. Because of the clear separation between the tool implementation and their access interface defined within the proxy, the tools can be considered "encapsulated" by their service proxies and the tool integration can be done at the proxy level.

Some implementation work was done to validate the possibility of proxy-based tool integration, as we will cover in the following subsections. No specific service was implemented for workflow modeling, as we believe we can rely on currently available solutions such as TRMS/GTLS [KOS2003], OmniFlow and MOSCITO. Each of those tools has its own tool encapsulation and activation strategies, but with little effort they can be adapted to use their workflow modeling and execution engines on top of Cave2's proxy-based encapsulation architecture.

6.3.1 Repository Service

In Section 5, we presented the object-oriented framework which is the foundation of the Cave2 system. In such framework, we could identify design data primitives and design data visual representations (the latter as part of the primitives for design tool development). The separation of visualization and the actual data model follow the motivations reported in [GAM95, KRS88] when detailing the Model-View-Controller model. In our OO framework, we provide primitives for modeling both the model and the view, while the logic for the controller should be provided by the design tools. Such implementation is trivial when dealing with standalone, single-user tools, but can be rather complex when it comes to multi-user, distributed usage. If collaborative usage is needed, further complexity arises. In order

to save tool developers from such complexity, we provide a Repository Service to deal with issue of multi-user distributed design, and a Collaborative Service to handle the possibilities of cooperation among designers. The former is described ahead, while the latter is covered within the next subsection.

The data repository architecture is a key point on the development process of a design environment. In previous implementations of CAD Frameworks, the core of the whole system was the design database. In most of the cases, the database was customized or implemented specifically to support the Framework data model. In [WAG94], the following issues are stated as reasons for that:

- lack of support for modeling, storing and retrieving complex data objects;
- lack of resources to control multiple representations for a same object;
- lack of resources for version and configuration management;
- inadequate integrity verification mechanisms;
- available transaction models are not adequate as units of consistency, restoring and synchronization;
- lack of resources to support collaboration among designers;
- concurrent access control mechanisms - mainly based in locking - are inadequate.

In the tool interoperability arena, common design repositories demanded huge research efforts in the past, specially within the frame of the CAD Framework Initiative (CFI) [CFI94]. Still today, when the concept of a tightly integrated set of design tools is no longer the mainstream, and the design environments are composed by a heterogeneous set of best-of-class tools, this issue is driving the attention from both the technical and strategic sectors of the EDA industry, as seen in the OpenAccess project [CAD2001], a part of the OpenEDA initiative [OPE2002]. That project standardizes a set of functions for communication between the design tools and the design data repository, based on a reference implementation done by Cadence.

Both the CFI and OpenAccess, as well as many of the approaches covered in Section 3, try to overcome the limitations described in [WAG94], but are still built on top of classical database paradigms. Three of those paradigms were analyzed during the development of the Repository Service: relational database management systems (RDBMS), object-oriented database management systems (OODBMS) and shared object spaces. Not only the possibilities for storing design data were considered, but also the support for the Collaborative Service we intended to build. The following subsections highlight the analysis results, and derive from them the proposed approach for the Repository Service.

6.3.1.1 RDBMS

Created in the 1970s by Codd [COD70], the relational data model is being intensively used since then. Most organizations use RDBMSs in their data repositories, creating a huge market and, as consequence, significative technological advances in performance and reliability. According to this model, the data is divided in regular tuples, uniquely identified and grouped together in a relation. A set of relations - usually called tables, to use a simpler metaphor - makes the data repository. Tuples from different relations relate to each other through their unique identification keys.

While this model can be suitable for regular enterprise systems, it presents limitations on supporting a design environment, mainly due to its data modeling and data access strategy.

Its tuple-relation-based approach makes it difficult to model more complex data types. A simple logic schematic would require a complicated relational data repository schema to be properly stored. For some time, when better options were not available, such schemas were actually developed [HAY83]. Nowadays, automated solutions for the creation of relational schemes from complex data types are available, such as the object-relational mapping detailed in the next section, making the task of data modeling easier. However, in such cases the data repository management and maintenance is still very complex.

The data access in relational databases is based on read/write transactions, so all the tools using the data from the repository must work with local copies (Figure 6.6a). To insure the consistency between the copies within the tool and the original data in the repository, concurrency control procedures may be necessary, restricting the collaboration potential specially in the case of multi-user collaboration over the same data block.

6.3.1.2 OODBMS

Object-oriented database management systems have been researched for the last two decades, in order to provide persistence to the objects created by object-oriented programs - e.g. objects that can be used even after the program that created them has been terminated. Currently, several OODBMSs are available - some are commercial products, some are research results - and they can be divided on two classes, regarding the data access strategy.

The first strategy follows the relational model, so the applications work with local copies of the data from the repository (Figure 6.6a). This is mainly because such OODBMSs were built over RDBMSs, or they are actually RDBMSs hidden

behind a object-relational mapping interface - the objects are made persistent by storing its identifications, values and type information as tuples in a relation.

For those OODBMSs, the same advantages and disadvantages found in the relational model are valid. A possible exception may be the performance, which can be lower because of the object-relational mapping overhead.

The second data access strategy is defined by the concept of single instances of data blocks [MUE2000]. It means that the application don't have local copies of the data stored in the repository, but they have references to the actual stored object (Figure 4.9b). When the data is needed by the application, a remote method call is done to the object, which returns the data. This approach has some advantages on implementing synchronous multi-user collaboration, but in other hand creates a strong dependency on the reliability of the connection between the design tool and the repository server.

Regarding the data modeling, both strategies follow the basic concepts of the object-oriented paradigm, which offers rich semantics to model complex data types. However, some of the OODBMSs require special features from the objects that are going to be stored, such as the use of specific superclasses and the explicit declaration of methods that alter the object state [MUE2000, VER2000]. In most of the cases, the implementation of such features is simple and straightforward, but it can restrict the use of some modeling constructs, specially when working with languages without support to multiple inheritance (e.g. Java).

6.3.1.3 Shared Object Spaces

The concept of shared object spaces was introduced by Gelernter [GEL85] in the 1980s, and recently extended by the Jini group from Sun Microsystems [FRE99]. Its goal is to provide persistence services without all the complexity of RDBMSs or OODBMSs. In order to do that, the query engines - which are the main interface between application and repository in RDBMSs and OODBMSs - were substituted by a simpler lookup service. Furthermore, the mechanisms to grant the uniqueness of each data block are not present in the shared object spaces, allowing the storage of multiple copies of the same block, as showed in Figure 6.6c.

The data access strategy in shared object spaces also follows the model of read/write transactions, as in the relational model. Although, it grants the consistency of the data copies in the applications through a update/notify mechanism: every application is notified if the data they have copied from the repository is updated.

Using a different approach, shared object spaces can also be successfully used as a support for a collaborative design environment. While not allowing direct collaboration over a single data block, this approach can be easily used to implement design data versioning. Furthermore, it can be used to support some synchronous collaboration methodologies, where all except one of the members of the collaboration group have read-only access to the data block (i.e. eXtreme Programming [BEK99], Pair Programming [WIL2000]).

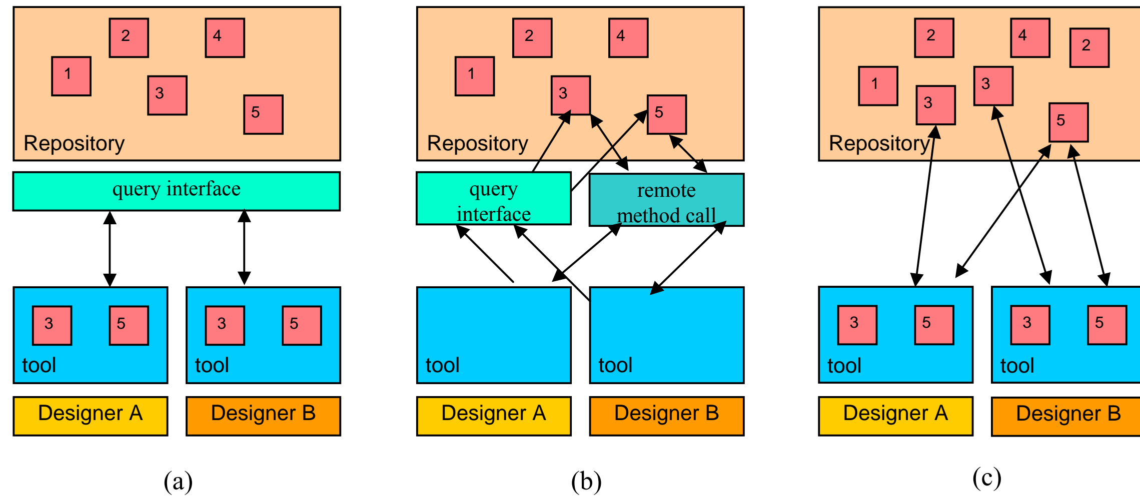


FIGURE 6.6 - Alternatives on Design Data Repository

6.3.1.4 Proposed Approach

Prototypes were implemented using the approaches described in the previous subsections, using subsets of the design data primitives framework:

relational database - behind an object-relational interface (MySQL [WIE2000], using LiDO from LIBeLIS [LIB2002]);

two flavors of object-oriented databases – single instance (Ozone [MUE2000]) and local copies (Versant [VER2000]);

an ObjectSpace implementation (JavaSpaces [FRE99]).

All of them provided the desired persistence capabilities to the design objects. All of them were transactional, so a well known consistency approach was available. The advantages and disadvantages presented in the previous sections were found, but no disadvantage was significant to make impossible the adoption of any of the approaches. Perhaps the high costs for model maintenance in the relational model would put it in disadvantage, but it should be considered feasible anyway.

However, the implementation issues on each case were slightly different, such as the transaction models (some were based on locking, while some also allowed an optimistic approach) and the way to handle remote access (shown in figure 6.6). Such implementation issues motivated the inclusion of an intermediate layer between the database and the design tools. Such intermediate layer – the Repository Service itself – would hide the implementation differences between the several repository models. Several implementation possibilities for such service were studied. One of them was influenced by the classic CAD Framework definition of Data Management Services – recently revisited by the OpenAccess approach referenced earlier - and defined a general API for accessing the repository, which was matched to each implementation-dependent query interface. Such matching procedure was not visible from the tool domain, so tool developers would not need to deal with repository implementation.

While such approach was completely acceptable as a Repository Service – furthermore, it would be rendered perfectly as a remote service within the Server Space because of its well defined API – it still delegates to the tool developers all tasks regarding managing data access sessions and transactions. However, this cannot be considered as a limitation, as session and transaction management are mainly done in the client side in most application domains.

In order to allow the CAD developers to focus only in the tool application domain, we extended the API-based approach by introducing an architecture inspired in the transparent persistency concept. This approach is also explored within the Java Data Objects architecture [ROO2003]. According to this technique, the persistence source should be hidden as much as possible, so the client can have the impression that it is dealing with regular in-memory objects, and not with database records. Our final implementation of the Repository Service uses such concept to create a new approach for design databases.

The proposed extension, included in the core of the implemented object-oriented framework, allows for the direct management of the data objects through their own API - for instance, calling a method *tempblock.addPort(new CaveVisualPort())* - instead of using a database API to do that - for example *insert into PORT (portkey, blockkey) values ('64', '12').*

The inclusion of the transparent persistence features into the Repository Service - which is a Service Space-compliant service - required specific implementation strategies. To follow the guidelines for resource distribution defined within the Service Space, the proposed Repository Service relies on proxy objects. Such objects are used as pipelines, routing the local method calls to their respective counterparts in the Repository Service. To grant the consistency between the repository and the user interface, their interaction follows the Observer pattern [GAM95].

The access to the Repository Service follows exactly the procedure described in subsection 6.2. For each design tool using the repository, a service proxy is loaded from the Service Space. However, such service proxy is not used directly as the complete interface to the repository. It plays its role only in the creation, removal and location of design objects. All other operations are handled by object proxies playing the role of each individual repository object. The Observer pattern actually keeps the consistency between those proxies and their respective visualizations. The consistency between the proxy and the repository object is done transparently by the Repository Service.

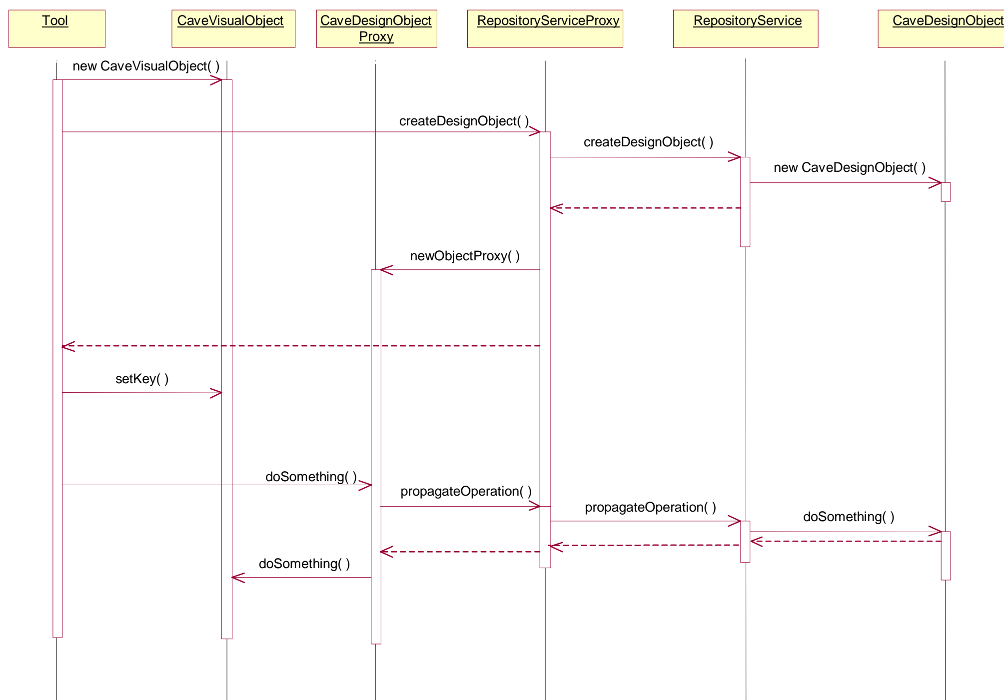


FIGURE 6.7 – Repository ServiceUML Sequence Diagram

In Figure 6.7, the usage procedure of the Repository Service is exemplified. The example assumes that the service was already looked up, contacted, and a service proxy has already been loaded. When a visual object is created by the tool, it should also be created within the Repository Service. This role is played by the service proxy, which creates the respective design object entry in the repository and creates a local proxy for the remote design object. Every further operation performed through that visual object will then be delegated to the design object proxy, which will notify both its remote counterpart and the visual object itself about the recently performed operation (denoted in the figure as a call to the method *doSomething*).

From the point of view of the tool developer, most of the functionality of the repository is hidden. The tool must only obtain one proxy per visual object, and deal with the proxies as if it would be dealing with the visual objects. This simplifies

significantly the development, because a tool can be first developed standalone, dealing only with its local visual objects, and then further extended to use the Repository Service only by changing the method calls from the visual objects to the design object proxies.

To make this possible, most of the complexity of the object storage is dealt by the framework. So, the following issues must be addressed:

class matching - the matching between visual objects and design objects - i. e. when a *CaveDesignBlock* is created, which design object should be included in the database;

creation, removal and query of design objects in the repository – depending on the underlying database, the procedures will be different;

creation of design object proxies and the implementation of the communication between the proxies and their respective design objects in the repository – several implementation alternatives for the communication are possible;

method matching - the matching between operations in visual objects and design objects;

key assignment – every design object must have an identification key, so other objects - such as their proxies - can access them for update notification purposes;

offline operation - the proper operation should be possible even in case the communication channel between the service proxy and the Repository Service is not available.

The first issue is relatively simple to solve, as in most cases we can expect a 1-to-1 relationship between visual objects and design objects: every time a visual object representing a port is created, a design object representing a port will be created, and so on. In the current implementation, the relationships between the classes that model the pairs visual-design are handled by a specific object within the Repository Service. The matching procedure is currently hardcoded within that object. To allow more flexibility, such relationships could be defined in an external file loaded dynamically every time a match is needed, and future implementations should provide such facility – the changes are minimal, and not visible from outside the matching object.

To handle the creation, removal and retrieval of design objects from the underlying database, implementation-dependent procedures are needed. To hide such implementation-dependent procedures, we divided the server component of the Repository Service in two parts and defined a minimal communication interface. The first part would handle then the interface with the service proxy, while the second part would match such interface into repository-specific calls. In the case of transactional

repositories, the boundaries of the transactions should also be handled, as this is not explicitly done by the design tools.

The creation of design object proxies and the communication between them and their respective counterparts in the design repository can be done in a variety of ways. The proxies can be generic enough so that they just forward to the Repository Service all the events they receive from the user interface, or they can encapsulate intelligence so that they can check the semantics of those events, thus forwarding to the Repository Service only the valid ones. The first strategy can be used for every kind of user interface and visual object, while the second one must be based on proxies which are particularly tailored to a given type of tool and visual object. In other words, the creation of semantics-aware proxy involves an instantiation of a type, requiring that a class defining the design object proxy interface to be created in advance for each class of design object.

The proposed implementation includes generic proxy objects, which can provide capture events from all design visualization primitives of the OO framework, as well as their possible extensions. Once captured, those events can be forwarded to the remote Repository Service according to the availability of the underlying communication channel. As mentioned earlier, such generic proxies do not perform any analysis regarding the semantic of the captured events. In many cases, however, a more detailed analysis of the semantics is necessary in order to optimize the communication: semantically invalid events are not propagated to the remote server, and semantically correct events can be performed to the visualization at the same time it is being performed to the remote design objects.

In order to perform such semantic analysis, proxy objects which are specific to a given visual object are necessary. Each design object proxy should implement the same methods implemented by its design object counterpart, allowing for individual behavior to be executed within each method call. In the current implementation, such proxy classes are coded manually in a tedious process, but further automation can be provided in a procedure similar to the enhancement process described in [ROO2003]. In any case, all such proxies inherit the communication mechanisms implemented by the generic proxy classes mentioned above, so the specific methods can handle only the domain-specific semantic analysis and reuse from the superclasses all generic procedures that handle the communication with the remote repository. In Appendix 2, a class diagram depicts the class hierarchy of the proxy objects within the Repository Service.

The communication between proxies and the Repository Service can also have several implementation possibilities, such as dedicated socket connections, remote method invocation and distributed events. All three of them are supported by the Service Space definition. The use of sockets, however, would require the complete communication protocol to be described from scratch. The use of RMI simplifies the development, as it provides a higher level interface as a foundation to the communication protocol. However, the use of RMI communication between each design object proxy and its design object counterpart may be too costly, as it would

require a dedicated RMI connection per proxy, and hundreds of proxies may be concurrently in use by a given tool. To optimize such procedure, we combined the RMI approach with a distributed events approach. The proposed implementation is built on top of a single RMI connection between the service proxy and the repository. Through this unique RMI connection, all the communication between the tool and the repository is implemented as a series of events.

Before we cover the issues of method matching, key assignment and offline operation, let's take an overview on the functionality of the Repository Service based on the definitions we already have. Such functionality is depicted in Figure 6.8, where we can see also its published service interface. As mentioned before, this is the minimum interface for performing its function. Further extensions can implement a more complex interface, perhaps externalizing the transaction control or performance-related options.

As depicted in Figure 6.8, the minimum interface comprehends only four methods: *create*, *delete*, *retrieve* and *trigger*. Their actual implemented names, parameters and return values are shown in Appendix 3. The first three methods are clearly related to the creation, exclusion and retrieval of design objects from the repository. Such calls are explicitly done by the design tool and pipelined to the Repository Service by the service proxy using RMI. The retrieval will take advantage on the transactional infrastructure of the underlying data repository, when available, but this is transparent to the service users.

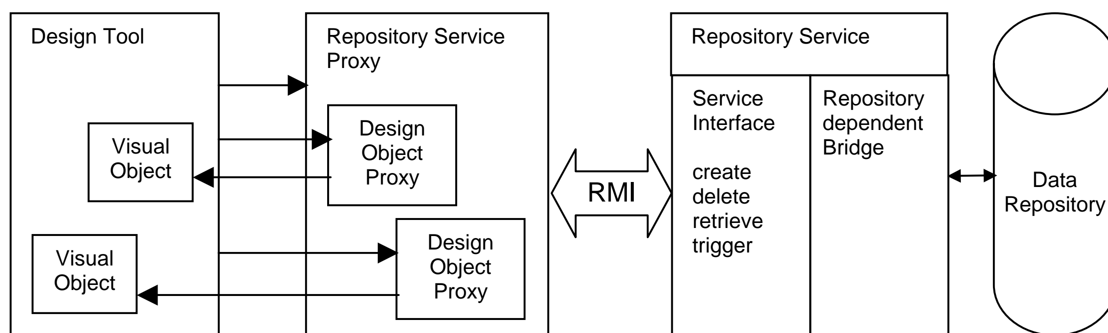


FIGURE 6.8 – Overview of the Repository Service implementation

The updates on existing design objects are handled by the design object proxies. Such updates are triggered by the user interface and propagated to the proxies as method calls, as mentioned early in this subsection. When its methods are called, the proxy object instantiates an event object, which encapsulates its identification key, the name of the called method and passed parameters. The actual parameters are only included in the event object when they are instances of primitive types, such as strings of characters or numbers. If a visual object is passed as a parameter, the proxy passes its key instead, as the serialization of the actual object would be too costly and would incur on further consistency problems.

The event object is passed to the service proxy, which uses the RMI connection to call the *trigger* method from the remote service. The event object is passed as a parameter, so the Repository Service can locate the actual design object within its repository and call the referenced method with the provided parameters.

Our implementation of the event objects is built on top of Jini remote event model. It follows the basic event model introduced in Java 1.1. Such model defines that an event consumer must register with every event producer it intends to keep track. The event producers should handle such registration procedure and notify all register consumers in the case of a state change. A producer can produce events associated to different state changes, so different types of events can be differentiated by the event ID attribute and by the actual class of the event object. Further complexity should be handled when the events are required to be sent via network. The Jini remote event model uses Java RMI to do so, and already implements facilities to assign sequence numbers and overcome network failure. The former is needed to grant the correct order in the event processing by the consumer, while the latter – based on the concept of exceptions – allows the handling of lost events.

Now that we have covered in further detail the way operations are propagated from the visual representation to the repository service, let’s come back to the issues that are still opened: method matching, key assignment and offline operation.

In order to match method names of visual objects and design objects, we use the simplest possible approach by giving the same name to the related methods in both visual and design domains. During the current implementation such practice was sufficient, as we had a 1-to-1 relationship, but if in a particular case this can not be guaranteed – such as when one method in the visual object must be matched by more than one method in the related design object (e.g. a method *removeAllPorts* is available in a visual object, but must be mapped to many calls to *removePort* in the design object) - a mapping scheme in an external object should be implemented (such as the case of class matching described earlier).

In order to match methods by name, we use the features of reflection and dynamic binding from Java. With reflection, we can check during runtime whether a given design object has a method with the name and arguments specified within the event object by the design tool through the proxy object. If negative, an exception is thrown, so the proxy object aborts the operation and performs no change in the visual object. If the method is available, the binding is done and the method is called.

Regarding the issue on identification key assignment, the major problem is to grant its global uniqueness. If only one Repository Service is available in the network, this is a trivial task as an incremental key can be used. If more than one Repository Service is allowed - and we have no reasons to ignore such possibility - a mechanism is needed to grant the uniqueness of the keys issued by the several

repositories. This is actually a classic problem in distributed systems, and most approaches fall into variations of the following mechanisms:

- a centralized entity external to the repositories manage the key assignment procedure for all repositories. It has a trivial implementation, but introduces a single point of failure because the whole system cannot create new objects if such centralized entity is unavailable. Furthermore, such centralized entity contributes negatively to the system scalability;
- a complex key can be used, so several entities can issue keys. In the complex key, the issuing entity is also identified, so the uniqueness problem is reduced to the local repository;
- a key with an extremely high number of possibilities can be used, so several entities can assign new keys by generating it randomly. While the uniqueness is not guaranteed, the possibilities of two equal keys can be made very small.

We adopted the second approach, so each repository can issue keys which are locally unique, and a repository identification is added to the key in order to grant its global uniqueness. However, this can be further developed in the framework extensions. An explicit extension point was included, through the abstract class *CaveID*. Its only defined method – *equals* – allows any possible implementation, granted that the equality of two IDs can be assessed. An UML class diagram in Figure 6.9 depicts such approach.

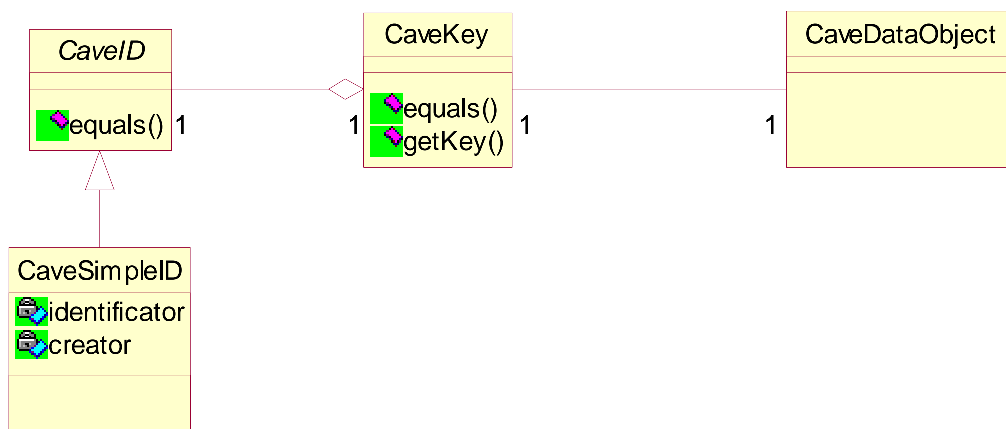


FIGURE 6.9 – Design data identification keys

In order to allow for offline mode - when the connection between the Repository Service and its service proxy is interrupted - we added the following functionality to the service proxy:

assignment of temporary keys - during the period the service proxy is offline, it is allowed to issue identification keys, so new objects can be created without the intervention of the Repository Service. Such keys are temporary and are substituted as soon as the connection is reestablished;

pooling of events - all the outgoing events are queued, so they can be sent to the Repository Service as soon as the connection is reestablished.

Such infrastructure can handle both the cases when the designer needs to be disconnected from the network and the cases of network or repository failure. The issues regarding the synchronization of multi-user data access are covered within the next Section.

As a summary, the proposed approach for the Repository Service provided to design tool developers a much higher level of abstraction of the design tool repository by hiding completely the implementation details. Instead of query languages or transaction management, tool designers can focus on the actual design data model, because all updates are done through regular method calls, which are transparently encapsulated into events sent to the remote service.

6.3.2 Collaboration Service

The Repository Service described in the previous subsection covered the proposed infrastructure to support the transparent distribution of the design data storage. However, for the sake of simplicity such infrastructure was presented in a way that deliberately ignored the possibility of concurrent access to a repository. During its design and development, however, such possibility was always considered and played an important role on the design decisions that were taken. This is because the Repository Service was designed to be extended by the Collaboration Service, which would deal with all issues related to concurrent access to the design data.

In the Repository Service, the Observer design pattern was used to implement the consistency between one model and multiple views. The bigger picture shows us, however, that we have to deal with the possibility of multiple views running in different nodes of a network. Essentially, this situation can also be handled using the same pattern, but the implementation of the communication between subject and observers should handle the distributed nature of the problem.

The Collaboration Service is an extension of the Repository Service in the sense that it has an inheritance relationship. All Collaborative Services are in fact Repository Services with some extra functionality, so the behavior we described in the previous section also applies for the Collaborative Service. It also relies on object proxies, which use the service proxy to send events to the remote data repository,

which is encapsulated by the Collaboration Service itself. The main difference between them is the fact that the Collaboration Service must deal with conflict resolution. While the service proxy of the Repository Service could always create/remove objects and trigger events, some of the operations received by the service proxy of the Collaboration Service must be analyzed before they can be executed or aborted. Such cases are domain-specific and are going to be discussed in the next Section.

6.3.3 Authentication Service

The authentication service is responsible for the identification of users and groups, as well as defining access permissions to all Service Space services. In the current implementation of Cave2, a simple authentication based in usernames and passwords was implemented. Access permissions are only managed within the Repository Service. Access lists are managed within projects, so each user will have specific access permissions within every project he/she is currently taking part. A more detailed description on this topic can be found in [SAW2002].

6.3.4 Prototyping Service

Aiming to support the incremental implementation of integrated systems, a Prototyping Service was included in the Service Space as an internal Framework service. Such service would rely on reconfigurable hardware modules, which can be looked up and leased as a prototyping platform for a designer.

Such service aims to to reduce the integration overhead of reconfigurable hardware modules and the design environment. We propose to reduce such overhead by raising the level of abstraction of the integration architecture, allowing the communication to be done via message passing, as proposed in the object-oriented paradigm. By using this approach, each reconfigurable hardware module could be seen by the rest of the Framework system as an object . Thus, it should be reconfigured and used through method calls. This would make a significant difference for the system designer, which would abstract the internal details of the reconfigurable module - a typical result of the encapsulation feature of object-oriented systems - and would design the whole system communication in the API level. In such approach, all the subsystems depending on the reconfigurable hardware module would call a configuration method to set up the desired functionality, and then call methods to pass the data to be processed and receive the results.

By relying in such infrastructure, the designer can have a simpler path from the high level design specification to its implementation. A typical design scenario where the prototyping service could be useful starts by a designer – or group of designers – instantiating design primitives to model a functional description of a

given system. Once that specification fulfills the functional requirements, it should be submit to successive synthesis steps in order to be implemented as a physical entity. Reconfigurable platforms are being used as an intermediate stage within such process, allowing system designers to verify the correctness of their designs prior to the final implementation. Our approach could provide a simpler way to integrate the functional specification with the prototyping platform, in such a way that they can inter-operate. This would allow a mixture of simulation and emulation in the functional level, because one could synthesize and implement part of the functional specification in the reconfigurable hardware and still be able to perform the functional simulation, as the rest of the specification would communicate with the prototype in the same way it did before with the functional description.

In Figure 6.10, an UML sequence diagram describes the functionality of the Prototyping Service. The procedure starts by the functional modeling by the designer, which is done by instanciating tools and design primitives from the Framework Server, and by storing them in the data repository. Once a functional model of the target system is ready, the designer starts with the prototyping step. In such step, the blocks of the functional model should be synthesized in order to be implemented in a reconfigurable platform. The synthesis tools are technology dependent, so the designer should integrate the tools supporting the desired type of synthesis as external services. Once a particular block of the functional specification is synthesized, it is included in the configuration bank.

The initialization of the Prototyping Service is also shown in Figure 6.10. For every reconfigurable hardware module which is available for prototyping, a backend module and a reconfigurable hardware service (RHS) are instantiated. The backend is responsible for handling the platform-dependent features of the reconfigurable hardware module, thus providing a common interface to the rest of the system. Typical functions performed by the backend include the configuration of the platform and the access to its memory modules. The RHS provides the actual encapsulation of the reconfigurable hardware module, providing a set of methods to access its functionality. The RHS is also responsible for the registration within the Service Space.

Once the RHS are active and there are configurations uploaded into the configuration bank, the prototypation service can be started. The designer substitutes the functional model block which has been prototyped with a proxy object provided by the prototyping service. Such proxy has the same interface as the prototyped block, so the rest of the functional model should operate properly without any updates.

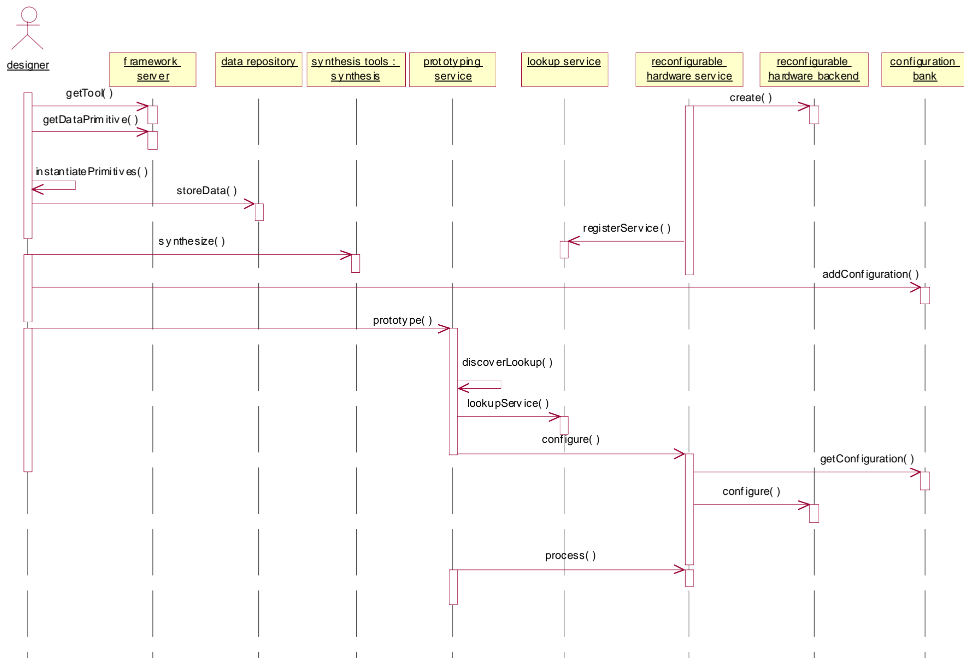


FIGURE 6.10 – UML sequence diagram for the prototyping service

When the functional simulation is started, the proxy object would communicate with the prototyping service, which will then lookup for a RHS and assign it to the proxy object. The RHS will then receive the configuration information from the proxy, and will be able to process in hardware all the functionality expected from the prototyped object.

Such approach would allow an implementation to be validated together with the functional specification of the system it will be part of. The Prototyping Service provides all the communication infrastructure, as well as support for the mapping of the data formats from the distributed objects domain to the hardware domain (Figure 6.11).

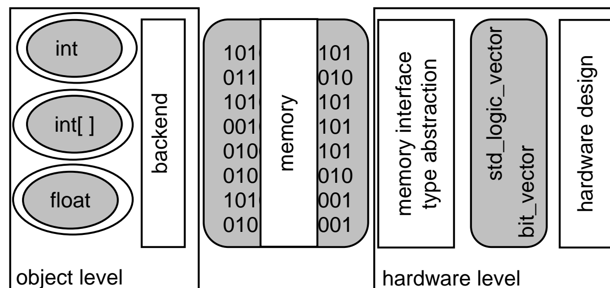


FIGURE 6.11 – Abstraction layers between object domain and hardware domain

Further details on the Prototyping Service can be found in [IND2003], and a case study based on its usage model is presented in Section 8.

6.3.5 Additional Services

The encapsulation of legacy design tools into the Cave2 Framework should be done in a similar way as the approach reported in the previous subsection. As every legacy tool has its own peculiarities, a backend module would be needed, in order to export the desired functionality as an API. On top of the backend module, a Service Space-compliant service definition should be implemented. Foundations for the implementation of this definition are available within the framework, so the legacy tool integrator has its task restricted to:

- implement the backend module;
- extend the Service Space service definition to implement the interface defined by the backend module API;
- extend the service proxy definition to implement the interface defined by the service definition;
- partition the desired functionality to be done within the Service Space (authentication, location of additional services, load balancing, caching, etc.) between the service and its proxy.

If user interface modules are needed for the legacy tool operation, such models should be made available in the Framework Server or be encapsulated within the proxy object, so they can be instantiated within the Cave2 cockpit and rendered as part of its GUI.

7 Supporting Collaborative Design

7.1 Introduction

The need for collaborative computer-aided design can be easily justified, as the complexity of designs continues to increase, demanding larger teams and higher productivity. Furthermore, the current shortage of qualified design engineers require companies to search for manpower from different parts of the world, and in many cases it is more convenient not to relocate them. As a third reason, there is the need for experience sharing among the members of a team, allowing the beginner to learn from the more experienced.

Some of the approaches reviewed in Section 3 support some kind of collaborative work, focusing on asynchronous collaboration by using data versioning or workflow modeling. Our approach differs from those by providing support to both synchronous and asynchronous collaboration, and by supporting an explicit separation of concerns between the design model and its visual representation. Both features are tailored specifically to the application domain of the Cave2 Framework.

The first feature fits to the application domain of integrated systems design, where synchronous collaboration is needed in the first steps of the design process and asynchronous collaboration is used in the late, implementation-related steps. For example, a high level of collaboration is expected on the first steps of the design - where product functionality and constraints are defined - because of the inherent multidisciplinary nature of such activities. Hardware engineers, software developers, marketing staff and product management are among the types of professionals that may participate synchronously in such collaborations. On the other hand, during the implementation steps - coding, hardware debugging, etc. - the collaboration level is expected to be low, because developers tend to work individually and asynchronously

The second feature takes into account the variety of possibilities for design entry and visualization in the area of integrated systems. Graphic schematics, such as logic netlists, UML diagrams and state charts, are often used concurrently with HDL and programming language code throughout the typical integrated systems design flow. To allow the collaboration among designers using different design visual representations, a flexible design representation should be created, and the consistency between such representation and its different visualizations should be kept.

The chosen solution used and extended the infrastructure provided by the object-oriented framework described in Section 5. The support for collaborative work was included in the classes within the design data representation framework,

and the behavior of the collaborative sessions is controlled by the Collaboration Service in the Cave2 Service Space. The following subsections detail both parts of the solution. First, the synchronous collaboration support is covered. In subsection 7.2, the separation between design model and visual representations is discussed. The approaches to consistency control are described in subsection 7.3. Then, the possibilities in asynchronous collaboration and covered. One of them, the inclusion of versioning support in the Cave2 design data model, is covered in subsection 7.4. Subsection 7.5 covers the inclusion of metadata in the design model – as comments, annotations and guidelines, for instance. This is probably the simplest yet the most used approach for asynchronous collaboration, and its application within the Cave2 environment improved its potential to support design experience exchange.

7.2 Design visualization issues

In order to allow multiple visual representations of a single design data block, the underlying framework should have facilities to allow the data blocks to maintain coherent visual representations of its state, even when they are under edition. For example, if a designer has two visual representations of a design – i.e. two schematic windows, one with an overview and other with the details of one block – the changes done in the data model through one of them should be notified to the other and the proper updates should occur.

In the literature, many successful approaches to define an architecture to model information and its visual representation consistently are based on the separation of the information model and its visualization, while providing a consistency control between them. This architecture was a key concept on the MVC (Model-View-Controller) framework within the Smalltalk programming language [KRS88] and was later formalized as the Observer design pattern by Gamma et. al. [GAM95], reviewed in subsection 4.2.2 of this text.

By decoupling the model and its visualization, it is possible to provide several different – but equivalent – visual representations for the same design. For particular representation formats, more efficiency can be achieved by using this approach, because the consistency between the model and the view must be kept only when a modification on the visual representation results also in modification in the model semantics.

In the Cave2 Framework, the design representation primitives - e.g. logic gates, functional blocks, etc. - are modeled as instances of a concrete class, which inherits behavior from an abstract class. This is common when it comes to object-oriented frameworks, because the abstract classes - while not modeling anything in the application domain - organize the class hierarchy and allow the assignment of common behavior to a particular set of objects. We use such concept to include the support for the collaborative methods: that support was included in the superclasses of all the design primitive classes. So, all the design objects in the Cave2

Framework - including the ones which will be integrated in future updates - will inherit such behavior.

By relying on such concepts, we could achieve complete separation of concerns regarding the design semantics and its graphical/textual representation, as they are modeled by different objects. Thus, we can support several visualizations - by different designers - from a single design block. To grant the consistency between the design semantics and its representations, as well as between the semantics of inter-related design blocks, we use update/notify mechanisms. Such mechanisms capture the interaction between the user and one of the views, update the respective semantics and notify all views to update themselves to reflect the possible changes.

A framework tailored to provide a flexible notification infrastructure is presented in [SHN2002]. It supposes a collaboration scenario where several users are updating a common data set, so the changes in the data state performed by a given user should be notified to the other ones. It divides the notification in two parts, the incoming notification and the outgoing notification. For each participant, the incoming notification denotes the changes performed by other users, while the outgoing notification propagates to the other users the change performed locally. Each of the types of notification is characterized by its frequency and its granularity. The frequency of the notifications can assume one out of three possibilities – instant, scheduled or user-driven – while granularity defines if a notification should be done for every state change, or only a defined subset of the changes.

In our implementation, we applied a notification-based approach similar to the one reported by [SHN2002]. Developed independently in its early steps, our approach benefit from the systematic view of the problem presented in [SHN2002] and applied such techniques into an extension of the underlying Repository Service infrastructure.

Using such features, several designers can work synchronously over a design. When two or more designers are working over a single design block, the object representing the design block semantics is stored in the Repository Service, while a view of that block is instantiated for each one of the designers and stored locally within the cockpit's JVM object heap. The interaction between the designer and the design block occurs through that view: when the view is modified in such way the design semantics changes, the view object notifies the block object in the Repository Service, so it can update its state to reflect the change. Once the state is updated, a notification is done to every view of that block, making the changes visible to the other users. It is important to note that when the interaction between the designer and the view is not changing the design semantics - e.g. when the designer moves a block in the design sheet without changing its interconnection with the other blocks - the update/notify mechanism may be not activated. This approach is called visually decoupled (Figure 7.1a), because the view for each user can - and probably will - be different from the others.

While allowing rich collaboration through the concurrent access to design blocks, this approach has the disadvantage of making impossible the spatial referencing of design blocks by the designers: one designer would fail to reference a particular block while communicating with another designer if he/she says e.g. "that FIFO in the left side of the encoder", because probably the blocks positioning in the screen of his/her colleague would be different. So, when spatial referencing is desired, we provide a visually coupled approach, where the update/notify mechanism is slightly different. Two implementations for this approach were experimented. In the first case there is only one view, stored together with the block in the Repository Service. Every designer would have in his/her GUI a reference to that view, so every update - even the ones which are transparent to the design semantics - would be noticed by every designer (Figure 7.1b). In the second case, we follow the same mechanism described for the visually decoupled approach, but updates and notifications are issued also for the events that change only the visual state.

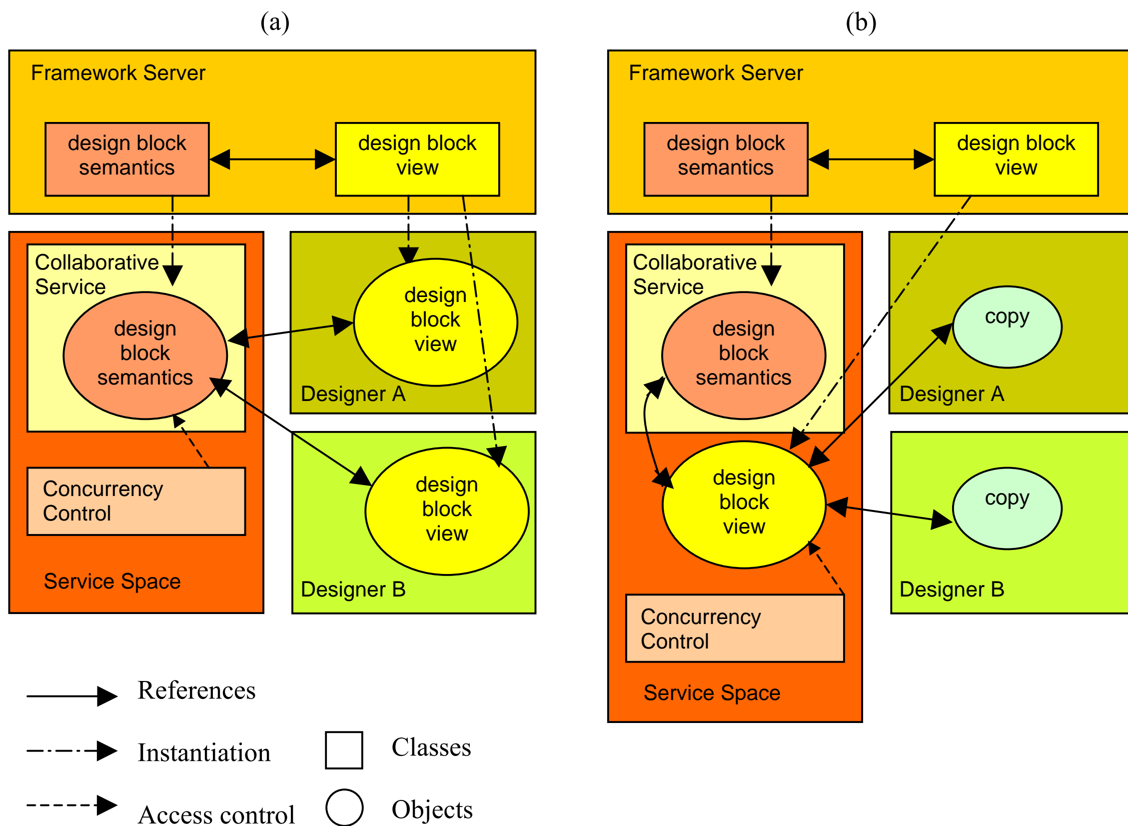


FIGURE 7.1 - Implementation alternatives for the visualization of design data

Taking into account the 1-to-many cardinality in the relationship of the design blocks with their views, it is easy to realize the need for an data access control mechanism. The possibilities of deadlock in the update/notify mechanism (i.e. when two views try to modify the semantics at the same time) and inconsistencies between the block and its views (i.e. when a view tries an update to the semantics before the reception of the notification of an update done previously) are some of the problems

that may occur. The Collaboration Service performs such control by providing a common interface for every tool in the Cave2 environment, allowing them to use the design primitives in a collaborative way. Located in the Cave2 Service Space and working together with the Authentication and Repository Services, the Collaborative Service uses their underlying infrastructure to organize the access to the design objects by the users.

7.3 Concurrency control issues

In order to organize the access to the design objects by multiple users, a concurrency control system is needed in order to grant the design objects consistency. Any particular concurrency control system makes assumptions regarding two basic properties of the application: the semantics of the object being shared and the consistency requirements of the application [MUN96]. In Cave2, those two properties are clearly divided respectively between the primitives within the object-oriented framework, covered in Section 5, and the concurrency control mechanisms of the Collaboration Service. Let's take a closer look on the issues stated by [MUN96] and derive what are the consistency requirements in Cave2.

A major consistency requirement on every data model is derived from the multiplicity of the its relationships. In the Cave2 design model, the multiplicities of the object relationships are shown in the UML class diagrams depicted in subsection 5.3. The most critical cases are the relationship roles that can be taken by one and only one object. In our implementation, such consistency control is partially granted by the model itself, as the relationship is modeled as a reference to a single object so no exceeding references are allowed. However, the possibility of zero references may arise - for instance if the referenced object is removed - and such possibility must be dealt by the concurrency control mechanism, as we will cover ahead.

Another major requirement is a very typical one when dealing with data models based on the *5-box* approach: instance ports must match the external ports of the instantiated block. This means that if a port is removed or added to the instance, the same must happen on the respective block. Furthermore, such change must be propagated to all the instances of such block. In our initial implementation, this was granted by the model itself, by cross-referencing the model and its instances, so each instance would be responsible for changing its respective block in the case of chance, and the block would be responsible for changing all other instances. Some implementation issues, described ahead in this subsection, motivated us to use another entity within the design data model to deal with this problem.

Yet other problems may arise, specially when the bi-directionality of the relationships modeled as a symmetric pair of references cannot be granted. It can happen, for example, when an instance of *ExternalPort* references an instance of *DesignBlock*, but that block does not have the port in its port list. In Ptolemy II

[LEE01], this issue is addressed by introducing a mutual-exclusion mechanism. Every design block in Ptolemy II model is assigned to a workspace when of its creation, and every time such block is updated, the complete workspace is locked, so no concurrent updates are done. Such approach allows consistency in a multi-threaded environment – a so-called *thread-safe* approach - and the locking of the complete workspace avoids the deadlock possibilities that may arise in multi-granular lock.

For our approach, such solution would not be feasible because it would hamper the collaboration possibilities. An alternative approach is to ensure that such relationship is atomically established. For instance, there is only a method to assign a port to a block - in the container - and such method implements the symmetric references in both container and contained elements. However, if no locking mechanism is used such implementation may have its consistency violated in an unsynchronized multi-threaded implementation, as the updates done by two different threads may overlap. Such issue must also be handled by the concurrency control mechanism.

Further consistency issues arise when the data model is extended to represent more specific design entities. Consistency rules such as “attributes associated to these entities should be kept within a given range of values” or “connections between ports can only be placed if their data types match” can be often found in design systems [WAG94]. In a data model specified as an object-oriented framework, such issues should be handled by the framework extensions alone. Examples of such rules can be found on the framework extensions implemented as case studies, detailed in Section 8.

Once we had highlighted the consistency issues that must be handled by the Cave2 Framework internal services, we should investigate the best mechanism to address them and yet comply with the main goal of Cave2, which is to support collaboration in a distribution design environment.

In database systems, concurrency control is usually based on serializable transactions [ESW 76]. In those cases, concurrent transactions should be isolated from each other, so the concurrent execution of a set of transactions is strictly equivalent to its serialized counterpart. For collaborative systems, such approach is often considered too restrictive [MUN96], because in many cases the interference between two transactions is actually the collaboration activity the system is supposed to support.

Several techniques are available on the literature regarding less conservative concurrency control methods aimed to support collaboration [ELI91].

Turn-taking protocols, such as floor control or pair programming, can be viewed as a concurrency control mechanism. The main problem with this approach is that it is limited to those situations in which a single active user fits the dynamics of

the session. It is particularly ill-suited for sessions with high parallelism, inhibiting the free and natural flow of information. Additionally, leaving floor control to a social protocol can result in conflicting operations: users often err in following the protocol, or they simply refuse to follow it, and consequently, several people act as though they have the floor.

Another concurrency control solution is to introduce a centralized controller process. Assume that data is replicated over all user workstations. The controller receives user requests for operations and broadcasts these requests to all users. Since the same operations are performed in the same order for all users, all copies of the data remain the same. This solution introduces the usual problems associated with centralized components - a single point of failure, a bottleneck, etc. - and several other problems also arise. Since operations are performed when they come back from the controller rather than at the time they are requested, responsiveness is lost. The interface of a user issuing a request should be locked until the request has been processed; otherwise, a subsequent request referring to a particular data state might be performed when the data is in a different state.

The dependency-detection model is another approach to concurrency control in multi-user systems. Dependency detection uses operation timestamps to detect conflicting operations, which are then resolved manually. The great advantage of this method is that no synchronization is necessary: non-conflicting operations are performed immediately upon receipt, and response is very good. Mechanisms involving the user are generally valuable in collaborative applications, however, any method that requires user intervention to assure data integrity is vulnerable to user error.

Reversible execution is yet another approach to concurrency control in collaborative systems. Operations are executed immediately, but information is retained so that the operations can be undone later if necessary. Many promising concurrency control mechanisms fall within this category. Such mechanisms define a global time ordering for the operations. When two or more interfering operations have been executed concurrently, one (or more) of these operations is undone and re-executed in the correct order. Similar to dependency-detection, this method is very responsive. The need to globally order operations is a disadvantage, however, as is the unpleasant possibility that an operation will appear on the user's screen and then, needing to be undone, disappear.

Another approach to concurrency control is operation transformation. This technique can be viewed as a dependency-detection solution with automatic, rather than manual, conflict resolution. Operation transformation allows for high responsiveness. Taking as an example a multi-user synchronous editor, when an operation is requested (i.e. a key is typed), the editor locally performs the operation immediately. It then broadcasts the operation, along with a state vector indicating how many operations it has recently processed from other workstations. Each editor instance has its own state vector, with which it compares incoming state vectors. If the received and local state vectors are equal, the broadcast operation is executed as

requested; otherwise it is transformed before execution. The specific transformation is dependent on operation type (for example, an insert or a delete) and on a log of operations already performed.

An hybrid concurrency control model called LasCoW was proposed in [PIE96], based on the different collaboration needs found among the different groups of users in a collaborative, distributed application. According to their approach, a hierarchical composition of user groups should be created and assigned a consistency domain. The consistency control within a domain is expected to be harder, while more flexible between domains, relying on the assumption that the interaction between domains is much smaller than intra-domain interactions. Implementation issues mentioned but not detailed in [PIE96] include the mobility of users across groups and the substitution of consistency control between domains.

Taking into account the reviewed alternatives on concurrency control mechanisms, we derived an hybrid approach aimed to match the particular needs of the Cave2 Framework. This approach is built on top of the Repository Service, described in subsection 6.3.1. That service provides a simple interface to hide the complexity of the data repository, which can be anything from a simple file system to a complex database management system. The Collaboration Service extend that approach by introducing the concept of a collaborative session. Such session comprehends a group of designers working together towards a common goal. We can assume that a repository would host design data for a several designer groups, with different goals, so concurrent collaboration sessions are likely to happen. It motivated us to pursue a multithreaded approach to the Collaboration Service, where each collaboration session is associated to an execution thread within the Collaboration Service. All the data access done within each collaborative session – thus within each execution thread of the Collaboration Service – must be performed within transaction borders, in order to ensure thread safety. However, such approach grants that the restrictions which are associated to the transaction control are enforced to the groups only, so the users within a collaboration session are not isolated from each other.

So, the proposed approach allows any designer to access freely the data from the repository. If the data is not being accessed, his/her tool will open a collaboration session and retrieve the data within a transaction. If the data is already under edition, the designer would have to access those data records within a collaborative session (thus, be handled by the same thread within the Collaboration Service which is already handling the access of the other designers working in those records). Figure 7.2 depicts such approach. In the figure, three designers access the data repository concurrently. Design data objects 1, 2 and 3 are accessed within the transactional control of collaborative session 1, while objects 9 and 10 are accessed within the transactional control of collaborative session 2.

The implementation of transaction mechanisms is often available in the underlying repository and already embedded in the *retrieve* method of the Repository Service, which is inherited by the Collaboration Service. However, if the underlying

repository does not offer transaction mechanisms, they must be defined within the service implementation.

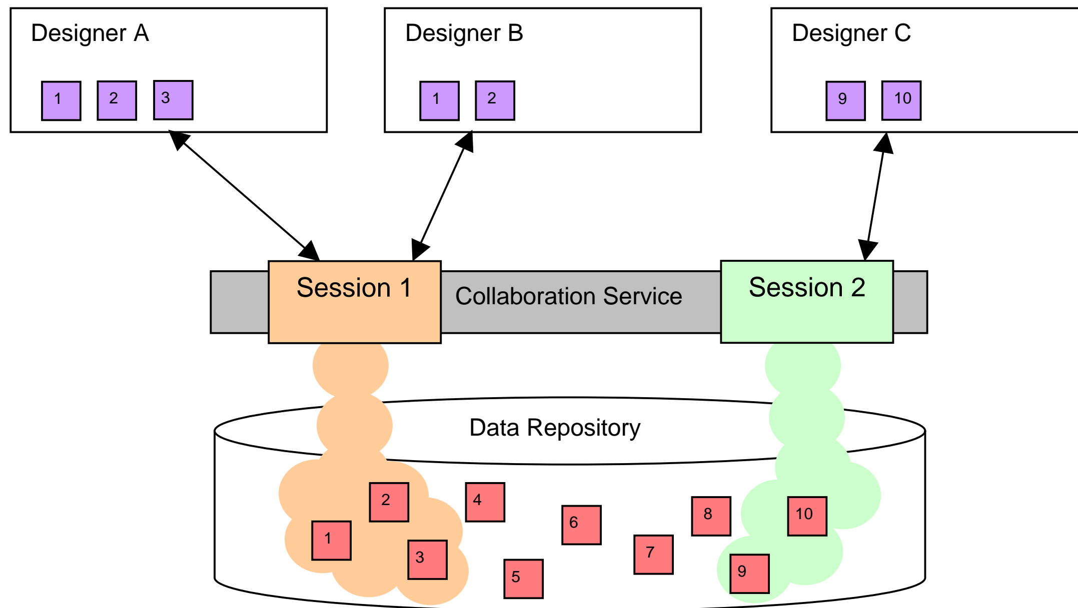


FIGURE 7.2 – Collaboration Service Overview

The presented approach insures the consistency between concurrent multi-user collaborative sessions. However, the concurrency issues within a collaborative session are still opened. To address such issue, our approach advocates for an internal concurrency control mechanism for each collaborative session. This allows flexibility on the collaboration, as each group of collaborating designers can choose the best collaboration methodology for its task. To validate such proposal, two internal concurrency control techniques were implemented.

Our first implementation was based on the floor control consistency control. A floor control methodology called Pair Programming [WIL2000] was extended to support any number of distributed collaboration partners, and an implementation was included in tool prototypes for both diagrammatic and textual design models. The details on the implementation of such prototypes are available in [HER2001, SAW2002]. The turn-taking is decided by consensus among the current token holder and the ones requesting it. While appropriate in the cases when one of the designers is more experienced than the rest – for instance, in training sessions – the turn-taking allowed a low degree of collaboration as most of the designers would be most of the time watching and suggesting, rather than actually designing.

To better understand the secondly implemented extension, let’s review the interaction between a designer, the design data model and its visualization according to the Repository Service scheme. For every update done by the designer, the design data proxy object is notified, an event is sent out to the Repository Service via its service proxy, and the design visualization is updated. Notice that there is no

requirement that the event should be acknowledged by the Repository Service before the visualization is updated. The design object proxies assumes that the event is received, so it immediately triggers the visualization update. The responsibility of the event delivery is delegated to the Service Space, which guarantees that it will happen successfully otherwise an error notification is issued. In our implementation, Jini's event model handles that, avoiding the possibilities of lost events or unordered execution. Taking this into account, and since the Repository Service ignores the possibility of multi-user access, and since major semantics-related consistency issues are usually handled by each individual proxy object, the immediate update of the visualization after the instantiation of the event object can be accepted.

If the possibility of multi-user access to the data is regarded, as it must be within the Collaborative Service, such procedure can no longer maintain the data consistency. Event ordering would become impossible, since this procedure will always execute local events before it executes remote events. To avoid that, the behavior of design data proxy objects in the Collaborative Service are slightly different from their Repository Service counterparts. For events which are particularly sensitive to the execution order, the update of the visualization is postponed until the event is actually executed in the design data object within the repository. This approach can be considered a variation of the centralized controller approach described earlier in the text. The difference is that only some of the events are resolved by such central control – performed within the collaborative session thread – while others which are not sensitive to the ordering are propagated immediately to the visualization.

The discrimination of events which are sensitive to the ordering and those which are not is domain-dependent. In order to avoid the problem of low responsiveness of the user interface – a consequence of the central controller – we should avoid as much as possible the postponed update of the design visualization objects. In our design data model, we can already discriminate events that can be executed directly, granting that high responsiveness of the user interface for those cases:

object creation – in the proposed model, the object creation is not sensitive to the ordering. A design block creation by one user can happen either before of after the creation of another block by another, because the final result would be the same. We can be sure that there are no relationship between both objects at the moment of their creation, as one user does not have yet the awareness of the creation by the other user;

block instantiation – the creation of instances of a block can also be done by different designers without conflicts or inconsistencies;

port assignment – the order of the port assignments to blocks is not sensitive to the order, as the ports are identified within a given block or instance by their object keys, and not by their order;

- superclass assignment – the proposed data model supports multiple inheritance, so types can have many parent types. The order the parent types are specified is not important, though;
- connection between ports – the proposed data model represents the connection between ports by transaction objects. Each transaction object has a list of ports which are connected, and the order of the ports within this list is irrelevant. For instance, the results are the same if port A is connected to port B and then port B is connected to por C, or vice-versa.

In other types of events, namely object deletion and attribute update – name change, for instance – the centralized control within the collaborative session thread should be enforced. As mentioned earlier, it reduces the responsiveness of the user interface – the event triggered by the user is not perceived immediately. This issue can be reduced by using tool-specific measures, such as updating the user interface twice, once when the event is triggered by the user and then when the event is acknowledged by the Collaboration Service. For instance, when a given block is deleted, its representation can be rendered in grey right after the user triggered the deletion, and removed completely when the acknowledge from the Collaborative Service is received.

For a visually coupled approach, state changes in the visual representation of the design data should also be propagated to collaboration parties. Such state changes are also encapsulated in events and propagated to the Collaboration Service via the service proxy, as the regular events. The order of such events is also not relevant, so they can be handled in the same way in collaborative sessions, without the use of a centralized control. For instance, a given block can be translated 200 points in the positive direction of the x-axis and then be added 20 points in its width, or vice-versa.

To implement the event propagation system, we followed some of the guidelines proposed by [SHN2002], so an incoming and an outgoing event queue is integrated into each collaborative session, as well as into each service proxy. Every event generated by the designers is queued in the outgoing event queue of their tools' service proxy. The events are then sent to the incoming event queue of the collaborative session, executed and placed in the outgoing event queue. The events are then transferred to the incoming event queue of each collaborating designer, and the events which were not yet reflected into the visual representation (because they were order-sensitive) are finally consumed. Figure 7.3 depicts this procedure.

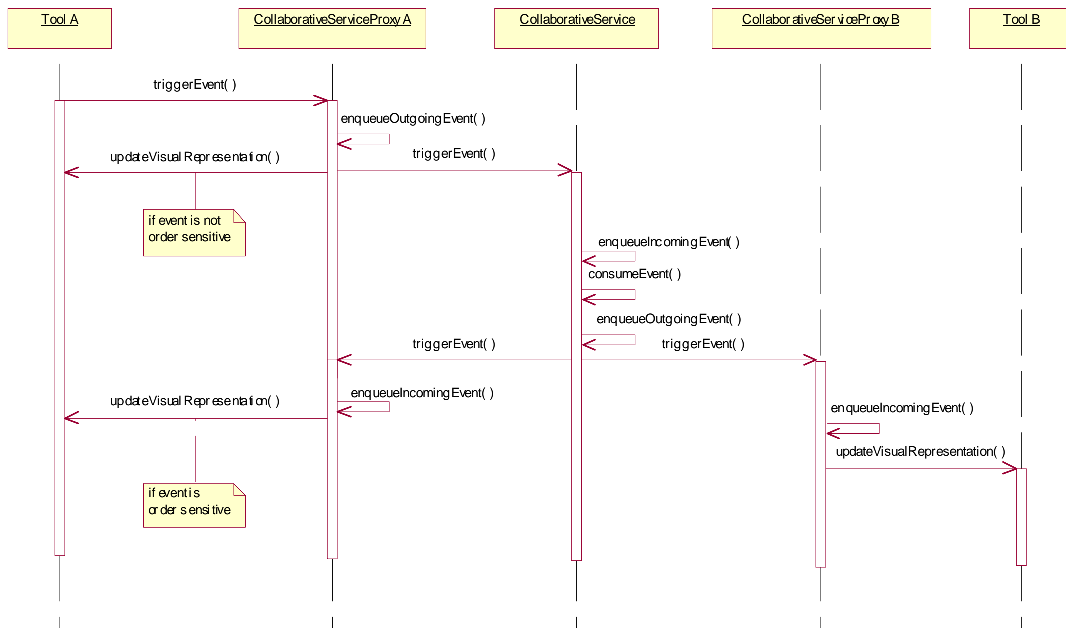


FIGURE 7.3 – Collaborative Service UML Sequence Diagram

Some implementation issues had to be solved in order to achieve the complete functionality of the Collaboration Service. The first one regarded the consistency between a block interface and the interface of its instances. As mentioned earlier, this issue was originally dealt by the data model alone. However, such scheme using cross-references between the blocks and their instances may create problems with the boundaries of the transactional control performed by the Collaboration Service. In the case of the concurrent access of instances of a given block in different collaborative sessions, the block-instance consistency maintenance will depend on the particular implementation of the transactional control. As an example, let us consider the access of an instance I1 within collaborative session S1, and the access of another instance I2 within collaborative session S2. Both I1 and I2 are instances of the block B. If the interface of I1 is changed, it must propagate such changes to B, which would then propagate to I2. If the underlying transactional mechanism is pessimistic, the collaborative session S1 would lock B once I1 obtains a reference to B in order to propagates the changes through method calls. This would hinder the access to B by the session S2, so the possible changes done within I2 would not be propagated until S1 commits.

In order to address such issue, we introduced an entity called Cave Design Unit Manager (CDUM), which is in the boundary between the data model and the Collaboration Service. It has the responsibility to propagate changes from instances to blocks and vice-versa, and it uses the Collaboration Service to do so. For every change performed within an instance, a new event is created by the CDUM and this event is included in the incoming events queue of the collaborative session which currently has access to the block referenced by the changed instance. If the block is not being accessed in the moment – thus, no collaborative sessions have it within

transactional boundaries – the CDUM updates the block directly within a self-initiated short-living transaction.

Once the block is updated, all its instances should be updated as well. If any of the instances is under control of a collaborative session, the CDUM uses the same procedure mentioned before, adding the event in the sessions' incoming queue. If a given instance is not being accessed, the CDUM initiates a short-living transaction to do the update directly. Notice that the CDUM will only propagate events which are not order-sensitive, as it does not have any mechanism to acknowledge the successful event processing. In the current implementation, order-sensitive events - such as the removal of a port - can only be accepted if the collaborative session can have the block and all its instances within the boundaries of its transaction.

Other implementation issues could be solved with the introduction of the CDUM. First, it can be used to store the visual representation of its respective design unit, so it can provide it to design tools that access the design unit for the first time. This feature is not necessary for a visually decoupled approach, as the tools may be able to create a visualization based on the block model data alone, but it is critical for the implementation of a visually coupled approach, in order to provide a starting point for the collaboration between partners.

The CDUM also maintains a log of events consumed by its respective design unit, its ports and – if hierarchical – its top level contents. Such log can be used to synchronize visualization objects which were offline (no connection between their proxies and the Collaboration Service) during periods when there were changes done through other visualization objects. Such synchronization, however, should be initiated and performed by the tools themselves. By using together the log of events and the stored visualization, a design tool can always build the right visualization when joining either a visually coupled or visually decoupled ongoing collaboration session.

7.4 Versioning Support

In many cases concurrent access conflicts cannot be solved automatically, and in some cases there is not even possible to achieve conflict resolution at all. Furthermore, in some cases the interest of the designers is to keep the conflicting versions of the design as alternatives. All such scenarios are usually comported by versioning control systems, for instance some of the approaches reviewed in Section 3. Some of such approaches are generic enough to be applied to any kind of data model, so we rely on some of its concepts to implement versioning support within the Cave2 Framework. However, the integration of those approaches into the Collaboration Service is not straightforward, because of the differences on the level of access to the designed data allowed by our approach – based on multiple proxies – in opposition to the centralized approach behind most of the versioning

control systems. A solution must be found, in order to allow the following functionality to be implemented:

a visual representation object may ask its proxy to create a new version of its corresponding design object and execute all subsequent changes in the new version;

a visual representation object may ask its proxy object to navigate back and forth through the version history of its corresponding design object;

during synchronization, a service proxy of a Collaboration Service may need to create new versions of design objects which were updated while it was offline;

a designer wants to edit freely the design blocks that are currently controlled by another collaborative session, so he/she can decide to derive a new version instead;

two designers are asynchronously working in a given design block, and they reach a point when they should merge their implementations (this functionality is not supported by all the versioning systems reviewed in Section 3).

In order to preserve the independence from design data and design management data within the framework, our solution for the versioning control is embedded in the identification key of each design data object. We extended the key - which was originally used to support the queries over the design data - in order to include the version information. Thus, each design object proxy is associated to a given design object through its identification and version. In the repository, each key references the keys of the previous and next versions of a given object. In order to support merge operations, references to more than one previous version is necessary. Such references model a version graph which is similar to some of the versioning systems reviewed in Section 3. It also supports graphs which are not tree-like, allowing the possibility of merges.

According to the Cave2 definition of design updates, one version of a given block would be differentiated from another by the set of events they have on their update log (maintained by the Cave Design Unit Manager). Thus, subsequent versions of a given block should store only the events that were issued after its creation. However, as the previous version may also evolve independently, it is necessary to the new versions to keep track on the last pertinent event from the parent's log, so if a complete event history is needed it can be reconstructed. In our implementation, a reference to the last pertinent event on the previous version is included as part of the entry in the previous versions list. If the version is a merge, each of previous versions will be entered in the list with its last pertinent event tag.

Notice that the described approach of versioning can support the most immediate need for versioning – the need for managing successive revisions. Other

planes of versioning – such as managing different views or different alternatives – are not supported by the that approach.

To address such shortcoming, we introduce a composite key that supports multiple planes of versions, such as the STAR Framework approach [WAG94]. The composite key re-implements the ViewGroup approach from [WAG94], so it is not assigned to any design object but references a group of other keys, which may or may not be a composite one. Figure 7.4 depicts the complete implementation of such functionality within the Cave2 Framework. The composite key is modeled by the *VersionGroupKey* class, which implements a Composite pattern together with the abstract *VersionKey* class.

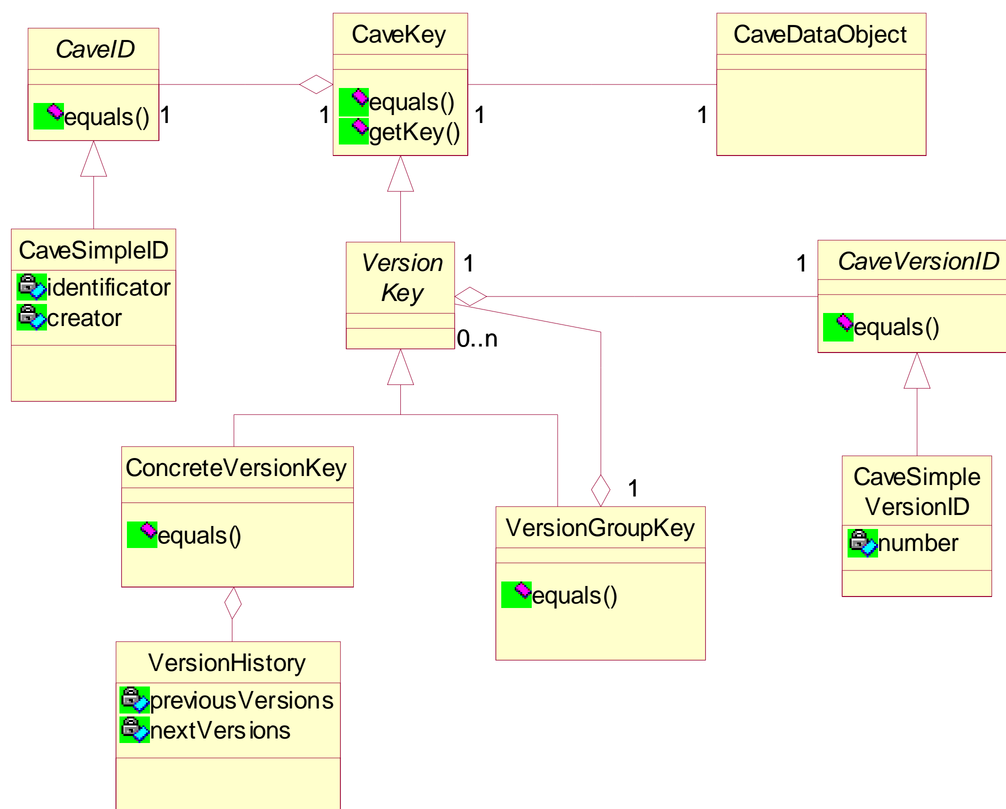


FIGURE 7.4 – Embedding versioning information within identification keys

The same strategie we used to allow further implementations of design data identification tags was used for the version tags. The procedure of matching both identification and version tags is denoted by the method overriding shown in the subclasses of *VersionKey*. The *ConcreteVersionKey* actually manages the incoming and outgoing paths of its node in the version history graph, while the *VersionGroupKey* grants the possibility of further levels of versioning within a node.

7.5 Metadata Support

The incorporation of metadata is probably the simplest yet the most used approach for asynchronous collaboration in design. Comments in HDL or programming code and annotations in schematics and diagrams have been used successfully for decades, and such practice is unlikely to disappear within designers, no matter how much the collaboration support is improved in the design tools. In order to support such practice, the Cave2 Framework should incorporate metadata into its data model.

Our definition of metadata covers all the meaningful information about the design included by the designers or tools, but which is not part of the design data itself: design documentation, design requirements and constraints, design management information, support for data authentication and security guidelines are some of the information that may be modeled as metadata.

In order to open new possibilities on design documentation, we propose the support of metadata in multiple formats: multimedia metadata. Such approach was already introduced in the first version of Cave, which organized multimedia metadata inside hyperdocuments. In Cave2 the same functionality is provided, but additional infrastructure is needed. In the previous version, the metadata units were available as files inside of a HTTP server, so they could be directly accessed by the designers through predefined URLs. The search was done using regular web search engines, and there was no explicit relationship between the metadata unit and its respective design data unit.

In Cave2, the metadata units are included as part of the design data model, so their relationship is explicitly maintained by the framework. Thus, each metadata object stores the metadata contents and a reference to the respective design data object. In order to allow its access via HTTP, every metadata unit is able to export itself as a file. To handle the different media types – which should be notified to the HTTP server by the generated file extension – the metadata objects also store the MIME type of its content.

In Section 8.4, a case study using extended metadata objects is presented. Such objects were used as building blocks of web-based training and educational activities.

8 Case Studies

8.1 Introduction

This section of the text describes some of the implemented applications of the proposed framework. In subsection 8.2, the Prototyping Service is used to implement a cryptography system based on the DES algorythm. Subsection 8.3 presents IBlaDe, a design tool supporting Interface-based Design methodology. In the same section, several extensions to the framework are introduced. Such extensions were necessary for the implementation of the IBlaDe tool and validate the extensibility of the Cave2 Framework. Finally, subsection 8.4 describes an extension of the framework supporting the authoring of educational and training material.

8.2 Prototyping Service

In order to validate the proposed approach for the Prototyping Service, a cryptography system was implemented. It was designed over the proposed infrastructure, so the system could take advantage on reconfigurable hardware modules which were available in the network.

The first experiment covered the application scenario where a developer can incrementally prototype the target system described in the functional level. The chosen example is a messaging system that sends and receives encrypted messages using the DES encryption algorithm. We implemented the whole system functionality using Java language, so that the potential users can evaluate if it fulfill their functional needs. In the next step, we started the incremental prototyping of the system by implementing some system modules in reconfigurable hardware. So, the DES encrypt and decrypt modules had to be converted to HDL in order to be synthesized and implemented in our FPGA prototyping board. We used an HDL core for the DES algorithm [FRI2002] in this implementation, but when such possibility is not available the conversion can be done with design automation tools such as Forge [DAV2001] or even by hand if the design isn't too complex. We used the type abstraction and the memory access interfaces depicted in Figure 6.11, so the HDL core could be integrated easily.

After the configuration was generated, it was stored in a JavaSpaces repository. A Jini service federation was initialized within the Service Space, and a service for the FPGA board was registered on it.

In the application side, we replaced the software objects which were performing the DES encrypting and decrypting by proxy objects with the same external interface. The rest of the application objects were not changed, because the

API they used to communicate with the DES objects was kept. The internal implementation of the DES objects was changed into proxies, relying on our implementation of a Jini client, so they contact the FPGA service every time a DES encryption or decryption was requested by the application. The FPGA service downloads the configuration from the JavaSpaces, program the FPGA, receives the data from the proxy objects, maps it into the FPGA memory, starts the FPGA execution, reads the processed results and returns them to the proxy object.

We successfully implemented such scenario, and the use of such prototyping strategy was found very convenient. It made possible the functional validation of the prototyped design block, as it was tested together with the rest of the functional description. The use of the proposed abstraction layers between the object domain and the hardware domain allowed a clear separation of concerns, making easier the development on each of the sides.

After the experimentation of the proposed approach as a support for incremental prototyping, we decided to test its suitability on the support of distributed processing. We envisioned a scenario where a particular computational task could be used remotely by a device with small computational power, like a mobile phone or PDA (Figure 8.1). In such case, the device would not be interested on simulating the communication between parts of a system, but actually request a particular data processing task which would be too costly for it to implement alone. In such cases, the ubiquitous nature of our approach would be critical, because the small device could be mobile - perhaps the processing reconfigurable units too - and a greater variety of tasks to be performed could be available. We used the same DES implementation described in the previous scenario, but this time its implementation could be seen by the application system as an ubiquitous reconfigurable co-processor.

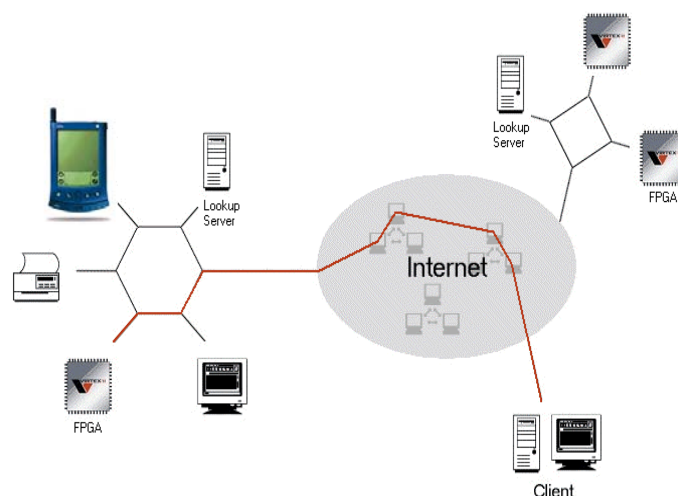


FIGURE 8.1 – Case study on reconfigurable computing on demand

The implementation was successfully achieved, and the application used our infrastructure to request to the Jini service federation for a co-processor for

computationally intensive tasks. As expected, the actual execution time of the DES algorithm in the FPGA implementation is much faster than the implementation is software, as shown in Table 8.1. The first two lines show the maximum data rates for the encryption and decryption in the application without the use of the reconfigurable co-processor. The third line shows the data rate obtained by the co-processor.

TABLE 8.1 - DES algorithm implementation comparison

Processor	Clock (MHz)	Data rate (Mbits/s)
Sun JVM over Intel Pentium MMX	133	0.162
Sun JVM over AMD Athlon XP-1700	1467	1.295
Xilinx XCV-800	27.23	108.93

However, such gain could not be delivered to the client applications. The overhead of the FPGA configuration and of the data transfer from the host computer to the reconfigurable hardware module were too big, hindering the major goal of providing a significant acceleration compared to the computation that would be done locally by the client device. Such overhead can be reduced if there is no need for frequent reconfiguration of the modules - e.g. when there are many reconfigurable hardware modules in the Jini federation so the client can query for a module already configured with the needed functionality.

The data transfer overhead can be also minimized, if the host computer and the reconfigurable hardware module are connected through a faster interface. In our prototype, the communication between the host computer and the FPGA board was done via a parallel interface, and the access to the SRAM banks was very slow. Much better results could be obtained by using a reconfigurable hardware module connected to the backend through a high speed bus, such as in the Pilchard platform [LEO2001].

8.3 IBlaDe

In order to validate the extensibility features of the proposed framework, a new design tool was implemented on top of the framework foundations. Several tools were already implemented during the Cave2 prototyping cycles, such as those referenced in subsection 5.2. None of them, however, take advantage on all the framework features, as some of them were introduced in the latest prototyping cycle. The IBlaDe tool was developed primarily to validate those features, but it also has its own merit described in the subsection 8.3.1. The resulting extension on the Cave2 Framework is described in subsection 8.3.2.

8.3.1 Interface-based Design

The separation between system functionality and implementation during design was the key concept behind interface-based design (IBD) [ROW97]. This methodology proposed the successive refinement of the intercommunication structures between design modules - thus the functionality provided to other modules - aiming to accelerate the verification procedure on the early phase of the design process. Relying on the assumption that a lower level of detail of the interconnection models can be accepted when designing the system functionality, the IBD methodology proposed the substitution of cycle-accurate simulations by untimed transaction-based verification, which require less computing power thus reducing the verification time.

The possibility to successively refine the implementation of a given functional composition is an appealing idea to the designers. It fits well to the iterative methodology they are used when designing systems in a lower level of abstraction, such as RTL-level design, and the functional specification could be seen simply as a reference all the implementation requirements should comply with. Several approaches to wrap up IBD concepts in such a way were proposed by industry and academia.

While no design data model was presented together with the concepts of IBD, many authors have reported design systems that in one way or another would support those concepts. The Ptolemy II system [LEE01] separates clearly the communication between design modules from its internals by relying on the concept of actors. Each actor encapsulate a particular implementation but through inheritance it also can be seen as a kernel entity which handles hierarchical compositions and communication with other entities through ports. Zhu and Malik [ZHU02] brought from object-oriented design the Façade pattern [GAM95], aiming to have a clear definition on the externally-visible functionality of on-chip interconnection networks. The methodology is also available in a commercial verification product from Cadence Design Systems Inc., taking advantage on the acceleration achieved by raising the abstraction level from communication signals to transactions.

A slightly different approach has been taken by several groups [VAR2002] [BRU2002], which advocate for the explicit definition of the interface of each system module. Most of them use the Unified Modeling Language (UML) to do so, probably motivated by the its rich and expressive set of modeling constructs, as well as its success stories in the software design domain.

This case study rely on some of the previous work and oppose to some others while trying to address the following issues:

the functional specification and its implementation should be completely separated, but the relationships between them should be well defined;

the functional specification of the system may also be refined during the design process - designers must be used to moving targets - so the implementation model should be aware of such changes and keeps its consistency;

the design is driven by functionality, but its end deliverable is a structure, so there must be a clear path from functionality to implementation and the designer should be supported by the design tools along this path;

there are too many system design languages, but they share a significant amount of semantic constructs. Since the design tools will not be able to support many of those languages, a set composed by the most relevant constructs should be devised.

Before we introduce the implementation of such features as an extension to the Cave2 Framework, let's take a closer look on the design activity we intend to support. We assume a design as a set of inter-communicating entities, each of them providing functionality to the others through well defined access points. The communication among entities can be seen as the subsequent access of each others functionality through those access points. We can expect recurrent patterns of calls to the access points, as we design more complex communication between entities such as when the computation which is performed jointly involves several steps performed by each one of entities, or when a complex handshaking protocol between them is needed. Those patterns can be easily identified, and an experienced designer would like to use a set of well-known, validated solutions every time such patterns arise. So, for every communication procedure we can define a transaction: a well defined sequence of process activations by the communicating entities. Such transaction defines the roles of the participating entities by formalizing the required functionality from each of them. This definition can actually be based on the access points to the functionality rather than to the functionality itself, so that the transactions can be kept intact in case of entity implementation refinement.

We exemplify this concept using a hypothetical reconfigurable system inspired on [MIG2002].

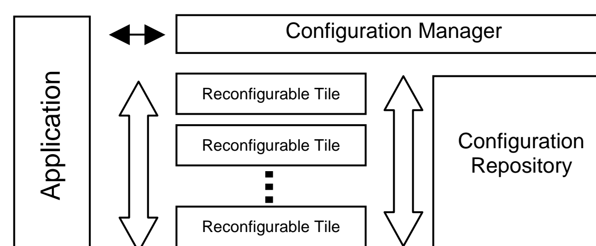


FIGURE 8.2 – Exemplifying Interface-based Design

The structure of the system we base our example on comprehends a configuration manager, which controls a set of reconfigurable tiles. According to the application's needs, the manager allocates reconfigurable tiles to implement a particular computation. The specification of such computation processes are stored as configuration information in a repository.

In Figure 8.3, a communication transaction among the system entities is depicted. Notice that among the several process calls one can outline sub-transactions that can be considered as standalone, for instance the configuration of a tile by the configuration manager, or the configuration loading request from the configuration manager to the configuration repository. In many cases, such sub-transactions can reuse the communication behavior of previously designed systems, because even though the actual system functionality differs, the communication pattern is the same. To be able to reuse the communication patterns regardless on the actual entity implementation can be considered one of the main contributions of IBD, and our approach tries to support this feature.

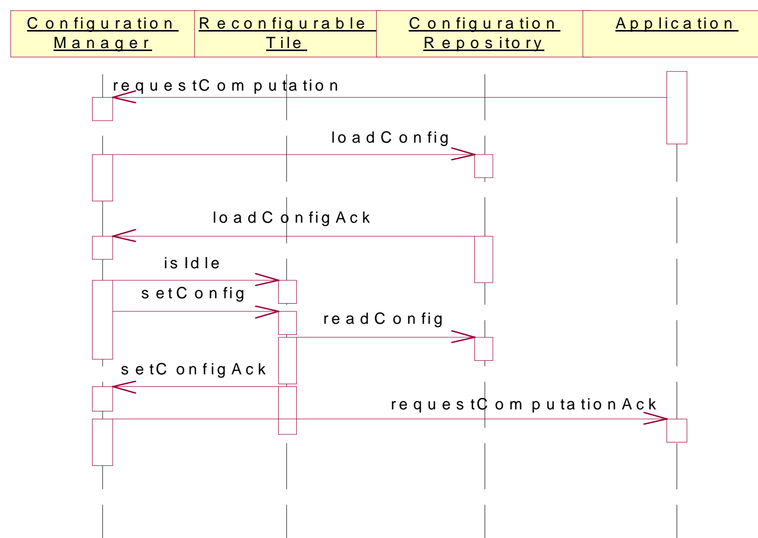


FIGURE 8.3 – Communication transaction among system parts

8.3.2 Supporting Interface-based Design

This case study is based on the corollary that a single design data model should be able to cover the functional specification and its structural counterpart, in order to grant the consistency between them. In [LEE2001] and [ZHU2002] the interface is modeled implicitly within the structural model, so the consistency is kept but both modeling activities have to be interleaved. In [VAR2002] and [BRU2002] two distinct models are usually necessary, so the functional and structural modeling can be done separately but there is the need for an external

consistency assurance. To take the best from both worlds, we decided to define a single design data model which allows for explicit interface definition. To accomplish the fourth issue presented in the previous subsection, we had to rely as much as possible in the set of semantic constructs available by most of the system design languages, so the natural choice for explicit interface definition was UML itself. But instead to propose yet another UML extension we formalized a set of associations between the UML constructs and the data model behind the Cave2 Framework, so the explicit interface definition of a system could be included in the same schema as the structural information. The resulting model is shown partially in Figure 8.4.

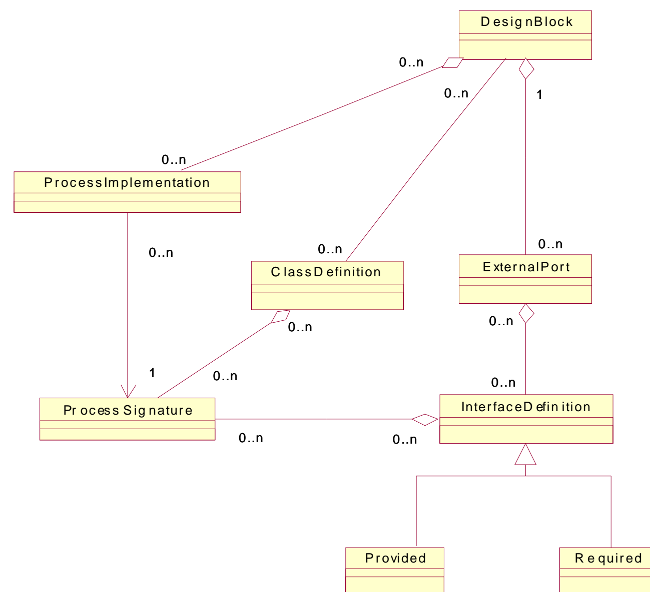


FIGURE 8.4 – Modeling structural and functional semantics

The primitive construct of the extended design data model is the design block, following the model described in subsection 5.3. It represents a design entity, such as an actor in [LEE2001] or a PE in [ZHU2002]. Its implementation is simply modeled as the set of processes it executes. Our model makes no assumption on how those processes are described, but we cover ahead in the text some strategies we used to support different description languages.

The rest of the model deals with the design block interface and most of it can be derived from UML diagrams. The signature of each process - identifier, activation method, types of the required and generated data - is explicitly defined. A set of process signatures can be grouped in a class definition or an interface definition. To demonstrate our rationale for grouping process signatures, we recall the example introduced in the previous subsection. That example shows that a design block is expected to provide and require access points to its functionality when taking part in a given communication transaction. In our model, the access points are modeled as process signatures, and they are expected to be grouped according to the transactions

they intend to participate. In the example, the configuration manager would require from the reconfigurable tile the access points for *isIdle* and *setConfig* and would have to provide *setConfigAck*.

This concept is included in our design data model as an interface definition, which is a set of process signatures that can be either provided or required by the design block for a given transaction. The assignment of required and provided interfaces of a given block is done through a port, which denotes one communication transaction.

In addition to the interface definition, we also support the concept of class definition. Unlike an interface, which aims to define the requirements for a particular communication transaction, the classes define a type for the design block. This type denotes the functionality that can be expected from this block in every transaction, thus its set of process signatures is added to the provided interfaces of every port.

For the sake of simplicity, we did not include in Figure 8.4 the inheritance relationship, which is supported in interface and class definitions, allowing for a richer functional composition model. Such relationship, however, can be seen in Figure 5.6 in the class *TypeDefinition*, which is superclass to both *ClassDefinition* and *InterfaceDefinition*.

At this point we can derive our first consistency rules for the design data model:

rule 1: for every process implemented by a design block, the respective process signature must be included either on its class definition or in the interface definition of at least one of its ports;

rule 2: for every communication transaction, all the process signatures included in the required interfaces of a participating block should be matched by a process signatures included in the provided interface of another participating block;

rule 3: every block should have one port for every communication transaction it takes part.

In Figure 8.4, the concept of transaction was not shown within the model. In the unextended Cave2 data model shown in Figure 5.6, a transaction is aggregate of ports (both instance and external). It allows a transaction to span different hierarchical levels. To extend the concept of communication transaction among hierarchical levels, we introduce the concept of internal ports (Figure 8.5). As its external counterparts, they exist solely to organize the assignment of required and provided interfaces to each transaction. The relationship between external and internal ports has a 1-n multiplicity, so we can have more than one internal port representing a

single external port. This is exactly how we can support the concept of transactions inside of hierarchical blocks.

In Figure 8.6, a hierarchical design block B1 is partially depicted. One of the external ports of B1 is shown as a white box, and close to it are its correspondent internal ports shown as box with diagonal stripes. The circles linked to each port denote the interfaces it implement: white circles are provided interfaces and gray for required interfaces. We can notice that the second consistency rule derived previously also applies for the virtual transaction between one external port and its respective internal ports.

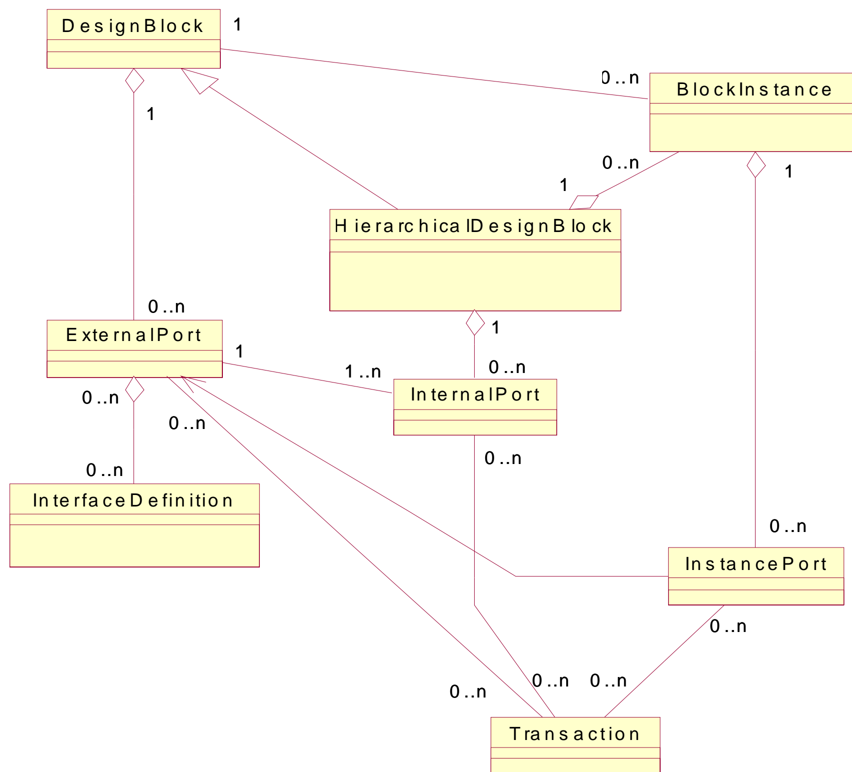


FIGURE 8.5 - Modeling structural hierarchy

Also in Figure 8.6, we can see two instances I1 and I2 which have B1 as container. The instance ports are shown as box with horizontal stripes. The circles linked to the instance ports denote the interfaces implemented by their respective external port (instance ports don't store their own interfaces, but reference an external port of the design block being instantiated). The connection between instance ports and internal ports define the partitioning of the transactions among the composing instances of a hierarchical module. In this example, we see that B1 provides the interfaces A, B and C and requires Y and Z. Its internal ports partition such interface - and thus the provided functionality - among the instances so that I1 would provide A

(through D, which inherits from A) and I2 would provide B and C through each of its ports.

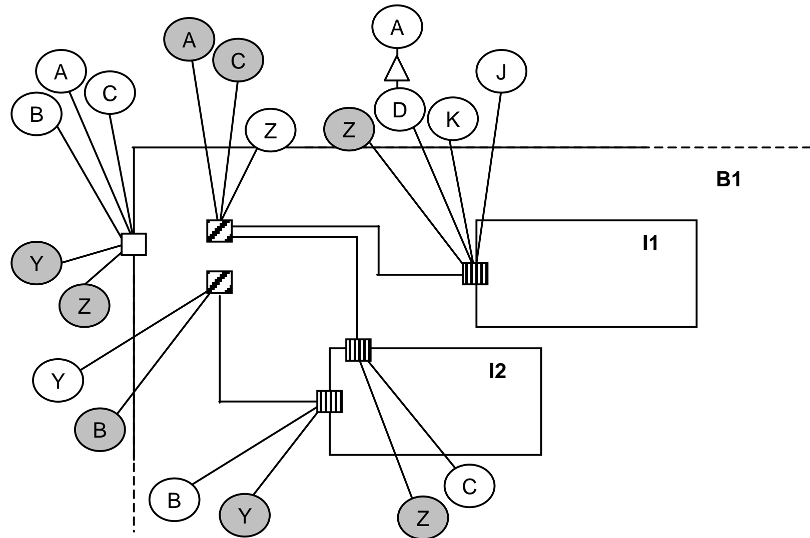


FIGURE 8.6 – Interface definitions in hierarchical blocks

The transaction definition should also store the dependency relationship among the processes called during its execution. Currently, a dependency graph is used to model such relationships.

We can at this point devise other consistency rules:

rule 4: the required interface of an internal port is an exclusive subset of the provided interface of its respective external port;

rule 5: the occurrence of transactions follow the table below, where *x* denotes the possibility of transactions and s denotes the transactions subject to rule 4.

TABLE 8.2 - Occurrence of Transactions

ports	*internal*	*external*	*instance*
internal	*x*	*s*	*x*
external	*s*	*x*	
instance	*x*		*x*

8.3.3 Implementation Issues

The extensions described in the previous subsections were implemented within the Cave2 Framework, so a design tool supporting IBD can be constructed. The consistency rules described in the previous subsection were incorporated into a subclass of the Collaboration Service, so they can be enforced within collaboration sessions. Such rules provide foundation for static validation on the communication structures, giving the designers and tools the necessary information in order to locate and correct discrepancies between the functional and structural models. This is of particular importance for the case when the functional and structural design are not being performed by the same team, as well as in the case of changes on the functional specification.

To allow more efficient collaboration, the consistency rules number 2 and 4 are not enforced at all times, so one designer can include a hierarchical block which is not completely implemented into the design repository and yet declare the intended interfaces for the external ports, so that block could be seen as complete from the outside and would be able to be instantiated by another hierarchical block.

The developed tool supporting Interface-based Design is called IBlaDe, and it is an extension to the Blade tool reported in [BRI2001, BRI2002], which is itself an extension of the framework primitive *CaveGraphicEditor*. IBlaDe inherits from Blade all its features to deal with hierarchical modeling of the structure of integrated systems. In order to allow the modeling of the function of such systems, a second diagram view was included into the tool graphic engine. Such view can be toggled on or off according to the designer's intention. When the functional description is being used, a division in the working area of the tool – called "blade edge" – is set, so both structural and functional descriptions can be seen. When the designers focus the structure only, the "blade edge" can be deactivated and IBlaDe would behave basically in the same way as its superclass. Figures 8.7 and 8.8 show snapshots of IBlaDe's GUI.

Notice that when the "blade edge" is activated, the designer can assign type definitions to each of the structural ports. Such assignment is represented by the lines crossing the "blade edge". When the structural ports are assigned to a given type definition, they are considered to implement the functionality provided by that type, granted that they can obtain the functionality required by the type definition from another port participating the transaction. In order to define the dependency graph organizing the provided and required functionality for each transaction, IBlaDE uses a GUI for editing an UML sequence diagram showing the interaction between all the types taking part in a given transaction.

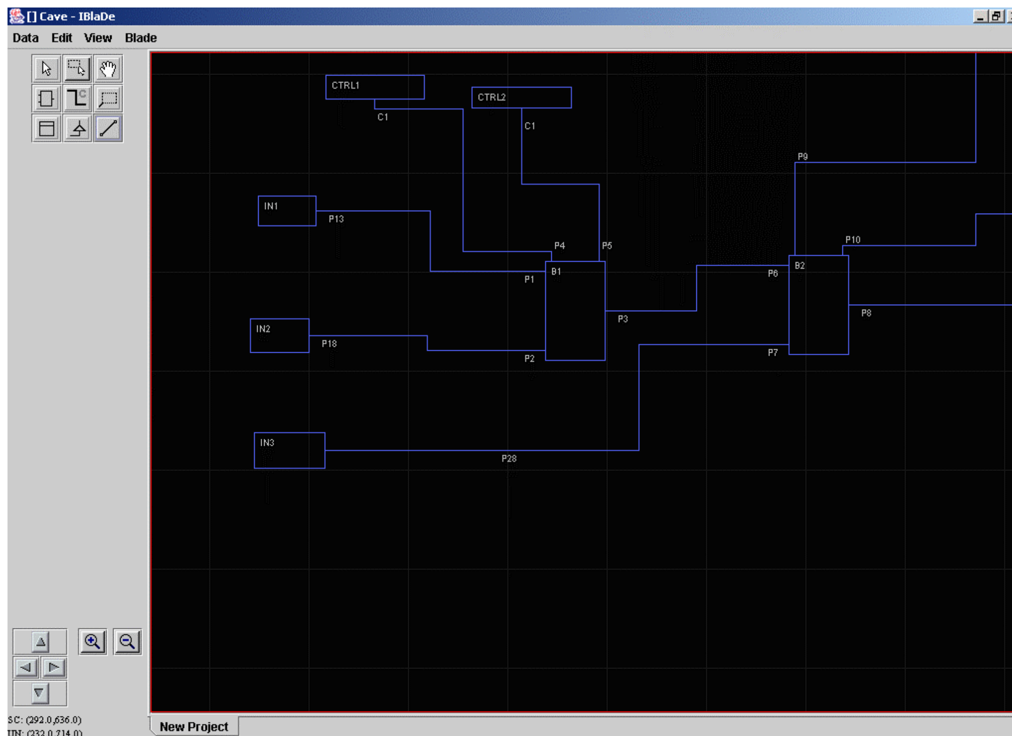


FIGURE 8.7 – IBlaDE GUI snapshot - structural view

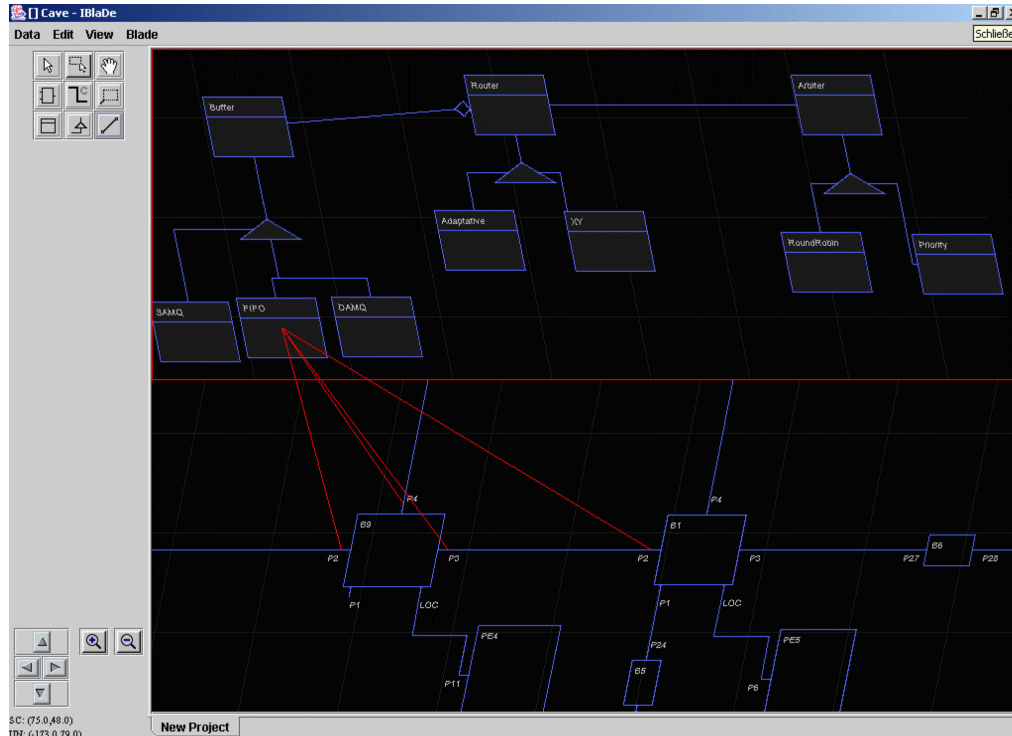


FIGURE 8.8 – IBlaDE GUI snapshot - structural and functional view

8.4 Educational Metadata

As in every discipline, the authoring of online educational material for microelectronics design is costly and requires expertise that the educators/authors may not have. Furthermore, such costs are easily underestimated because it's not easy to foresee the wide range of tasks involved in the online content creation activity: graphics, user interface, navigational structure, data storage and access control, besides the technical content itself.

This case study presents an approach aimed to reduce such authoring costs by taking advantage on design documentation content which can be available in design databases. While one should not expect to find important theoretical concepts and foundations in a design database, good design practices and real-world solutions are likely to be found. Such material can be highly valuable when integrated to online training material.

At this point, it is important to differentiate the definitions of online learning and online training we use in this case study. Online learning comprehends the use of a networked system to support students on learning fundamental subjects. Online training supports the applicability of such concepts – and perhaps some new concepts derived from the first ones – through practical activities. In the specific case of microelectronics design, online learning should provide the basics concepts – algorithms, models of computation, languages, methodologies – while online training should provide practical design activities performed by students or trainees already proficient in microelectronic design fundamentals.

Due to the nature of the material we can be reuse from the design documentation available in design databases, we restrict our approach to online training only. The expected benefits are still significant, because the online training can be applied right after the learning activity (either online or in-class) or later on, as support for continuous education and knowledge recycling.

Design documentation has always been regarded as a critical part of the design overall activity. Nowadays, the need for reuse of intellectual property has increased even more the importance of the design documentation, as the reusability of an IP core depends heavily on the quality and accuracy of its documentation. However, the design documentation is often done after the correspondent design activity, with a different set of tools and sometimes even with a different team. In order to grant the reusability of the design documentation for training purposes, an architecture for the integration of the design data model and its documentation should be devised. This case study uses the Cave2 design data model – particularly its extensions supporting design metadata (subsection 7.5).

The design automation tools which are built over the Cave2 Framework can use the metadata models to store non-structured data that complement

the actual design data, such as design constraints, test data and design documentation. As the metadata model is completely integrated with the design data model, it can be queried in the same way, so actual parameters from a given design block – such as a window size of a Viterbi decoder – can be used to query the documentation units associated to it. To allow the access of such content in an online training course, we developed a bridge between the Cave2 repository and the WWW environment. Such bridge allows the metadata information to be extracted from the design repository through an HTTP request, as it would happen in a WWW server.

While the bridge between the Cave2 design repository and the WWW environment allowed the access of design documentation material during online training activities, it didn't do very much to improve its reusability. The following issues were still prevented the wide reusability of the design documentation as training support material:

- the Cave2 repository supports queries against its design data model, but in many cases educational material should be queried against features which are specific to the educational domain, such as resource type, typical learning time, language, etc. [LTS2002];
- the creation of hierarchical composition of existing metadata blocks should be supported, in order to allow the reuse of coarse-grained blocks of content. For instance, an exercise could be defined as the composition of the documentation of the design blocks it uses, plus some instantiation guidelines extracted from designs which have used those blocks before. Such aggregate is likely to be used in more than a single training session, so the aggregation structure should be stored somehow.

To address such issues, we adopted the Learning Object Metadata standard (LOM), proposed by the IEEE Learning Technology Standards Committee (LTSC) [LTS2002]. Such standard provides guidelines to create metadata information for learning modules. By including LOM functionality in the Cave2 data model, we were able to export LOM descriptions – either as Java objects or specially formatted XML files – describing the documentation content available in Cave2 design data repository. Furthermore, with LOM we are able to specify the aggregation of documentation blocks by specifying the relationship among their LOM representations (there is a specific entry within the standard where such relationships are defined).

All LOM descriptions files are stored in a different repository, with a different query interface, so they can be queried against their educational content according to the LOM standard. Each LOM description has a reference – in the form of a URL or URI – to its actual learning object. Such information is critical when the training material is rendered to the trainee/student, because the metadata provide only the training organization. The actual content is still stored in the Cave2 design data repository. Our implementation of the reference relies on the HTTP bridge we

implemented initially, so a design repository query can be embedded in the URL which is included in every LOM description.

To validate our approach, we used a LOM-based authoring system to built training material by reusing design documentation. The validation used the Course Editor [HÖR2001] (Figure 8.9), an authoring tool designed to construct multimedia-based hierarchical structures (course modules) from existing learning objects stored in a database. Such tool was originally built to support the k-MED environment [KME2003], a knowledge based learning system for medical students, but it is generic enough to be applied to other LOM-based approaches.

In Course Editor, a course module is represented by a treelike structure to which a set of components are connected according to a relationship of IsPartOf-HasPart type, specified in the LOM standard draft. The system uses an extension to the LOM standard defined in [HÖR2001], which specifies six different aggregation levels, according to the granularity of the learning object. Only two of those, Atom and Subatom, describe actual data and point to physical resources, while one, Collection of Subatoms, is used to organize Subatoms to be presented stand-alone. The other three levels - course, chapter and collection of atoms, are abstractions conceived to organize components in a course that can be presented linearly to learners. Such components have in common their coarse granularity, and are the actual product of the Course Editor.

The complete subsystem was implemented as shown in Figure 8.10, providing support from the complete process: content is generated by designers – in the form of documentation, which is exported through the LOM interface – and authors; the authors organize the training in modules and sessions using LOM collections; trainees access the training sessions using a viewer application (e.g. a web browser), which load the training content using HTTP according to the definitions in the LOM collections.

With such approach, we contributed to the reduction of training material authoring costs by allowing the extraction of documentation content from design databases, as well as its reusability by using the LOM standard. While our main focus is on training, we foresee the application possibilities in online learning as well, granted that learning material on fundamental concepts is included.

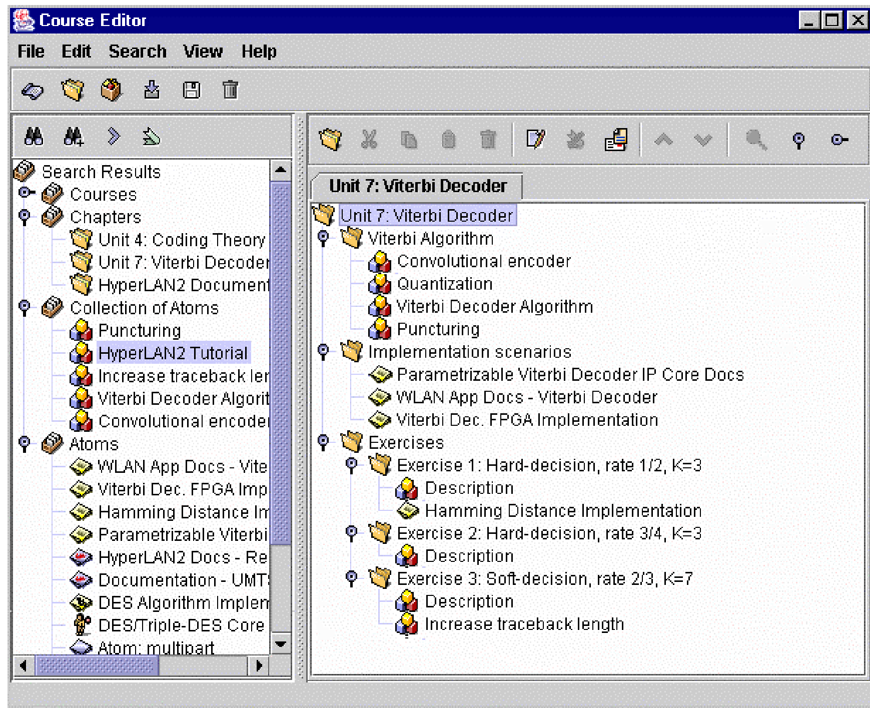


FIGURE 8.9 – Course Editor GUI

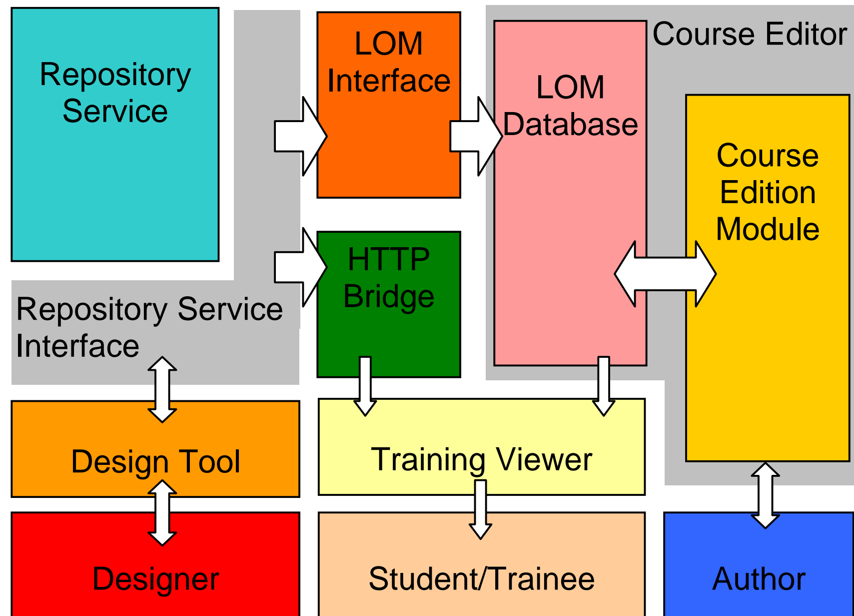


FIGURE 8.10 – Case Study: Metadata as training material

9 Conclusions and Future Work

9.1 Conclusions

The thesis work described in this text is built on top of a heterogeneous set of disciplines, such as CAD for Microelectronics, Software Engineering, Database Systems, Distributed Systems and CSCW. Its contributions are also relevant within the borders of several of those disciplines. The resulting product of this thesis work – the Cave2 Framework – implemented and validated the new concepts and ideas which provided the necessary extensions to the state-of-the-art in the forementioned disciplines, in order to make possible the collaborative design of integrated systems in a distributed environment. Being an enabling technology, the Cave2 Framework cannot be completely evaluated alone. Just like a programming language or a development methodology, the actual contributions of both the CAD Framework and the OO framework parts of Cave2 can only be fully appreciated on its extensions and applications.

The following subsections detail the most significant contributions in their respective technical areas and summarize the presented work with a comparison with the previous approaches.

9.1.1 CAD Frameworks

The classical approach on CAD Frameworks was based on the visionary structure of layers of services on top of a well defined data model. The underlying infrastructure of those Frameworks was also standardized, so that the upper service layers and the data model could migrate easily from one hardware platform to another. The approach presented in this text revisited the classical approach and applied on it the recent techniques developed in software engineering and platform-independent computing to overcome many of the limitations found in the past.

The classical CAD Frameworks were built on top of immutable data models, as the integration between the design tools depended on the strong standardization of those models. The balance between flexibility and standardization was not easy to be achieved, and the lack of one or another are often referred as reason for the commercial failure of the CAD Framework concept.

In the present approach, such problem was solved with the use of the concept of object-oriented framework. Such concept advocates for the abstract definition of recurrent patterns on the communication behavior between elements in the data model. While defined in an abstract level, those patterns are also propagated

through inheritance to the concrete extensions of the data model. By using OO frameworks, we could achieve a significant degree of standardization by defining semantic dependencies between the data model elements in an abstract level. The flexibility of the data model is also granted by its extensibility, which allows successive concretizations of the relations specified in the abstract level. In other words, the proposed design data model is only a starting point, allowing for extension and specialization, yet keeping in those extensions all the compatibility with the CAD Framework services.

Apart from that, the usage of an OO framework also brought other novelties to the CAD Framework development. This is mainly because of the application of design patterns - well known architectural solutions validated in other application domains which were not thoroughly explored in previous implementations of CAD Frameworks. The proposed implementation of the Observer pattern is one of such cases, allowing for a clear definition of the design data model and its visualization possibilities inspired in the classic MVC framework. Unlike previous approaches which delegated the design visualization techniques to the tool developer, the proposed framework provides a full set of design visualization primitives, together with a flexible set of control protocols which manage the consistency between the design data objects and their respective views. Such feature was critical to the successful implementation of the support for collaborative design. The usage of the Proxy pattern is another example, bringing significant advantages to the implementation of transparent distribution of CAD resources, as implemented in the Cave2 Service Space (Section 6). Independently from the proposed approach, other research groups have also concurrently developed proxy-based resource distribution and reported similar experiences [KOS2003, MUR2003, MUR2003a], which gives additional hints on the appropriateness of the solution.

A common assumption among framework designers states that the best way to learn how to build an object-oriented framework is by using a previously built one. Following that advice, the development of the OO framework in Cave2 was strongly influenced by the some of the frameworks within JFC. Many elements of the Cave2 Framework, such as data structures and GUI modules, are extensions of the JFC classes. In the case of GUI elements, another link can be noticed, as the JFC also implements the classic MVC framework. Yet, during the development of the framework extensions – such as the Prototyping Service and IBlaDe, reported in subsections 8.2 and 8.3 – we could experience the difficulties reported by other OO framework developers [BOC99] regarding the trade-off between usability and reusability. In many cases, it is very hard to implement the desired functionality while constrained by the reusability guidelines. In IBlaDe, for instance, we had to temporarily bypass some of the framework guidelines by using harmful programming practices during the tool development stage, and then rework part of the framework in order to accommodate some peculiarities which were needed by its GUI featuring two independent-yet-linked views of the design data ("blade edge" activation).

Another contribution of the present approach to the CAD Frameworks field is its extensive exploration of the platform-independent capabilities of the Java

technology. In former Framework implementations, several approaches on the underlying infrastructure were presented, in order to separate the Framework services from the underlying hardware and software platforms. None of them had a widely adopted solution such as the Java Virtual Machine, which has been already ported to all relevant hardware and software platforms. The issues on performance, which often arise when it comes to Java criticism, are not so relevant when compared to previous solutions on platform independence for CAD Frameworks, often based on scripting languages. Furthermore, the recent advances on JVM performance and the possibility of native code integration using JNI can make the gap between Java-based solutions and native code implementations even smaller, as reported in subsection 4.5.

9.1.2 Design Databases

The choice of the design database was a critical issue when developing a CAD Framework. First of all, the database should be able to support all the modeling constructs found in the design data model defined by the CAD Framework. Then, the database should provide all the facilities for consistency control for multi-user access, meet performance constraints, satisfy compatibility issues with the rest of the system, etc. All those constraints often led to the development of databases which were specifically tailored for a given CAD Framework. While such developments lead to important advances in the database domain, the dependency relationship that was created between the Framework and the database implementation was not beneficial. In order to avoid such dependency, several API-based approaches were proposed, some of them covered within subsection 6.3.1. Those approaches advocated for the abstraction of the database through a set of externally-accessible operations which can be mapped to manage different database implementations. API-based approaches are currently being introduced as industry standards for design databases.

The research work reported on this text included extensive investigation on databases and their suitability as design data repositories. Several commercial products, open-source packages and research prototypes were analyzed, in order to experiment on how much is the Framework influenced by the database modeling strategy. The results of our experiments showed that a significant amount of work is required in order to conjugate the data handling within the application domain with the data management within the database domain. In other words, the CAD tool developer should be aware of both the data models used within the tool and the query languages to interact with the persistent data in the database. The API-based approach reduces the problem but does not eliminate it, as the API can be seen as just another query language, database-independent though.

In order to allow the CAD developers to focus only in the tool application domain, we extended the previous approach by introducing an architecture inspired in the transparent persistency concept. The proposed extension, included in the core of the implemented object-oriented framework, allows for the direct management of the data objects through their own API instead of using a database

API to do that. This means that the CAD designer can create the complete tool as a stand-alone application, dealing only with its internal data models in memory, and then migrate it to the proposed design database architecture with little effort or even automatically. All the additional functionality regarding management of data keys, data consistency rules, distributed storage and retrieval is embedded in the framework.

The advantages of the proposed approach are a superset of the advantages of the API-based approach, as it provides a higher level of abstraction of database implementation details while providing the same kind of database-independence (the proposed approach also uses an API - the Service Interface - between the transparent persistence layer and the database implementation, as shown in Figure 6.8). The disadvantages of the proposed approach are the same of the API-based approach: performance degradation due to the abstraction layers overhead and because possible implementation-specific performance enhancements are transparent to the tool developer.

9.1.3 Collaborative Design

Collaboration among designers was always a peripheral concern among CAD Framework developers, as their major challenge was the integration of design tools. The work described within this text had collaborative design as a scope, thus it had to extend the CAD Framework domain until it overlapped with the CSCW domain. The architectural extensions which made this possible are already summarized in subsection 9.1.1, but there are some contributions of this thesis that are also valuable in the collaborative design domain itself.

Section 7 provided an overview on the proposed support to collaborative design. An overview on the general issues associated to collaboration in design was presented, and particular issues on the integrated systems design domain were derived. From the multi-level nature of the integrated systems design flow, the need for supporting both synchronous and asynchronous collaboration was derived. Synchronous collaboration was considered suitable support brainstorming and conceptual design and initial development steps, where a variety of professionals with distinct technical background is needed. During the implementation and validation steps the designers often work asynchronously, so this possibility should be considered as well.

For the asynchronous collaboration, the proposed framework provides the same level of support granted by previous approaches, embedding multiple planes of versioning and design metadata objects into the design data model. The advantages of the proposed approach in this regard are derived from the advantages of using an extensible object-oriented framework instead of a static data model, summarized before.

The major contributions are found on the synchronous collaboration support. Two collaboration methodologies were implemented - visually coupled and visually decoupled – by exploring the separation between design data model and its views. The visually coupled methodology defines a single instance of the design data and a single view of the data which is shared among all designers. The visually decoupled methodology deals with different visualizations for each user. The consistency between the data model and its views is enforced by update/notify mechanisms. All updates to a given data object are notified to all its views. The same concept is also used to update related data objects, for example the update of the instances of a given block when that block is updated.

The availability of both collaboration methodologies can be justified by their qualitative and quantitative differences. The visually coupled approach provides a very close collaboration between designers, and is particularly suitable for training sessions, because it allows the actions of a given designer to be watched step by step. On the other hand, the visually decoupled approach allows designers to work synchronously and yet with a certain degree of independence, so the burdens of the collaboration are not so high. Regarding the quantitative aspect, the visually decoupled approach is by far more efficient. Our experiments showed that more than 80% of the update/notify event traffic in a collaborative session are generated by visual synchronization, so the visually decoupled methodology should be the preferred choice when the underlying network infrastructure has low bandwidth.

To keep the consistency of multi-user updates and notifies, we also had to deal with concurrency control mechanisms. Thus, the update/notify events are deployed within collaborative sessions, which are isolated from each other by transactional boundaries. Such consistency control scheme avoid the interference between non-collaborating groups of designers. Within a collaborative session, the strictness of the transactional control should be avoided in order to foster collaboration, so the proposed approach allows for custom extensions for the concurrency control. Two techniques were implemented, but future extensions tailored to other collaboration methodologies can be included.

The first technique to be implemented, inspired in the Pair Programming Technique, was used to validate the visually coupled methodology. It is based on floor control, so the collaboration potential is reduced as only one designer can be fully active at a given time. The second technique, used in both visually coupled and decoupled methodologies - is an original hybrid technique, tailored to the application domain of integrated systems design. It uses event filtering to extract the semantics of every update and notifies the generator of the update immediately in the case of order-insensitive updates. In the case of order-sensitive updates, it uses a centralized controller. By relying on domain-specific filtering, it improves significantly the responsiveness of the interface with the designer.

9.1.4 Summary

The work presented in this text revisited the CAD Framework concept to address the need for computational infrastructure supporting collaboration among designers in a distributed environment. A number of engineering techniques were employed to accomplish this goal, some of them in novel ways, as reported in the previous subsection. The validation of those techniques was done during the implementation steps of the Cave2 Framework and its extensions. Table 9.1 shows the comparison of the resulting Framework with the previous work reviewed in Section 3.

TABLE 9.1 – Comparison between CAD systems supporting distributed, multi-user design of integrated systems

Tool	Supports methodology management using workflow techniques	Supports design data versioning	Abstracts the complexity of the distribution of CAD resources over networks	Platform independent	Provides extensible design data modeling constructs
Nelsis	X	X			
Version Server		X			
STAR		X			
Ulysses & Odyssey	X	X			
WELD	X		X	X	
OmniFlow	X		X	X	
ASTAI(R)	X	X	X		
Moscito	X		X	X	
PPP			X	X	
JavaCAD			X	X	
Ptolemy II				X	X
Cave			X	X	
Cave2	*	X	X	X	X

Besides the features shown in Table 9.1, the developed framework is the first of its kind to provide support to synchronous collaboration.

The advances achieved on the development of Cave2 are expected to support the development of front-end tools tailored to the novel design methodologies, where the collaboration between designers with different expertises is needed. The system description used in such tools has a high abstraction level and is often a mixture of textual and diagramatic representations, matching exactly the target scenario of Cave2.

However, the developed CAD Framework is not of production quality and should be considered only a reference implementation. Yet the Framework is

suitable to support academic or even industrial prototypes, is the software architecture validated by Cave2 that should be considered the major contribution of this thesis.

9.2 Future Work

Among the collaboration support techniques found in CAD Frameworks, the only on which was not completely implemented within the Cave2 Framework – marked with a star in Table 9.1 – is the support for workflow modeling and execution. While the Service Space architecture provides for a service encapsulation strategy, there is no specific workflow modeling and execution engine within Cave2. This is because of the evident maturity on this research area within EDA where significant advances were achieved recently, as shown in tools like TRMS/GTLS [KOS2003], OmniFlow and MOSCITO has shown. Each of those tools has its own tool encapsulation and activation strategies, but with little effort they can be adapted to use their workflow modeling and execution engines on top of Cave2 proxy-based encapsulation architecture.

Yet regarding service encapsulation, future research is also needed to address the integration of the proxy-based encapsulation and reflection techniques [SUN97]. In such case, both the distribution of the services and their API can be partly abstracted. We have shown that the first technique allows for locally accessible modules - the proxies - which can hide the real location of the core of a design tool that can be installed in a different machine in the network. The second technique would allow the implementation of service proxies which are self-adaptable, thus able to inspect during execution time the interface of the proxies of other services in order to set up an optimized protocol for design data exchange.

Another possible extension would be the inclusion or integration of a functional simulation environment into the Service Space. Currently, the functional simulation is performed without a simulation backbone, so the interoperability between the simulating modules is achieved by ad-hoc agreement between the designers. No formal definition on models of computation or data abstraction is provided. A foreseeable starting point would be the Ptolemy II system, which includes an object-oriented framework for modeling and simulation of embedded systems. Ptolemy II is also developed on top of the Java technology, so the integration is made easier both because the conceptual and implementation-level compatibility.

Finally, there are several issues that were addressed only in its surface during the development work of this thesis, and that should be investigated further in the future development. For instance, the switching between synchronous and asynchronous modes of collaboration, or the switching between visually coupled and visually decoupled collaboration, can be better explored. The major concern in this case is the synchronization of different views, and many approaches can be offered, such as automatic view generation, progressive user-driven synchronization through step-by-step agreement, or even the complete override of one designer's view by the

view of a colleague. Another example is on the event propagation between the tool and the Repository and Collaboration Services. There is plenty room for optimizations, specially among visual-only events: strongly correlated events can be combined, such as the creation of a block will usually be followed by the creation of ports, or the numerous events generated by dragging a given object can be reduced to a smaller yet equivalent subset.

Appendix 1 Cave Development Timeline

1996 Preliminary studies on platform-independent design tools
First implementations on hypermedia support for design environments

Related publications:
INDRUSIAK, L. S., REIS, R. A. L. A World Wide Web Based Microelectronics Tutorial In: XI UFRGS Microelectronics Seminar, 1996, Porto Alegre.
INDRUSIAK, L. S., REIS, R. A. L. Microelectronics Learning Using WWW and VRML In: Workshop on Multimedia and Virtual Worlds (IFIP WG 9.5), Intenational IFIP 9.4 Conference, 1997, Florianópolis, SC.
INDRUSIAK, L. S., REIS, R. A. L. Microelectronics Education using WWW In: 1999 International Conference on Microelectronic Systems Education, 1999, Arlington.
Proceedings. Los Alamitos: IEEE Computer Society Press, 1999. p.43 - 44
INDRUSIAK, L. S., REIS, R. A. L. Microelectronics Education Using WWW and CAD Tools In: XI Brazilian Symposium on Integrated Circuits Design, 1998, Armação de Búzios,RJ. Proceedings. Los Alamitos: IEEE Computer Society, 1998. p.31 - 36

1997 First approach on WWW-based tool integration
Hyperdocument-based user interface adopted
Encapsulation of foreign tools using CGI
Development of an object-oriented library of reusable tool blocks was started
Implementation of CIF2VRML tool - 3D Visualization of layout data using VRML

Related publications:
INDRUSIAK, L. S., REIS, R. A. L. Visualização 3d do Layout de Circuitos Integrados Utilizando Vrml In: I Workshop de Realidade Virtual, 1997, São Carlos, SP. p.177 - 186
INDRUSIAK, L. S., REIS, R. A. L., GRALEWSKI, D. D., BRONDANI, C., BRASCO, F. F. Ambiente de Concepção de Circuitos Integrados Baseado Em Www In: IX Salão de Iniciação Científica, 1997, Porto Alegre, RS.
INDRUSIAK, L. S. Ambiente de Apoio Ao Projeto de Circuitos Integrados Utilizando World Wide WebIn: II Semana Acadêmica do CPGCC/UFRGS, 1997, Porto Alegre.
INDRUSIAK, L. S., REIS, R. A. L. A Www Approach For Eda Tool Integration In: X Brazilian Symposium on Integrated Circuits Design, 1997, Gramado, RS.
INDRUSIAK, L. S., REIS, R. A. L. A Www Approach For Eda Tool Integration In: XII UFRGS Microelectronics Seminar, 1997, Porto Alegre, RS.
INDRUSIAK, L. S., REIS, R. A. L. 3D Circuit Layout Visualization Using VRML In: XII UFRGS Microelectronics Seminar, 1997, Porto Alegre, RS. p.77 - 80

1998 Refinement on the WWW-based tool integration architecture
Hyperdocument-based design flow modeling
Implementation of first prototypes of design entry and visualization tools: Jale (layout) and Jase (schematic)
Encapsulation of foreign tools using Servlets
Designer coordination and communication tool - Cadena - is implemented

Related publications:
INDRUSIAK, L. S., GRALEWSKI, D. D., REIS, R. A. L. Building Server Side Applications With Java In: XIII SIM - Microelectronics Seminar, 1998, Bento Gonçalves, RS.
INDRUSIAK, L. S., BRONDANI, C., BRASCO, F. F., REIS, R. A. L. Graphic Schemes Development To Jale In: XIII SIM - Microelectronics Seminar, 1998, Bento Gonçalves, RS.
INDRUSIAK, L. S., REIS, R. A. L. A Case Study For The Cave Project In: XI Brazilian Symposium on Integrated Circuits Design, 1998, Armação de Búzios, RJ. Los Alamitos: IEEE Computer Society, 1998.
INDRUSIAK, L. S., REIS, R. A. L. Project Management and Design Methodology Support for the Cave Project: A Hyperdocument-Centric Approach In: XII Brazilian Symposium on Integrated Circuits Design, 1999, Natal, RN. Los Alamitos: IEEE

Computer Society Press, 1999.
INDRUSIAK, L. S., WINCKLER, M. A. A., GRALEWSKI, D. D., REIS, R. A. L. JALE - JAVA Layout Editor In: XIV SIM Microelectronics Seminar, 1999, Pelotas, RS.
HERNANDEZ, É. B., SAWICKI, S., INDRUSIAK, L. S., REIS, R. A. L. WWW as an Environment to IC Project: The JASE Tool (Java Schematic Editor) In: XV Microelectronics Seminar, 2000, Torres, RS, Brasil.

1999 Cave is chosen as infrastructure for the DInCAD Cooperation Project (Brazil-Germany)
Architectural upgrades were defined, in order to support DInCAD goals (collaborative design and learning)
Transitional architecture implementation, in order to allow migration from hyperdocument-based model to a fully object-oriented model
CIF2VRML upgrade to generate 3D models with visual simulation capabilities
A distributed database scheme for VHDL metadata is implemented
Cave is presented in the DATE (Munich) and DAC (New Orleans) University Booth

Related publications:
INDRUSIAK, L. S., OST, L. C., REIS, R. A. L. Visualização 3D de Circuitos Integrados usando Modelos VRML In: IV Simpósio Nacional de Informática, 1999, Santa Maria, RS.
INDRUSIAK, L. S., OST, L. C., REIS, R. A. L. Dynamic 3D Models of Integrated Circuits Using VRML In: II Workshop Brasileiro de Realidade Virtual, 1999, Marília, SP. Anais. São Carlos, SP: UFSCar, 1999. p.95 - 101
INDRUSIAK, L. S., REIS, R. A. L. 3D Integrated Circuits Layout Visualization using VRML In: Winter Simulation Conference - International Conference on Web-based Modeling and Simulation (WEBSIM'99), 1999, San Francisco, CA, USA. Proceedings. San Diego: Society for Computer Simulation International, 1999. p.177 - 181
INDRUSIAK, L. S., REIS, R. A. L., BECKER, J., GLESNER, M. DInCAD: Distributed Internet-based CAD Methods for Future Complex Microelectronic Systems In: V Workshop of the German-Brazilian Bilateral Programme for Scientific and Technological Cooperation through BMBF and CNPq, 1999, Koenigswinter.
INDRUSIAK, L. S., REIS, R. A. L. From a Hyperdocument-Centric to an Object-Oriented Approach for the Cave Project In: XIII SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN - SBCCI '2000, 2000, Manaus. Proceedings. Los Alamitos: IEEE Computer Society, 2000. p.125 - 130
INDRUSIAK, L. S., PALMA, J. C. S., BRUNONI, J. L., MACHADO, P. B. Sistema de Indexação Distribuída de Descrições VHDL de Circuitos Integrados usando Interface WWW In: IV Simpósio Nacional de Informática, 1999, Santa Maria, RS.
INDRUSIAK, L. S., REIS, R. A. L. 3D integrated circuit layout visualization using VRML. Future Generation Computer Systems. Amsterdam: , v.17, n.5, p.503 - 511, 2001.

2000 Implementation of the architectural upgrades - new version of the Cave environment, fully object-oriented is named Cave2
FPGA platforms are integrated to Cave using the CaveJTAG tool
Tropic layout generation tool is integrated to Cave through the WTropic tool
CIF2VRML and Jale tools are merged into Jale3D, incorporating capabilities for edition and visualization of MEMS
First implementation of Homero, a tool for textual design entry
Cave2 early prototype is presented in the DAC University Booth (Los Angeles)

Related publications:
INDRUSIAK, L.S. Architectural Evolution for the Cave Design Automation Framework. Trabalho Individual. Porto Alegre: PPGC UFRGS, 2000. 46 p.
FRAGOSO, J.L.; MORAES, F.; REIS, R. WTROPIC: A WWW-Based Macro-Cell Generator. In: XIII SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN - SBCCI '2000, 2000, Manaus. Proceedings. Los Alamitos: IEEE Computer Society, 2000.
BECKER, J., MAYER, U., GLESNER, M., INDRUSIAK, L. S., REIS, R. A. L. Providing Flexible Internet Infrastructure for FPGA-Based CAD Courses In: EWME 2000 - European Workshop on Microelectronics Education, 2000, Aix en Provence.
OST, L. C., MAINARDI, M. L., INDRUSIAK, L. S., REIS, R. A. L. Jale3D - Platform-independent IC/MEMS Layout Edition Tool In: 14th Symposium on Integrated Circuits

and Systems Design, 2001, Pirenopolis. Proceedings. Los Alamitos: IEEE Computer Society, 2001. p.174 - 179
INDRUSIAK, L. S., REIS, R. A. L. Using Cave Environment for Remote FPGA Programming In: XV Microelectronics Seminar, 2000, Torres, RS, Brasil.
BECKER, J., GLESNER, M., MAYER, U., HOLLSTEIN, T., INDRUSIAK, L. S., REIS, R. A. L. An Internet-Capable CAD Suite for the Mulit-Level Design of Complex Microelectronic Systems In: DATE 2000 - Design, Automation and Test In Europe Conference, 2000, Paris. Proc. of Design, Automation and Test In Europe Conference - User Forum. , 2000. p.303
INDRUSIAK, L. S., HERNANDEZ, É. B., SAWICKI, S., REIS, R. A. L. Homero - Um editor VHDL Cooperativo via Web In: IWS '2001 - VII Workshop Iberchip, 2001, Montevideo.

2001 Definition of the architecture for collaborative design support
Jase is replaced by BLADE, a tool for hierarchical description using diagrams
Homero is used as a case study for a Pair Programming related collaboration methodology - preliminary results presented at DAC (Las Vegas) and DATE (Munich) University Booth
The Cave data model is significantly extended and upgraded, separating visual and semantic concerns in order to allow efficient synchronous and asynchronous collaboration
Two modes for synchronous collaboration are defined: visually coupled and visually decoupled
The first specification of the Service Space was derived
Paper reporting the experiences on collaborative design on top of Cave2 receives "Outstanding Paper Award" at SBCCI

Related publications:
INDRUSIAK, L. S., REIS, R. A. L., GLESNER, M. Collaborative Learning by Sharing Design Experience In: 4th European Workshop on Microelectronics Education, 2002, Vigo.
BRISOLARA, L. B., INDRUSIAK, L. S., REIS, R. A. L. Modelagem Orientada a Objetos de Primitivas de Projeto de Sistemas Eletronicos voltada para Colaboracao In: IWS 2002 - VIII Workshop Iberchip, 2002, Guadalajara.
SAWICKI, S., INDRUSIAK, L. S., REIS, R. A. L. Projeto Cooperativo no Ambiente Cave In: IWS 2002 - VIII Workshop Iberchip, 2002, Guadalajara.
BRISOLARA, L. B., INDRUSIAK, L. S., REIS, R. A. L. Developing an Hierarchical Schematic Editor to WWW In: XVI Microelectronics Seminar, 2001, Santa Maria.
INDRUSIAK, L. S., BECKER, J., GLESNER, M., REIS, R. A. L. Distributed Collaborative Design over Cave2 Framework In: 11th IFIP International Conference on Very Large Integration, 2001, Montpellier.
INDRUSIAK, L. S., GLESNER, M., REIS, R. A. L. Comparative Analysis and Application of Data Repository Infrastructure for Collaboration-Enabled Distributed Design Environments In: DATE 2002 - Design Automation and Test in Europe, 2002, Paris. Proceedings. Los Alamitos: IEEE Computer Society, 2002.
INDRUSIAK, L. S., HERNANDEZ, É. B., SAWICKI, S., REIS, R. A. L., BECKER, J., GLESNER, M. Distributed System-Level Design Using Pair-Programming over Cave In: DATE Conference - Design Automation and Test in Europe, 2001, München. Demonstrations at the University Booth of the DATE Conference 2001. Tübingen: Wilhelm-Schickard-Institute for Computer Science, 2001. v.2001. p.26
SAWICKI, S.; BRISOLARA, L.B.; INDRUSIAK, L.S.; REIS, R.A.L. Collaborative Design using a Shared Object Spaces Infrastructure. In: Symposium of Integrated Circuits and Systems Design, SBCCI, 17., 2002, Porto Alegre. Proceedings... Los Alamitos: IEEE Computer Society Press, 2002.
BRISOLARA, L. B., INDRUSIAK, L. S., REIS, R. A. L. Blade: A Hierarchical diagram editor target to collaboration In: 17th South Microelectronics Seminar, 2002, Gramado. Proceedings. , 2002.
SAWICKI, S., BRISOLARA, L. B., INDRUSIAK, L. S., REIS, R. A. L. Collaborative Design based on Shared Object Spaces In: 17th South Microelectronics Seminar, 2002, Gramado. Proceedings. , 2002.

2002 Definition of the Repository and Collaboration Services
Extensions for educational and research usage of the remote FPGA programming were implemented, following Service Space guidelines

Jini-encapsulated FPGA Board demonstrated at DAC University Booth, New Orleans
PETRA tool for power estimation of functional models relies on Cave2 data modeling during its development step
IBlaDe tool was developed, as a case study of the Cave2 data model extension, supporting interface-based design
Transparent persistency concept is implemented within the Repository Service
Design metadata and versioning support are included in the Cave2 data model
Design metadata is reused as training material using LOM descriptions and HTTP bridging in the Repository Service

Related publications:
INDRUSIAK, L. S., LUBITZ, F., GLESNER, M., REIS, R. A. L. Ubiquitous Access to Reconfigurable Hardware: Application Scenarios And Implementation Issues In: Design Automation and Test in Europe (DATE), 2003, Munich. Proceedings. Los Alamitos: IEEE Computer Society, 2003. p.940 - 945
INDRUSIAK, L. S., BECKER, J., GLESNER, M., REIS, R. A. L. Distributed Collaborative Design over Cave2 Framework In: 11th International Conference on Very Large Scale Integration of Systems-on-Chip, 2001, Montpellier. SOC Design Methodologies. Boston: Kluwer Academic Publishers, 2002. p.97 - 108
INDRUSIAK, L.S., GLESNER, M., REIS, R., ALCANTARA, G., HOERMANN, S., STEINMETZ, R. Reducing Authoring Costs of Online Training in Microelectronics Design by Reusing Design Documentation Content. In: 2003 International Conference on Microelectronic Systems Education (MSE), 2003, Anaheim. Proceedings. Los Alamitos: IEEE Computer Society, 2003.
INDRUSIAK, L.S., GLESNER, M., REIS, R. Supporting Consistency Control on Functional and Structural Views in Interface-based Design Models. In: Forum on Specification and Design Languages (FDL), 2003, Frankfurt.
INDRUSIAK, L. S., GLESNER, M., REIS, R. Computational Infrastructure for the Collaborative Design of Integrated Systems over a Distributed Environment. In: In: E-Colleg Workshop on Challenges in Collaborative Engineering (CCE), 2003, Poznan. (to appear)

2003 Cave2 design data model used as foundation in experiments on UML-guided design space exploration
IBlaDe demonstrated at DAC University Booth, Anaheim
Design metadata reuse as training material is demonstrated at DAC University Booth, Anaheim

Appendix 2 Cave UML Class Diagrams

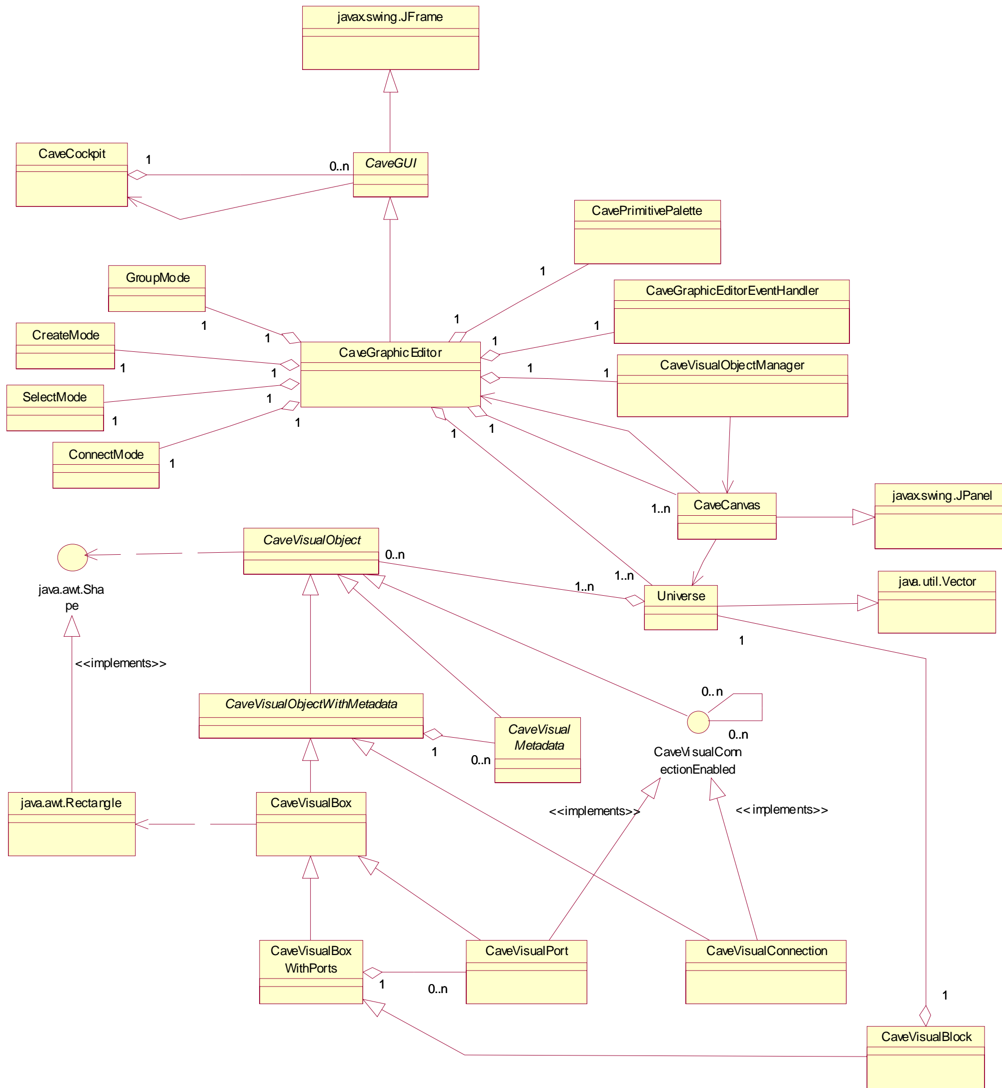


FIGURE A2.1 – Cave2 Design Tool Primitives (partial)

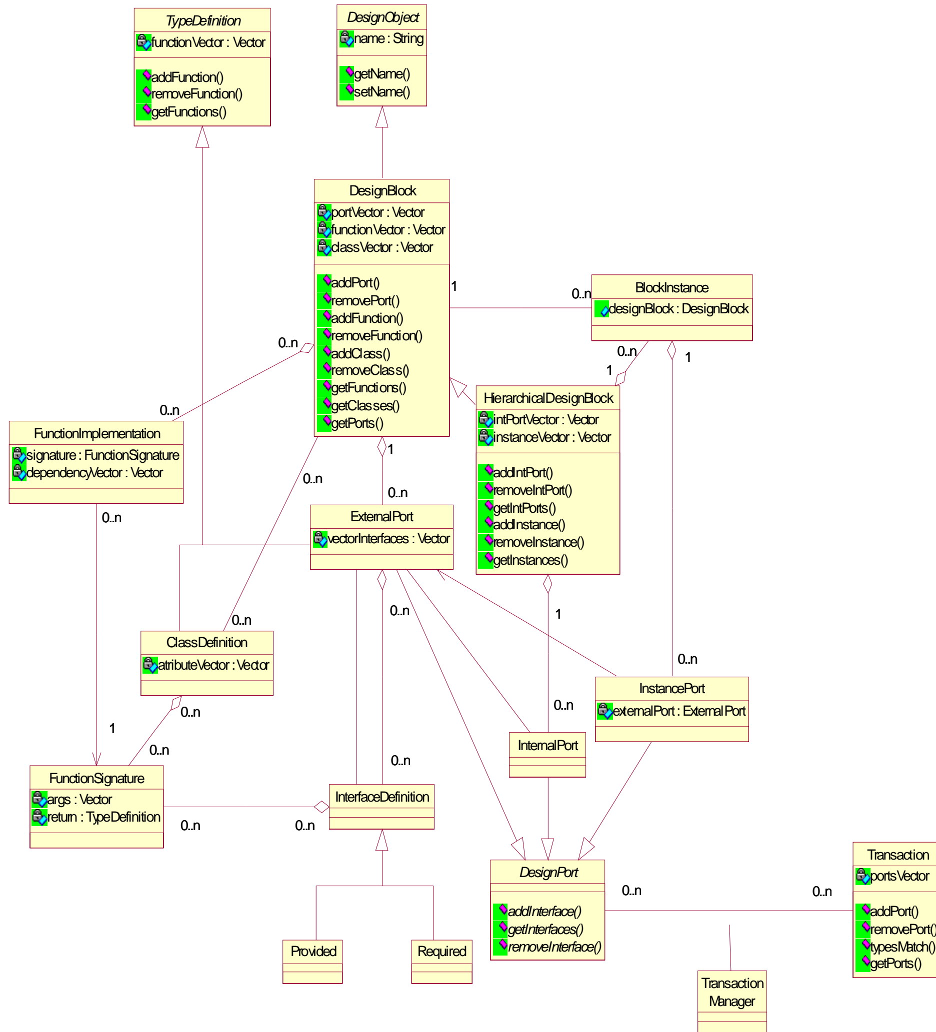


FIGURE A2.2 – Cave2 Design Data Primitives (partial)

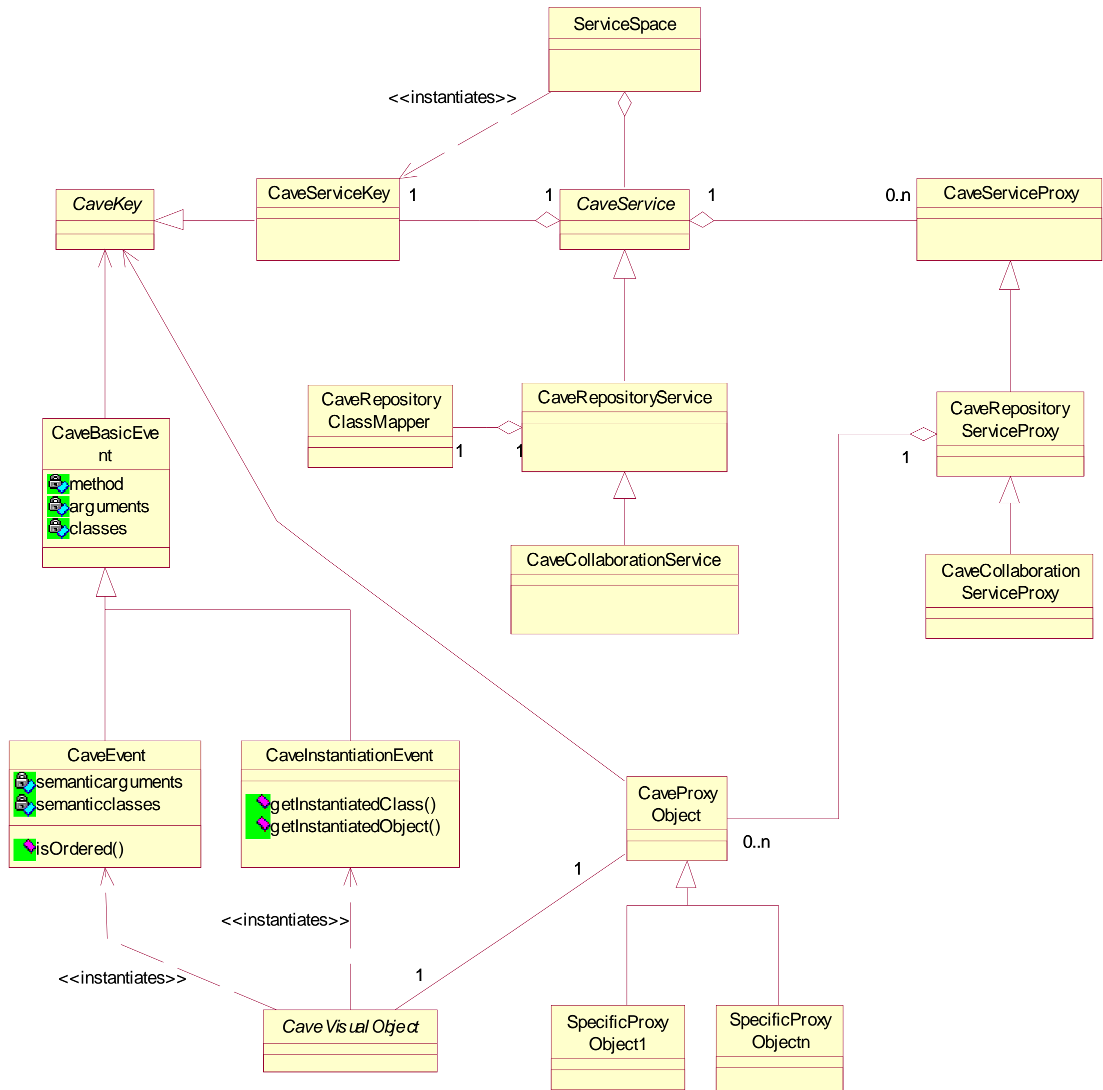


FIGURE A2.3 – Cave2 Repository and Collaboration Services (partial)

Appendix 3 Cave2 Code Statistics

Package	Classes	Functions	NCSS
.	3	18	220
cave.awt	10	79	459
cave.cif	13	314	3249
cave.collaborative.chat	5	35	278
cave.collaborative.connect	2	4	42
cave.collaborative.service	23	149	1677
cave.design	13	95	820
cave.design2	15	70	227
cave.graphic	40	318	2804
cave.graphic.graphic2D	26	266	1783
cave.io	2	6	64
cave.moo	1	7	113
cave.protocol	1	1	8
cave.repository	13	91	528
cave.server	9	16	118
cave.text	1	7	76
cave.tools	37	220	3991
cave.tools.blade	11	63	804
cave.tools.homero	11	71	707
cave.tools.iblade	14	78	953
cave.tools.iblade.seqdiag	1	1	28
cave.util	3	7	31
Total	254	1916	18980

Appendix 4 Cave2 Code Documentation

Available online at http://www.inf.ufrgs-br/~cave .

Appendix 5 Summary in Portuguese Language

Um Framework de Apoio à Colaboração no Projeto Distribuído de Sistemas Integrados

Leandro Soares Indrusiak 1,2

Ricardo A. L. Reis 1

Manfred Glesner 2

Resumo: O trabalho de pesquisa apresentado neste artigo tem por objetivo apoiar o projeto distribuído de sistemas integrados, considerando especificamente a necessidade de interação colaborativa entre os projetistas. O trabalho enfatiza particularmente alguns problemas que foram considerados apenas marginalmente em abordagens anteriores, como a abstração da distribuição em rede dos recursos de automação de projeto, a possibilidade de interação síncrona e assíncrona entre projetistas e o suporte a modelos extensíveis de dados de projeto.

Abstract: The work described in this paper aims to support the distributed design of integrated systems and considers specifically the need for collaborative interaction among designers. Particular emphasis was given to issues which were only marginally considered in previous approaches, such as the abstraction of the distribution of design automation resources over the network, the possibility of both synchronous and asynchronous interaction among designers and the support for extensible design data models.

1 Instituto de Informática, UFRGS, Caixa Postal 15064, 91501-970, Porto Alegre, RS, Brasil
{reis@inf.ufrgs.br}
2 Microelectronic Systems Institute, TU Darmstadt, Karlstr. 15, 64283 Darmstadt, Germany
{indrusiak, glesner@mes.tu-darmstadt.de}

1 Introdução

O trabalho de pesquisa apresentado neste artigo tem por objetivo apoiar o projeto de sistemas integrados em ambiente distribuído, considerando especificamente a necessidade de interação colaborativa entre os projetistas. Esta necessidade é claramente destacada em publicações como o SIA Roadmap [1] e o Medea+ Roadmap [2], que analisam os principais problemas enfrentados pela industria de semicondutores e as possíveis soluções advindas de pesquisa nos próximos 10 anos. O presente trabalho enfatiza particularmente alguns problemas que foram considerados apenas marginalmente em abordagens anteriores, como a abstração da distribuição em rede dos recursos de automação de projeto, a possibilidade de interação síncrona e assíncrona entre projetistas e o suporte a modelos extensíveis de dados de projeto.

Tais problemas requerem uma infra-estrutura de software significativamente complexa, pois possíveis soluções envolvem diversos módulos, desde interfaces com o usuário (GUIs) até bancos de dados e *middleware*. Para construir tal infra-estrutura, várias técnicas de engenharia foram empregadas e algumas soluções originais foram desenvolvidas. A idéia central da solução proposta é baseada no emprego conjunto de duas tecnologias homônimas: CAD Frameworks (ambientes integrados de apoio ao projeto) e frameworks orientados a objeto. O primeiro conceito foi criado no final da década de 80 na área de automação de projeto de sistemas eletrônicos e define uma arquitetura de software em níveis, voltada ao apoio a desenvolvedores de ferramentas de projeto, administradores de ambientes de projeto e projetistas. O segundo, desenvolvido na última década na área de engenharia de software, é um modelo para arquiteturas de software visando o desenvolvimento de sub-sistemas reusáveis de software orientado a objeto. No presente trabalho, propõe-se a criação de um framework orientado a objetos que inclui conjuntos extensíveis de primitivas de dados de projeto bem como de blocos para a construção de ferramentas de CAD. Esse framework orientado a objeto é agregado a um CAD Framework, onde ele passa a desempenhar funções tipicamente encontradas em tal ambiente, tais como representação e administração de dados de projeto, versionamento, interface com usuário, administração de projeto e integração de ferramentas.

2 Comparação com trabalhos anteriores

A interoperabilidade entre ferramentas de projeto foi um dos tópicos mais importantes da pesquisa cobertos pela área da automatização de projeto de circuitos integrados nos últimos trinta anos. Recentemente, a interoperabilidade entre projetistas passou a receber atenção, e a necessidade de técnicas específicas para suportar a comunicação, a coordenação e o compartilhamento de dados entre grupos de projetistas ficou clara [1,2]. A razão é óbvia: a complexidade do projeto de circuitos integrados está aumentando mais rápido que o previsto [3]. Observa-se também que o mercado afasta-se lentamente do paradigma baseado em computadores pessoais, passando a um cenário onde os recursos computacionais estão distribuídos entre diversos dispositivos de menor porte. Estes dispositivos oferecem ao usuário possibilidades mais simplificadas de operação – essa é justamente uma das vantagens do novo paradigma – mas por outro lado a complexidade de projeto de tais dispositivos é ainda elevada. Essa complexidade – que pode envolver subsistemas digitais, analógicos, óticos e eletromecânicos, bem como as interfaces de programação de tais subsistemas – só pode ser dominada por grupos de projetistas trabalhando colaborativamente na busca de soluções dos problemas de projeto.

A pesquisa na área de projeto colaborativo pode ser considerada uma combinação dos esforços de pesquisa em CAD (projeto assistido por computador) e CSCW (trabalho colaborativo assistido por computador). Ambas áreas já incorporam uma quantidade significativa de conhecimento. A pesquisa inter-disciplinar envolvendo ambas áreas também já atingiu certa maturidade, especialmente nas áreas de apoio ao projeto de engenharia mecânica e civil. Na área abordada no presente trabalho – projeto de sistemas integrados de hardware e software, a pesquisa em projeto colaborativo é ainda incipiente, e boa parte dos trabalhos disponíveis na literatura foram analisados em [4].

As abordagens visando apoiar projeto colaborativo podem ser caracterizadas de acordo com a taxonomia de tempo e espaço utilizada na área de CSCW [5] (Tabela 1). Considerando o caso onde equipes de projetistas estão distribuídas geograficamente, assumimos que a infraestrutura de apoio ao projeto colaborativo deva contemplar os tipos de colaboração mostrados na segunda linha da tabela.

Tabela 1. Taxonomia tempo-espaco para sistemas CSCW

	mesmo tempo	tempos distintos
Mesmo espaço	Interação face-a-face	Interação assíncrona
Espaços distintos	Interação Síncrona distribuída	interação Assíncrona distribuída

Trabalhos anteriores relativos à área de projeto colaborativo contemplavam principalmente processos de colaboração assíncrona. Contribuições de Katz [6], Harrison [7] e Wagner [8] na pesquisa e desenvolvimento de modelos de dados de projeto suportando versões possibilitaram um aumento da eficiência nos casos onde grupos de projetistas trabalham concorrentemente de forma assíncrona. Contribuições em gerência de metodologias de projeto e modelagem de fluxo de projeto – tais como de Brglez [9] e Schneider [10] – também são relevantes, uma vez que permitem que os líderes de grupos de projetistas definam e disponibilizem a suas equipes um conjunto de regras e procedimentos a serem seguidos durante o processo de projeto.

A abordagem descrita no presente trabalho difere dos trabalhos anteriores por (1) prover suporte tanto à colaboração síncrona quanto assíncrona, bem como por (2) definir explicitamente uma separação de domínios entre o modelo da semântica de projeto e sua representação visual.

A primeira característica é de particular importância para a automação de projeto de sistemas integrados, pois a colaboração síncrona é necessária nas etapas iniciais do projeto enquanto a colaboração assíncrona ocorre nas etapas finais, durante a implementação e verificação do sistema sendo projetado. Por exemplo, um alto potencial de colaboração pode ser identificado nos primeiros passos de processo de projeto, quando a funcionalidade e os requisitos técnicos do produto são definidos, justamente devido a multi-disciplinaridade de tais atividades. Engenheiros de hardware, programadores, gerentes de marketing e de produto são alguns dos profissionais que estariam envolvidos em tais atividades, onde a colaboração síncrona seria indispensável. Por outro lado, durante as etapas de implementação e verificação – desenvolvimento e integração de componentes de hardware, programação, *debugging*, etc. – o potencial de colaboração não deve ser muito elevado, pois desenvolvedores tendem a trabalhar individualmente e de forma assíncrona.

A segunda característica leva em conta as várias possibilidades de entrada e visualização de dados de projeto de sistemas integrados. Descrições gráficas, tais como esquemáticos de circuitos, diagramas de estado e diagramas UML, são usados concorrentemente com descrições textuais na forma de código de linguagens de descrição de hardware ou linguagens de programação, resultando em um cenário onde a colaboração pode ser dificultada pela heterogeneidade entre as formas de modelar e visualizar os dados de projeto. Para minimizar esse problema, o presente trabalho possibilita que projetistas utilizem diferentes formas de entrada e visualização dos dados de projeto, sempre mantendo a consistência entre cada visualização e a semântica do projeto.

A solução proposta no presente trabalho utilizou e ampliou o ambiente de projeto Cave [11], incluindo em sua estrutura um *framework* orientado a objetos que é responsável pela modelagem de dados de projeto e pela instanciação de ferramentas de apoio ao projeto. O suporte ao projeto colaborativo foi incluído nesse *framework*, de forma que futuras extensões ao modelo de dados de projeto ou ao conjunto de ferramentas poderão também utilizar tal recurso. A versão ampliada foi chamada de Cave2, e inclui também um conjunto de serviços que controla o funcionamento das sessões de projeto colaborativo chamado Service Space.

3 Arquitetura proposta

Nos últimos 20 anos, vários grupos se dedicaram à pesquisa na área de automação de projeto utilizando os chamados *frameworks* de CAD. Esses *frameworks* foram criados para definir uma camada entre o sistema operacional das estações de trabalho e os demais softwares que são necessários durante o projeto de um sistema integrado [12]. Dessa forma, toda a atividade de projeto seria realizada através de um ambiente integrado de software de CAD. A funcionalidade de tais *frameworks* inclui gerência de dados, suporte a desenvolvedores de ferramentas, suporte a integração e comunicação entre ferramentas.

Entretanto, as tendências no mercado de automação de projeto de sistemas eletrônicos seguiu o conceito de "best-tool-of-the-class": projetistas preferiram usar ferramentas individuais, específicas para determinadas atividades e desenvolvidas por diferentes fornecedores, ao invés de adotar a solução completa de um único fornecedor. Essa tendência se deve ao fato de que nenhum fornecedor é capaz de prover a melhor ferramenta para cada etapa do projeto, tal a complexidade de cada uma delas hoje

em dia. Essa situação, aliada ao fracasso das iniciativas de padronização de frameworks de CAD integrando soluções de múltiplos fornecedores, resultou na rejeição do conceito de frameworks de CAD sob o ponto de vista comercial.

Apesar disso, vários grupos de pesquisa chegaram a resultados significativos, e que podem ainda ser usados no suporte à colaboração entre projetistas [7,8,13]. A abordagem proposta leva em conta tais avanços, mas os utiliza dentro de um novo contexto ao fazer uso de técnicas de engenharia de software que não estavam disponíveis quando do desenvolvimento da primeira geração de *frameworks* de CAD.

3.1 Modelagem de dados e de ferramentas usando frameworks orientados a objetos

De acordo com Johnson [14], um *framework* orientado a objetos é um projeto reutilizável de software definido por um conjunto de classes abstratas e pela maneira pela qual as instancias dessas classes colaboram entre si. Levando em conta tal conceito, exploramos neste trabalho a possibilidade de que um framework de CAD possa incorporar um framework orientado a objetos, aqui definido como um conjunto de classes abstratas que modelam as primitivas de dados de projeto bem como as ferramentas de CAD que as manipulam. Esse framework orientado a objetos reutiliza diversos padrões de projeto, que são soluções já validadas para problemas tipicamente encontrados em arquiteturas complexas de software. A análise detalhada da arquitetura proposta neste trabalho mostra claramente a aplicação dos padrões Composite, Observer, Proxy, Chain of Responsibility e Façade [15], entre outros. Ao incluir um framework orientado a objetos, foi possível atingir um nível mais alto de abstração no que tange a funcionalidade de frameworks de CAD: gerencia e armazenamento de dados, padrões de comunicação entre ferramentas e suporte a extensibilidade do ambiente de automação de projeto.

O ambiente de projeto é implementado pelas sub-classes concretas derivadas do núcleo de classes abstratas do framework. Esse núcleo é dividido em dois pacotes principais. O primeiro é um conjunto extensível de primitivas de interface gráfica e de automação de projeto, usados para a instanciação de ferramentas de CAD que fazem a interface entre o projetista e o ambiente de projeto. Tais ferramentas são dinamicamente montadas de acordo com as necessidades e objetivos do projetista. O segundo pacote é um conjunto de primitivas de dados de projeto, que são instanciadas pelos projetistas à medida em que eles interagem com as ferramentas ao criar um projeto. A Figura 1 mostra como ambos conjuntos de classes interagem com os projetistas durante o processo de projeto. Primeiramente, as ferramentas são requisitadas e executadas através da instanciação e integração das primitivas de interface gráfica e automação de projeto. Uma vez instanciadas, essas ferramentas passam a construir o projeto instanciando as primitivas de dados de projeto sob o comando do projetista.

Para armazenar os dados de projeto ao final de cada sessão de projeto, bem como para permitir acesso multi-usuário aos dados - um requisito para colaboração síncrona - faz-se necessário um conjunto de serviços de persistência e consistência de dados. Figura 2 ilustra como tais serviços apóiam o ambiente de projeto ao permitir o acesso multi-usuário às primitivas de dados instanciadas pelos projetistas.

Ao incluir um framework orientado a objetos no núcleo do ambiente Cave, um grau de padronização significativo foi atingido uma vez que as dependências semânticas entre os módulos de ferramentas de CAD e as primitivas de dados de projeto estão definidas no nível abstrato do framework. Essa padronização, entretanto, não reduziu a flexibilidade do ambiente de projeto, pois sua inerente extensibilidade permite concretizações sucessivas das relações especificadas no nível abstrato. Em outras palavras, os módulos de ferramentas de CAD e as primitivas de dados de projeto são apenas pontos de partida, possibilitando extensões e especializações sem perder a compatibilidade com os serviços e recursos do ambiente de projeto, descritos na próxima subseção. Na Figura 3, pode-se ver uma parte do modelo de dados expresso no framework de primitivas de dados de projeto. Já a Figura 4 mostra seu potencial de expansão e atualização ao ilustrar um modelo de dados estendido para atender as necessidades de implementação de uma ferramenta para projeto baseado em interfaces (Interface-based design) [16].

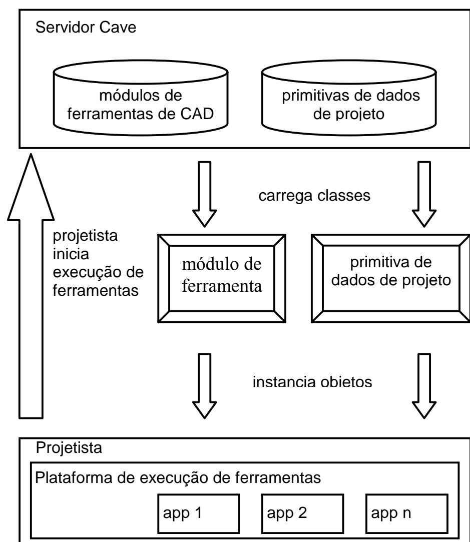


Figura 1. *Frameworks* de dados de projeto e de ferramentas no servidor Cave

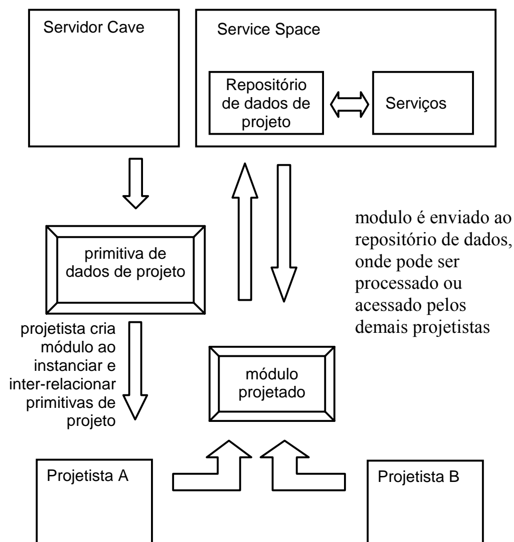


Figura 2. Compartilhamento de dados de projeto através do Service Space

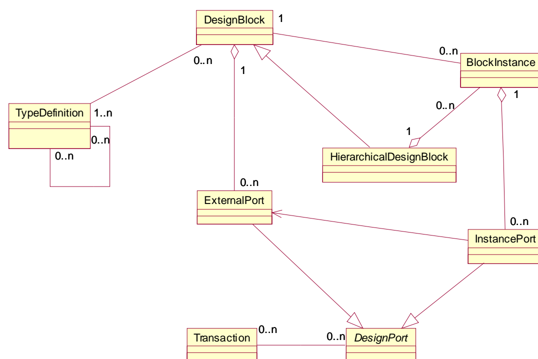


Figura 3. Diagrama de classes mostrando parcialmente o framework de primitivas de dados de projeto

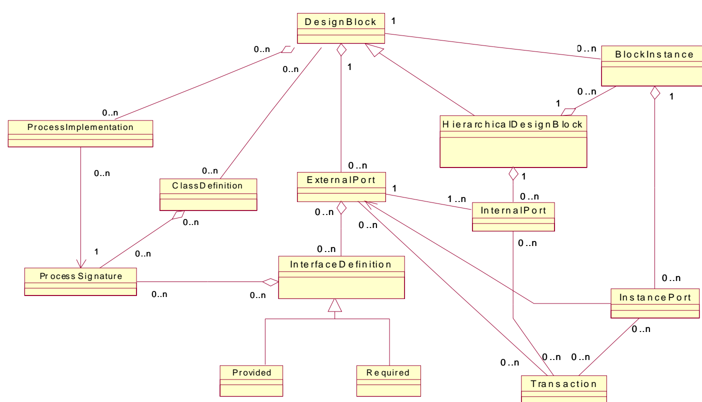


Figura 4. Extensão do framework de primitivas de dados de projeto visando suportar projeto baseado em interfaces

3.2 Integração de serviços

A abordagem aqui proposta inclui um elemento chamado Service Space, mostrado na Figura 2. A funcionalidade do Service Space inclui (1) a integração de ferramentas de CAD externas; (2) um ambiente de execução para os serviços internos, como repositórios de dados, controle de consistência e autenticação de usuários; e (3) a infraestrutura de localização e disponibilização de serviços, visando possibilitar a inclusão e exclusão dinâmica de serviços.

Existem várias possibilidades para a implementação desse elemento, tais como CORBA, *webservices* baseados em SOAP, Jini, além de outras abordagens não relacionadas com o paradigma de orientação a objetos. A escolha da tecnologia Jini para a implementação do presente protótipo foi baseada em algumas de suas peculiaridades, como o fato dela ser baseada em Java (assim como a implementação do framework orientado a objetos incluso no Cave2), de ter ferramentas de desenvolvimento disponíveis gratuitamente e de incorporar a maioria dos recursos necessários para localização e disponibilização de serviços. Jini também inclui um modelo de programação – desenvolvido sobre as fundações da linguagem Java – de forma a prover ao desenvolvedor as primitivas necessárias para a implementação de cessão de serviços (*leases*), transações e eventos. A chamada remota de métodos também depende de recursos da tecnologia Java, mais especificamente do pacote JavaRMI. Tal modelo de programação usa *proxies* para permitir referências locais a objetos remotos, permitindo que no domínio da aplicação todas as chamadas de métodos sejam feitas a objetos residentes na memória local, abstraindo as dificuldades inerentes à utilização de subsistemas remotos.

O Service Space está acessível através de um protocolo de descoberta (*discovery*), que permite aos clientes utilizar o servidor de *lookup*. Todos os serviços conectados ao Service Space usam a interface *Join* para notificar sua localização na rede e suas características de acesso. Já os clientes usam a interface de *Lookup* para procurar pelos serviços que pretendem utilizar. Os serviços conectados ao Service Space incluem ferramentas de CAD externas assim como serviços internos que são parte do ambiente implementado, como os mecanismos de autenticação de usuários, vários módulos de controle de acesso concorrente (bloqueios, transações, etc.), a interface de acesso ao repositório de dados e os serviços de prototipação. Tanto os serviços internos quanto as ferramentas externas podem ser incluídos dinamicamente, contribuindo assim com a escalabilidade dessa solução. Figura 5 traz uma visão geral dessa abordagem.

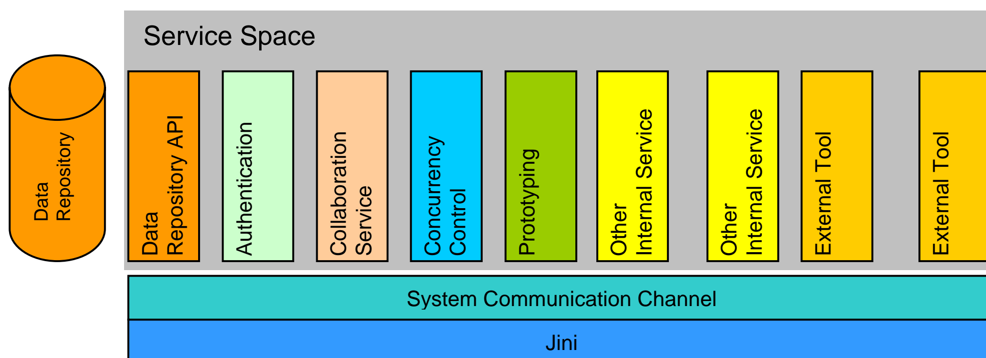


Figura 5. Arquitetura de integração de serviços no Service Space

A integração de serviços no Service Space utiliza estruturas de *proxy*, contribuindo com a transparência na distribuição de recursos no ambiente de rede. Todo cliente utilizando um dado serviço interage apenas com um *proxy* de serviço disponível localmente. O *proxy*, por sua vez, propaga dados e instruções de controle para a verdadeira implementação do serviço sempre que necessário. Essa abordagem é particularmente útil na simplificação da integração de ferramentas externas, bem como na modelagem e execução de *workflows*. Devido à clara separação entre a implementação da ferramenta e sua interface de acesso – definida pelo *proxy* - as ferramentas podem ser consideradas encapsuladas e a integração pode ser feita no nível dos *proxies*.

4 Visualização e Semântica de Projeto

Conforme descrito na seção anterior, tanto o modelo de dados de projeto quanto os módulos de ferramentas de CAD de Cave2 são implementados como frameworks orientados a objetos. Esses frameworks são inter-relacionados por natureza, pois os módulos de ferramentas de CAD devem poder instanciar, visualizar e modificar os dados de projeto de acordo com a intenção do projetista. Além disso, nossa abordagem explora o inter-relacionamento entre ambos frameworks para suportar o acesso colaborativo aos dados de projeto, fazendo com que a semântica do projeto – representada por instâncias de primitivas de dados de projeto – mantenha-se consistente com uma ou mais visualizações do projeto – representadas por módulos de interface gráfica de ferramentas de CAD.

Para permitir múltiplas representações visuais de uma única instância de dados de projeto, o framework deve incluir recursos para permitir que os blocos de dados mantenham suas representações visuais coerentes com o seu estado mesmo quando estejam sendo alterados. Por exemplo, se um projetista mantém duas representações visuais de um bloco de projeto – como no caso de duas janelas em um editor de esquemáticos, uma com a visão geral do projeto e ouotra com os detalhes de um dos blocos – as alterações no modelo de dados resultantes da interação do projetista com uma das representações visuais devem ser notificadas a outra delas, para que ela possa atualizar-se e manter-se consistente ao estado do modelo.

Várias das abordagens disponíveis na literatura para manter a consistência entre a informação e suas representações visuais definem a separação explícita entre o modelo da informação e suas visualizações, e prevêem a existência de um mecanismo de consistência que controla as interações entre o modelo, suas visualizações e o mundo externo. Essa arquitetura é a base do *framework* MVC (Model-View-Controller), incluso na linguagem de programação Smalltalk [17] e posteriormente formalizado como o padrão de projeto *Observer* por Gamma et. al. [15].

Desacoplando o modelo e suas visualizações, facilita-se a implementação de várias visualizações diferentes - porém equivalentes – do mesmo bloco de projeto. Para algumas formas particulares de representação onde a visualização incorpora elementos adicionais aos definidos pelo modelo, pode-se ainda flexibilizar o mecanismo de consistência de forma a propagar apenas as alterações que resultem em modificação no estado do modelo. Neste trabalho, exploramos essa possibilidade na implementação de duas metodologias para colaboração: visualmente acoplada e visualmente desacoplada. Tais metodologias, detalhadas em [21], permitem que múltiplos projetistas compartilhem um modelo de dados de projeto sem que necessariamente compartilhem uma única visualização. Usando a metodologia visualmente desacoplada, pode-se definir quais alterações feitas por um projetista devem ser propagadas aos outros projetistas, permitindo maior flexibilidade no controle de consistência entre visualizações. O único requisito é que as alterações que resultam em mudanças na semântica do modelo de dados de projeto sejam obrigatoriamente propagadas.

No *Framework* Cave2, as primitivas de representação de projeto – portas lógicas e blocos funcionais, por exemplo – são modeladas como instâncias de uma classe concreta, que por sua vez herda parte de sua interface e comportamento de uma classe abstrata. Essa abordagem é comum em *frameworks* orientados a objetos, pois as classes abstratas - mesmo não modelando diretamente nenhum elemento do domínio da aplicação – tem grande importância na organização da hierarquia de classes e na atribuição de comportamento comum a uma classe de objetos. Usamos essa abordagem para incluir em Cave2 o suporte às metodologias colaborativas: as bases do mecanismo de controle de consistência entre o modelo de dados de projeto e suas múltiplas visualizações estão incluídos nas superclasses abstratas do *framework* orientado a objetos – as primitivas de dados de projeto e de elementos de interface gráfica. Assim, todos os modelos de dados projeto usados no Cave2 – incluindo aqueles que serão integrados em atualizações futuras – vão herdar tal comportamento.

Ao aplicar tais conceitos, foi possível separar completamente a semântica de projeto de suas formas de visualização, pois são modeladas por objetos diferentes. Assim, podemos permitir múltiplas visualizações – por diferentes projetistas – de um único bloco de dados de projeto. É importante ainda ressaltar a possibilidade de múltiplas formas de visualização – por exemplo um dado bloco de projeto pode ser visto como um esquemático gráfico ou como uma descrição textual na forma de HDL (Hardware Description Language).

Para garantir a consistência entre a semântica do projeto e suas visualizações, assim como entre a semântica de blocos de projeto inter-relacionados, usamos mecanismos de notificação de atualizações (*update/notify*). Estes mecanismos capturam a interação entre o usuário e uma das visualizações, atualizam a respectiva semântica de projeto e então notificam as demais visualizações para que atualizem-se, a fim de refletir possíveis mudanças.

Um *framework* que implementa a infraestrutura necessária para prover um serviço flexível de notificação foi apresentado por Shen e Sun [18]. Esse *framework* considera um cenário onde vários usuários atualizam simultaneamente um conjunto comum de dados, e as atualizações realizadas por cada usuário devem ser notificadas aos demais. A notificação é dividida em duas partes, a de saída e a de chegada. Para cada usuário, a notificação de chegada representa uma alteração realizada por outro usuário, enquanto a notificação de saída representa a propagação aos outros usuários da alteração realizada localmente. Cada um desses tipos de notificação é caracterizado por sua freqüência e sua granularidade. A freqüência pode assumir uma de três possibilidades – instantânea, escalonada ou definida pelo usuário – enquanto a granularidade define se a notificação deve ser feita para cada alteração de estado, ou apenas para um sub-conjunto pré-definido de alterações.

Na implementação do sistema aqui descrito, aplicamos um sistema de notificação semelhante ao de Shen e Sun. Inicialmente desenvolvido de forma independente, nossa abordagem foi beneficiada pela visão sistemática do problema apresentada por [18]. O mecanismo resultante foi integrado ao repositório de dados de projeto apresentado a seguir.

5 Repositório de dados de projeto

Um repositório de dados no Cave2 oferece um alto nível de abstração se comparado a bases de dados de projetos inspiradas em consultas ou API (*query-based ou API-based).* A fim de permitir que os desenvolvedores de CAD se concentrassem somente nos domínios de aplicação da ferramenta de projeto, a abordagem baseada em API foi ampliada com a introdução de uma arquitetura baseada no conceito de persistência transparente [19]. De acordo com esta técnica, o mecanismo de persistência de objetos de dados deve ser escondida sempre que possível. Assim, o cliente pode ter a impressão de estar tratando com objetos regulares em memória, e não com registros em uma base de dados. A arquitetura proposta, incluída no núcleo do *framework* Cave2, permite a gerência direta dos dados dos objetos através de sua própria API - por exemplo, chamando o método *tempblock.addPort(new CaveVisualPort())* ao invés de usar uma API de banco de dados ou uma linguagem de consulta para fazer isso, como o trecho de código abaixo.

insert into PORT (portid, name, type) values (64, ‘CTRL2’, ‘in’)
insert into PORTBLOCK (portkey, blockkey) values ('64', '12')

O acesso ao repositório é baseado em *proxies*, assim como cada serviço do *Service Space* do Cave2. Para cada ferramenta de projeto que estiver usando o repositório, um *proxy* de serviço é carregado do *Service Space*. Entretanto, tal *proxy* de serviço não é usado diretamente como uma interface completa ao repositório. Seu papel restringe-se à criação, remoção e localização de objetos de projeto. Todas operações restantes são manipuladas pelos *proxies* de objeto*,* que representam individualmente cada objeto do repositório. A consistência entre o *proxy* de objeto e o respectivo objeto de projeto armazenado no repositório é feita de forma transparente, e está descrita a seguir.

O diagrama de seqüência UML, ilustrado na Figura 6, descreve o procedimento usado no repositório de dados do Cave2. O exemplo descrito supõe que o serviço de repositório já foi encontrado, contatado, e que um *proxy* de serviço já foi carregado. Quando um objeto visual é criado pela ferramenta de projeto, deve ser também criado dentro do repositório. Este papel é desempenhado pelo *proxy* de serviço, que cria a respectiva entrada do objeto de projeto no repositório, criando também um *proxy* local para o objeto de projeto remoto. Cada operação adicional executada através desse objeto visual será delegada ao *proxy* do objeto de projeto, que notificará sua contraparte remota e o seu próprio objeto visual sobre a operação recentemente executada (denotado na figura como uma chamada ao método *doSomething*).

Do ponto de vista do desenvolvedor da ferramenta, grande parte da funcionalidade desse repositório é transparente. A ferramenta deve obter somente um *proxy* por objeto visual, e lidar com os *proxies* como se estivesse tratando de objetos visuais. Isso simplifica significativamente o desenvolvimento, porque uma ferramenta pode ser desenvolvida primeiramente *standalone*, tratando somente de seus objetos visuais locais, e, adicionalmente, pode ter sua funcionalidade ampliada para utilizar o repositório, mudando somente as chamadas de método dos objetos visuais para os *proxies* do objeto de projeto.

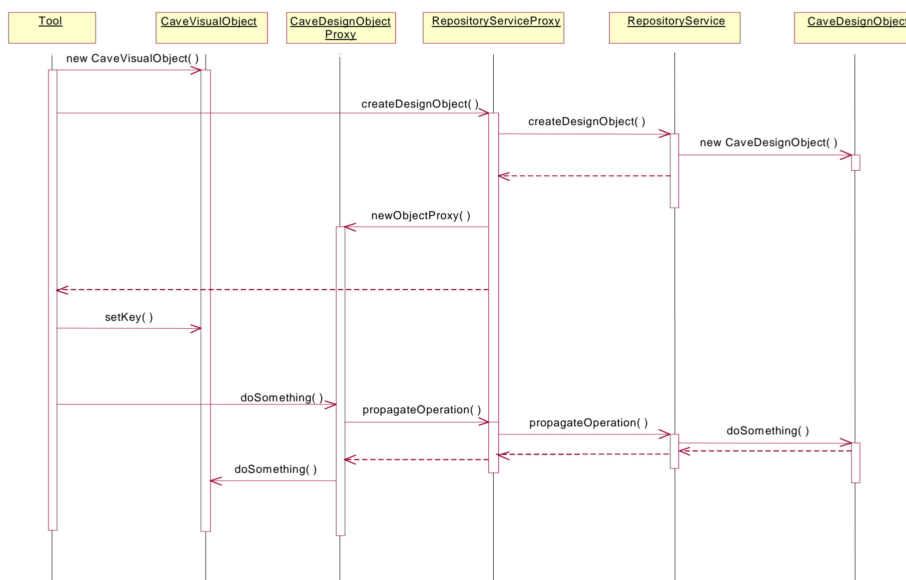


Figura 6. Acesso ao repositório de dados de projeto

A arquitetura Cave2 reduz o problema de manter a consistência dos dados dentro das sessões colaborativas de projeto no que tange a consistência entre a visualização e os objetos do projeto. Em outras palavras, as visões de projeto de cada um dos projetistas devem ser sincronizadas com os dados de projeto no repositório. Como descrito nas seções anteriores, tal consistência é reforçada com a interação entre os objetos visuais e objetos de projeto através dos *proxies*. Nossa abordagem para a sincronização entre os objetos visuais e os objetos de projeto é baseada nas técnicas de atualização/notificação *(update/notify)*, enviadas através dos objetos *proxies* (Figura 7). Tais atualizações são iniciadas pela interface gráfica da ferramenta de projeto e propagadas para os objetos *proxies* como chamadas de método. Quando seus métodos forem chamados, o objeto *proxy* instancia um objeto evento, o qual encapsula sua chave de identificação, o nome do método chamado e os parâmetros passados. Os parâmetros reais são incluídos somente no objeto de evento quando eles são instâncias de tipos primitivos, tal como *strings* de caracteres ou números. Se um objeto visual for passado como parâmetro, o objeto *proxy* passa sua chave de identificação, pois a serialização do objeto real seria demasiadamente custosa.

O objeto de evento é passado ao *proxy* de serviço, que usa uma conexão RMI para chamar o método que irá disparar o serviço remoto. O objeto de evento é passado como um parâmetro, de tal forma que o repositório pode encontrar o objeto de projeto real dentro de seu índice e chamar o método referenciado através dos parâmetros fornecidos.

Nossa implementação dos objetos de evento foi construída através da tecnologia Jini [20]. Ela segue o modelo básico de eventos introduzido na versão 1.1 da linguagem Java. Tal modelo define que um evento consumidor deve se registrar com cada evento produtor com o qual pretenda manter contato. Os eventos produtores devem executar tal procedimento de registro e notificar todos os consumidores caso um de seus estados mude. Um produtor pode produzir os eventos associados a diferentes mudanças de estado. Com isso, os diferentes tipos de eventos podem ser diferenciados pelo atributo *eventID* e pela classe real do objeto do evento. Uma complexidade adicional deve ser manipulada quando os eventos precisam ser enviados através da rede. O modelo de eventos remotos Jini usa Java RMI, que implementa recursos para re-seqüenciamento de eventos e tolerância a falhas na rede. O primeiro é necessário para determinar a ordem correta que os eventos recebidos serão processados pelo consumidor, enquanto o último - baseado no conceito de exceções - permite a recuperação de eventos perdidos.

Como descrito na Figura 7, a comunicação entre os *proxies* de objeto e suas respectivas contrapartes localizadas no repositório de projeto são baseadas na propagação dos eventos. Entretanto, a implementação de tal comunicação oferece várias alternativas. Os proxies podem ser genéricos o bastante para que possam enviar ao repositório todos os eventos recebidos da GUI da ferramenta de projeto, ou ainda encapsular alguma inteligência de forma a verificar a semântica dos eventos, encaminhando somente os válidos ao repositório. A primeira estratégia pode ser utilizada para qualquer tipo de interface com usuário e objeto visual, enquanto a segunda dever ser particularmente

implementada a um tipo específico de ferramenta e objeto visual. Em outras palavras, a criação de *proxies* que possam compreender a semântica de eventos envolve a criação de uma instância de um tipo, requerendo a criação prévia - para cada objeto de projeto - de uma classe que defina a interface do *proxy*.

A implementação proposta inclui os *proxies* genéricos, que podem capturar eventos de todas as primitivas de visualização do *framework* Cave2, bem como as suas possíveis extensões. Uma vez capturados, esses eventos podem ser enviados ao repositório remoto de acordo com a disponibilidade do canal de comunicação. Como mencionado anteriormente, tais *proxies* genéricos não executam nenhuma análise semântica dos eventos capturados. Em muitos casos, entretanto, uma análise mais detalhada da semântica é necessária a fim de otimizar a comunicação: os eventos semanticamente inválidos não são propagados ao servidor remoto, e os eventos semanticamente corretos podem ser executados na visualização ao mesmo tempo que estão sendo executados aos objetos remotos do projeto.

A fim executar tal análise semântica, são necessários *proxies* de objetos específicos para cada objeto visual. Cada *proxy* de objeto de projeto deve executar os mesmos métodos executados por seu respectivo objeto de projeto, permitindo que o comportamento individual seja executado dentro de cada chamada de método. Na implementação atual, tais classes do *proxy* são codificadas manualmente em um processo tedioso, mas uma automatização adicional pode ser fornecida em um procedimento similar ao processo descrito em [19]. Nesses casos, todos os *proxies* herdam os mecanismos de comunicação implementados pelas classes de *proxy* genérico mencionadas acima, reutilizando assim todos os procedimentos genéricos implementados nas superclasses para a comunicação com o repositório remoto. Somente os métodos específicos à análise semântica devem ser implementados.

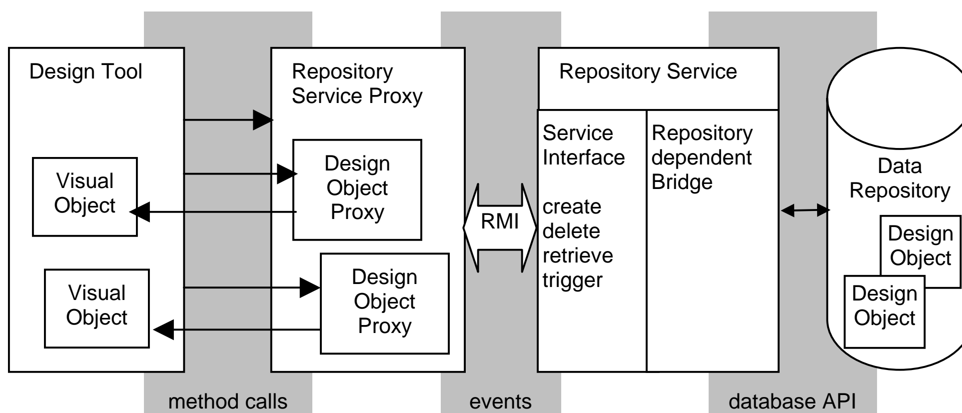


Figura 7. Sincronização entre visualização e semântica de projeto usando *proxies*

A comunicação entre os *proxies* e o repositório pode também ter várias possibilidades de implementação, tais como, conexões dedicadas usando *sockets*, invocação remota de métodos ou eventos distribuídos. Todas as três implementações são suportadas pelo Cave2 e definidas no *Service Space*. O uso de *sockets*, entretanto, requereria a descrição de um protocolo de comunicação completo. O uso de RMI simplifica o desenvolvimento, pois fornece uma interface de alto nível que serve de fundação ao protocolo de comunicação. Entretanto, o uso de uma comunicação RMI entre cada *proxy* de objeto de projeto e suas contrapartes pode ser muito custoso, requerendo uma conexão RMI dedicada por *proxy*, sendo que centenas de *proxies* poderiam estar sendo usados simultaneamente por uma dada ferramenta. Para otimizar tal procedimento, combinamos a abordagem RMI com uma abordagem de eventos distribuídos. A implementação proposta foi construída com uma única conexão RMI entre o *proxy* de serviço e o repositório. Através desta única conexão RMI, toda a comunicação entre a ferramenta e o repositório foi implementada como uma série de eventos.

6 Considerações Finais

O CAD Framework implementado neste trabalho foi chamado Cave2 e seguiu a clássica arquitetura em níveis apresentada por Barnes, Harrison, Newton e Spickelmier [12]. Durante o projeto

e a implementação do Cave2, uma série de avanços em relação às abordagens anteriores foi obtida com a exploração das vantagens advindas do uso de um framework orientado a objetos:

- uma vez que frameworks orientados a objetos são extensíveis por definição, o mesmo pode ser dito a respeito da implementação no Cave2 dos conjuntos de primitivas de dados de projeto, bem como de blocos para a construção de ferramentas de CAD. Isso implica que tanto o modelo de representação de projeto quanto os módulos de software processando tal modelo podem ser atualizados ou adaptados para uma metodologia de projeto específica, e que essas atualizações e adaptações ainda herdarão os aspectos arquiteturais e funcionais implementados nos elementos básicos do framework orientado a objetos;

- ambos os aspectos relativos à semântica do projeto e à visualização do projeto são partes do framework orientado a objetos, mas em modelos claramente separados. Isso possibilita o uso de várias estratégias para a visualização de um conjunto de dados de projeto, o que dá aos participantes de uma sessão de projeto colaborativo a flexibilidade de escolha individual de estratégia de visualização;

- o controle de consistência entre semântica e visualização - uma questão particularmente importante em um ambiente de projeto onde coexistem múltiplas visualizações de cada projeto - também está incluído nas fundações do framework orientado a objetos implementado. Esse mecanismo é genérico o bastante para ser usado também pelas possíveis extensões do modelo de dados de projeto, uma vez que ele é baseado na inversão de controle entre a visualização e a semântica. A visualização recebe a intenção do usuário e propaga esse evento ao modelo da semântica, o qual avalia a possibilidade de uma mudança de estado. Se positivo, ele dispara a mudança de estado em ambos os modelos de visualização e semântica. A abordagem proposta nesta tese usa tal inversão de controle para incluir um nível adicional de processamento entre a semântica e a visualização, visando o controle de consistência nos casos de múltiplas visualizações;

- para otimizar o mecanismo de controle de consistência entre semântica e visualização, uma abordagem baseada em eventos foi proposta, buscando discretizar cada interação entre o projetista e suas visualizações do projeto. A informação sobre cada uma das interações é encapsulada em um objeto-evento, que pode ser propagado para o modelo da semântica do projeto - e então para as demais possíveis visualizações - de acordo com a política de consistência que esteja sendo usada. Além disso, o uso de eventos permite que as interações do usuário com a visualização sejam acumuladas para uma posterior sincronização com a semântica do projeto, caso haja indisponibilidade de conexão entre elas;

- o uso de objetos de *proxy* aumentou significativamente o nível de abstração da integração de recursos de automação de projeto, pois tanto ferramentas e serviços remotos quanto os instalados localmente são acessados através de chamadas de métodos em um objeto local. A conexão aos serviços e ferramentas remotos é obtida através de um protocolo de look-up, abstraíndo completamente a localização de tais recursos na rede e permitindo a adição e remoção em tempo de execução;

- o CAD Framework foi implementato completamente usando a tecnologia Java, usando dessa forma a Java Virtual Machine como intermediário entre o sistema operacional e o CAD Framework, garantindo dessa forma a independência de plataforma.

Todas as contribuições listadas anteriormente contribuiram com o aumento do nível de abstração da distribuição de recursos de automação de projeto e também apresentaram um novo paradigma para a interação remota entre projetistas. O CAD Framework no qual tais contribuições foram aplicadas é capaz de suportar colaboração de granularidade fina baseada em eventos, onde cada atualização feita por um projetista pode ser propagada para o restante da equipe, mesmo que estejam todos geograficamente distribuídos. Isto pode aumentar a sinergia de grupo entre os projetistas e permitir uma troca mais rica de experiências entre eles, aumentando significativamente o potencial de colaboração quando comparado com abordages baseadas em acesso a arquivos e registros propostas anteriormente.

Três estudos de caso diferentes foram realizados para validar a abordagem proposta, cada um deles envolvendo um sub-conjunto das contribuições do presente trabalho. O primeiro utiliza a arquitetura de distribuição de recursos baseada em proxies para implementar uma plataforma de prototipação usando módulos de hardware reconfigurável [22]. O segundo estende as fundações do framework orientado a objetos visando suportar projeto baseado em interfaces [23]. Essas extensões - primitivas de representação de projeto e partes de ferramentas - são usadas na implementação de uma ferramenta chamada IBlaDe, que permite a criação colaborativa de modelos funcionais e estruturais de sistemas integrados. O terceiro estudo de caso aborda a possibilidade de integração de metadados

multimídia ao modelo de dados de projeto [24]. Essa possibilidade é explorada no contexto de uma plataforma online de educação e treinamento.

Referências

[1] SEMICONDUCTOR INDUSTRY ASSOCIATION. International Technology Roadmap for Semiconductors: 1999 edition. Austin: International SEMATECH, 1999.

[2] BOREL, J. et al.The MEDEA+ Design Automation Roadmap. Paris: MEDEA+ Office, 2002.

[3] BROWN, S.. Law of accelerating returns. Midyear Forecast, EE Times Special Report, 2000.

[4] INDRUSIAK, L. S. A Review on the Framework Technology Supporting Collaborative Design of Integrated Systems. Exame de Qualificação. Porto Alegre: PPGC UFRGS, 2002. 108 p.

[5] JOHANSEN, R. “Groupware: Computer support for business teams”. New York: The Free Press, 1988.

[6] KATZ, R. H. Towards a unified framework for version modeling in engineering databases. In: ACM Computing Surveys. Vol. 22, No. 4, December 1990. p. 375-408.

[7] HARRISON, D. S. et al. Data management and graphics editing in the Berkeley Design Environment. In: Proceedings of the IEEE International Conference in Caputer Aided Design, 1986.

[8] WAGNER, F. R.; LIMA, A.H.V. Design Version Management in the GARDEN Framework. In : Proceedings of the 28th Design Automation Conference, ACM/IEEE, June 1991. p. 704-710.

[9] BRGLEZ, F.; LAVANA, H. A Universal Client for Distributed Networked Design and Computing. In: Proceedings of the 38th Design Automation Conference, 2001. Los Alamitos: IEEE Computer Society, 2001.

[10] SCHNEIDER, A. et al. Internet-Based Collaborative Test Generation with MOSCITO. In: Proceedings of Design, Automation and Test in Europe, Paris, 2002. p. 221- 226.

[11] INDRUSIAK, L. S.; REIS, R. A. L. From a Hyperdocument-Centric to an Object-Oriented Approach for the Cave Project In: XIII Symposium on Integrated Circuits and System Design, 2000, Manaus. Proceedings. Los Alamitos: IEEE Computer Society, 2000. p.125 – 130.

[12] BARNES, T.J.; HARRISON, D.; NEWTON, A.R.; SPICKELMIER, R.L. Electronic CAD Frameworks. Boston: Kluwer Academic Publishers, 1992. 196 p.

[13] VAN DER WOLF, P.; BINGLEY, P.; DEWILDE, P. On the Architecture of a CAD Framework: The NELSIS Approach. In: Proceedings of the European Design Automation Conference, 1990. p. 29-33.

[14] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. Journal of Object-Oriented Programming, Vol 1 (2), 1988, pp. 22-35.

[15] GAMMA, E. et al. "Design Patterns: elements of reusable object-oriented software". Reading: Addison Wesley, 1995.

[16] INDRUSIAK, L. S., REIS, R. A. L., GLESNER, M. Supporting Consistency Control between Functional and Structural Views in Interface-based Design Models In: Proceedings of the Forum on Design Languages, 2003, Frankfurt.

[17] KRASNER, G. E.; POPE, S.T. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. Journal of Object-Oriented Programming, 1(3):26–49, Aug./Sep. 1988.

[18] SHEN, H.; SUN, C. Flexible notification for collaborative systems. In: Proc. of ACM Conference on Computer Supported Cooperative Work, 2002. p. 77-86.

[19] ROOS, R. M. “Java Data Objects”. London: Addison-Wesley, 2003. 264 p.

[20] LI, S. “Professional Jini”. Birmingham: Wrox Press, 2000. 886 p.

[21] INDRUSIAK, L. S.; BECKER, J.; GLESNER, M.; REIS, R. A. L. Distributed Collaborative Design over Cave2 Framework. In: 11th International Conference on Very Large Scale Integration of Systems-on-Chip, 2002, Montpellier. SOC Design Methodologies. Boston : Kluwer Academic Publishers, 2001. p. 97-108.

[22] INDRUSIAK, L. S.; LUBITZ, F.; GLESNER, M.; REIS, R. A. L. Ubiquitous Access to Reconfigurable Hardware: Application Scenarios And Implementation Issues. In: Design Automation and Test in Europe (DATE), 2003, Munich. Proceedings. Los Alamitos: IEEE Computer Society, 2003. p.940 – 945.

[23] INDRUSIAK, L. S.; REIS, R. A. L.; GLESNER, M. Supporting Consistency Control between Functional and Structural Views in Interface-based Design Models. In: Forum on Design Languages, 2003, Frankfurt. Proceedings. Gières: ECSI, 2003. CDROM.

[24] INDRUSIAK, L. S.; GLESNER, M.; REIS, R. A. L.; ALCÁNTARA, G. P.; HOERMANN, S.; STEINMETZ, R.Reducing Authoring Costs of Online Training in Microelectronics Design by Reusing Design Documentation Content. In: 2003 International Conference on Microelectronic Systems Education, 2003, Anaheim. Proceedings. Los Alamitos: IEEE Computer Society, 2003. p.57 – 58.

References

[ARN2000] ARNOUT, G. **C for System Level Design**. Available at: <http://www.systemc.org/papers/coWare.pdf>. Visited on 2000.

[ARO00] ARNOLD, K. et al. **The Jini specification**. Reading: Addison-Wesley, 1999.

[BAR92] BARNES, T.J. et al. **Electronic CAD *Framework*s**. Dordrecht: Kluwer Academic Publishers, 1992. 196 p.

[BEC97] BECKER, J. **A Partitioning Compiler for Computers with Xputer-based Accelerators.** 1997. Doctoral Thesis - Fachbereich Informatik der Universität Kaiserslautern, Kaiserslautern.

[BEC99] BECKER, J. et al. DInCAD: Distributed Internet-based CAD Methods for Future Complex Microelectronic Systems. In: WORKSHOP OF THE GERMAN-BRAZILIAN BILATERAL PROGRAMME FOR SCIENTIFIC AND TECHNOLOGICAL COOPERATION THROUGH BMBF AND CNPq, 1999, Koenigswinter. **Proceedings**… Koenigswinter, 1999. [S.l.: s.n.], 1999.

[BEK99] BECK, K. **Extreme Programming Explained**. Reading: Addison Wesley, 1999.

[BEN96] BENINI, L.; BOGLIOLO, A.; DE MICHELI, G. Distributed EDA tool integration: the PPP paradigm. In: INTERNATIONAL CONFERENCE ON COMPUTER DESIGN. **Proceedings**… [S.l.: s.n.], 1996. p. 448-453.

[BOC99] BOSCH, J.; MOLIN, P.; MATTSSON, M.; BENGTSSON, P.O. Object-Oriented Frameworks - Problems & Experiences. In: FAYAD, M.; SCHMIDT, D.; JOHNSON, R. (Ed.). **Object-Oriented Application Frameworks**. New Jersey: John Wiley, 1999. p. 55-82.

[BOO91] BOOCH, G. **Object-Oriented Design With Applications**. [S.l.]: Benjamin Cummings, 1991.

[BOS95] BOSCH, O.; WOLF, P.; HOEVEN, A. Design Flow Management: more than convenient tool invocation. In: RAMMIG, F.J.; WAGNER, F. R. (Ed.). **Electronic Design Automation Frameworks.** London: Chapman & Hall, 1995. p. 149-158.

[BRE95] BREDENFELD, A. Cooperative Concurrency Control for Design Environments. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1995, Brighton. **Proceedings**... Los Alamitos: IEEE Computer Society, 1995.

[BRG2001] BRGLEZ, F.; LAVANA, H. A Universal Client for Distributed Networked Design and Computing. In: DESIGN AUTOMATION CONFERENCE, 2001. **Proceedings**… Los Alamitos: IEEE Computer Society, 2001.

[BRI2001] BRISOLARA, L. B.; INDRUSIAK, L. S.; REIS, R. A. L. An Hierarchical Schematic Editor to WWW. In: MICROELECTRONIC STUDENTS FORUM, 2001, Pirenopolis. **Proceedings**… [S.l.: s.n.], 2001.

[BRI2002] BRISOLARA, L. B.; INDRUSIAK, L. S.; REIS, R. A. L. Modelagem Orientada a Objetos de Primitivas de Projeto de Sistemas Eletrônicos voltada para Colaboração In: WORKSHOP IBERCHIP, 2002, Guadalajara. **Proceedings**… [S.l.: s.n.], 2002.

[BRN98] BRONDANI, C.; BRASCO,F.F.; INDRUSIAK, L.S.; REIS, R.A.L. Graphic Schemes Development to Jale. In: MICROELECTRONICS SEMINAR, 13., 1998, Bento Gonçalves. **Proceedings**... Porto Alegre: CPGCC da UFRGS, 1998.

[BRO92] BROCKMAN, J.B.; COBOURN, T.F.; JACOME, M.F.; DIRECTOR, S.W. The Odyssey CAD Framework. **IEEE DATC Newsletter on Design Automation**, Spring 1992.

[BRO92a] BROCKMAN, J.B.; DIRECTOR, S.W. A Schema-Based Approach to CAD Task Management. IFIP WG 10.2 WORKSHOP ON ELECTRONIC DESIGN AUTOMATION FRAMEWORKS, 1992. **Proceedings**...Amsterdam: North-Holland, 1992.

[BRU2002] BRUSCHI, F.; DI NITTO, E.; SCIUTO, D. SystemC Code Generation from UML Models. In: FORUM ON SPECIFICATION AND DESIGN LANGUAGES, FDL, 2002, Marseille. **Proceedings**… [S.l.]: ECSI, 2002.

[BRW2000] BROWN, S. Law of accelerating returns. **EE Times Special Report**, Midyear Forecast, 2000.

[BUL2001] BULL, J. M. et al. **Performance evaluation of Java against C and Fortran for Scientific Applications**. Available at: <http://aspen.ucs.indiana.edu/CandCPandE/jg2001/C564bull/jgflangcomp_ccpe.pdf>. Visited on Aug. 9, 2002.

[BUS89] BUSHNELL, M.; DIRECTOR, S.W. Automated Design Tool Execution in the Ulysses Design Environment. **IEEE Transactions on Computer-Aided Design**, Piscataway, v.8, n.3, p. 279-287, Mar. 1989.

[CAD2001] CADENCE DESIGN SYSTEMS, INC. **OpenAccess C-Level API Reference**, Austin, Version 1.1.02, 2001.

[CAD2001a] CADENCE DESIGN SYSTEMS, INC. **Datasheet**: Cadence Virtual Component Co-Design. Available at: <http://www.cadence.com/datasheets/vcc_environment.html>. Visited on Nov. 15, 2002.

[CAR99] CARBALLO, J.A; DIRECTOR, S.W. Constraint Management for Collaborative Electronic Design. In: DESIGN AUTOMATION CONFERENCE, 1999. **Proceedings**… Los Alamitos: IEEE Computer Society, 1999. p.395-400.

[CFI94] CAD FRAMEWORK INITIATIVE INC. **Design Representation Programming Interface**. Austin, Document No. dit-92-S-1, 1994.

[CHA98] CHAN, F.; SPILLER, M.; NEWTON, R. WELD - An Environment for Web-Based Electronic Design. In: DESIGN AUTOMATION CONFERENCE, 1998. **Proceedings**… Los Alamitos: IEEE Computer Society, 1998. p. 146-152.

[CHU90] CHUNG, M.J.; KIM, S. An Object-Oriented VHDL Design Environment. In: DESIGN AUTOMATION CONFERENCE, 1990. **Proceedings**… Los Alamitos: IEEE Computer Society, 1990. p. 431-436.

[CLA2001] C-LAB. **Astai(R)**. Available at : <http://www.c-lab.de/astair/>. Visited on Feb. 14, 2002.

[COD70] CODD, E.F. A Relational Model of Data for Large Shared Data Banks. **Communications of the ACM**, New York, v. 13, n.6, p. 377-387, 1970.

[DAL2000] DALPASSO, M. et al. **JavaCAD Project**. Available at: <http://www.javacad.eu.org>. Visited on Dec. 5, 2001.

[DAN89] DANIELL, J.; DIRECTOR, S.W. An Object Oriented Approach to CAD Tool Control Within a Design Framework. In: DESIGN AUTOMATION CONFERENCE, 1989. **Proceedings**… Los Alamitos: IEEE Computer Society, 1989. p. 197-202.

[DAV2001] DAVIS, D. et al. **Forge-J:** High Performance Hardware from Java. Available at: <http://www.xilinx.com/forge/forge.htm>. Visited on Jan. 3, 2002.

[DEM94] DE MICHELI, G. **Synthesis and Optimization of Digital Circuits**. New York: McGraw-Hill, 1999. 576 p.

[DES2000] DESICS DIVISION. **Ocapi-xl**. Available at: <http://www.imec.br/ocapi>. Visited on Nov. 26, 2001.

[ELE2000] ELECTRONIC INDUSTRIES ALLIANCE. **Electronic Design Interchange Format**. Available at: <http://www.edif.org>. Visited on July 26, 2001.

[ELI91] ELLIS, C.A.; GIBBS, S.J.; REIN, G. L. Groupware: Some issues and experiences. **Communications of the ACM**, New York, v.34, n.1, p.38-58, Jan. 1991.

[ELL97] ELLSBERGER, J.; HOGREFE, D.; SARMA, A. **SDL - Formal Object-Oriented Language for Communication Systems**. [S.l.]: Prentice Hall, 1997.

[ESW76] ESWARAN, K.P.; GRAY, J.; LORIE, R.; TRAIGER, I.L. The notions of consistency and predicate locks in a database system. **Communications of the ACM**, New York, v.19, n.11, p. 624–633, Nov. 1976.

[FID90] FIDUK, K.W. et al. Design Methodology Management - A CAD Framework Initiative Perspective. In: DESIGN AUTOMATION CONFERENCE, 1990. **Proceedings**… Los Alamitos: IEEE Computer Society, 1990. p. 278-283.

[FRA2000] FRAGOSO,J.L.; MORAES, F.; REIS, R. WTROPIC: A Macro-Cell Generator on Internet. In: SIM, 15. ,2000, Torres. **Proceedings**... Porto Alegre: Instituto de Informática da UFRGS, 2000.

[FRE99] FREEMAN, E.; HUPFER, S.; ARNOLD, K. **JavaSpaces:** principles, patterns, and pratice. Reading: Addison Wesley, 1999.

[FRI2002] FREE-IP PROJECT. **Free-DES**. Available at: <http://www.free-ip.com/DES>. Visited on Sept. 7, 2002.

[GAJ00] GAJSKI, D. et al. **The SpecC Methodology**. Available at: <http://www.ics.uci.edu/~specc>. Visited on May 28, 2001.

[GAM95] GAMMA, E. et al. **Design Patterns:** elements of reusable object-oriented software. Reading: Addison Wesley, 1995.

[GED88] GEDYE, D.; KATZ, R. Browsing the Chip Design Database. In: DESIGN AUTOMATION CONFERENCE, 1988. **Proceedings**… Los Alamitos: IEEE Computer Society, 1988. p. 269-274.

[GEL85] GELERNTER, D. Generative Communication in Linda. **ACM Transactions on Programming Languages and Systems**, New York, v. 7, n. 1, p. 80-112, 1985.

[GIG2002] GIGASCALE SILICON RESEARCH CENTER. **Diva**. Available at: <http://www.gigascale.org/diva/>. Visited on Oct. 12, 2002.

[GIR87] GIRCZYC, E.F.; LY, T. STEM: an IC design environment based on the Smalltalk model-view-controller construct. In: DESIGN AUTOMATION CONFERENCE, 1987. **Proceedings**… Los Alamitos: IEEE Computer Society, 1987. p. 757-763.

[GOL2002] GOLDFEDDER, B. **The Joy of Patterns.** Boston: Addison Wesley, 2002.

[GOS96] GOSLING, J.; JOY, B.; STEELE,G. **The Java Language Specification.** Available at: <http://java.sun.com/doc/language_specification.html>. Visited on Dec. 5, 1996.

[GUP89] GUPTA, R. et al. An Object-Oriented VLSI CAD Framework. **IEEE Computer**, Los Alamitos, p. 28-37, May 1989.

[HAR86] HARRISON, D.S. et al. Data management and graphics editing in the Berkeley Design Environment. In: IEEE INTERNATIONAL CONFERENCE IN COMPUTER AIDED DESIGN, 1986. **Proceedings**… [S.l.: s.n.], 1986. p. 24-27.

[HAR90] HARRISON, D.S. et al. Electronic CAD Frameworks. **Proceedings of the IEEE**, Piscataway, v. 78, n. 2, p. 393-417, February 1990.

[HAY83] HAYNIE, M.N. The Relational Data Model for Design Automation. In: DESIGN AUTOMATION CONFERENCE, 1983. **Proceedings**… Los Alamitos: IEEE Computer Society, 1983. p. 599-607.

[HEI87] HEILER, S. et al. An Object-Oriented Approach to Data Management: Why Design Databases Need It. In: DESIGN AUTOMATION CONFERENCE, 1987. **Proceedings**… Los Alamitos: IEEE Computer Society, 1987. p. 335-340.

[HER2001] HERNANDEZ, É. B.; SAWICKI, S.; INDRUSIAK, L. S.; REIS, R. A. L. Homero - Um Editor VHDL Cooperativo via Web. In: WORKSHOP IBERCHIP, 7., 2001, Montevideo. **Proceedings**… [S.l. s.n.], 2001.

[HÖR2001] HÖRMANN, S. et al. Ein Kurseditor für modularisierte Lernressourcen auf der Basis von Learning Objects Metadata zur Erstellung von adaptierbaren Kursen. In: LLWA, 2001. **Proceedings**… [S.l.: s.n.], 2001. p. 315-323.

[HOL94] HOLMEVIK, J.R. Compiling SIMULA: A Historical Study of Technological Genesis. **Annals of the History of Computing**, Los Alamitos, v.16, n.4, p. 25-37, 1994.

[IFIP 91] IFIP 10.2 WORKSHOP ON ELECTRONIC DESIGN AUTOMATION FRAMEWORKS, 2., 1990, Charlottesville, USA. **Proceedings**... Amsterdam: North-Holland, 1991.

[IND97] INDRUSIAK, L. S.; REIS, R. A. L. A WWW Approach For EDA Tool Integration In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, 10., 1997, Gramado, RS. **Proceedings**… Porto Alegre: CPGCC UFRGS, 1997.

[IND98] INDRUSIAK, L. S.; REIS, R. A. L. A Case Study For The Cave Project. In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, 11., 1998, Armação de Búzios, RJ. **Proceedings**… Los Alamitos: IEEE Computer Society, 1998.

[IND98a] INDRUSIAK, L. S. **Ambiente de Apoio Ao Projeto de Circuitos Integrados Utilizando World Wide Web.** 1998. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IND99] INDRUSIAK, L. S.; REIS, R. A. L. Project Management and Design Methodology Support for the Cave Project: A Hyperdocument-Centric Approach In: BRAZILIAN SYMPOSIUM ON INTEGRATED CIRCUITS DESIGN, 12., 1999, Natal, RN. **Proceedings**… Los Alamitos: IEEE Computer Society Press, 1999.

[IND2000] INDRUSIAK, L. S.; REIS, R.A.L. From a Hyperdocument-Centric to an Object-Oriented Approach for the Cave Project. In: SYMPOSIUM OF INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2000, Manaus. **Proceedings**... Los Alamitos: IEEE Computer Society Press, 2000.

[IND2000a] INDRUSIAK, L. S. **Architectural Evolution for the Cave Design Automation Framework**. Trabalho Individual. 2000. (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IND2001] INDRUSIAK, L. S. et al. Distributed Collaborative Design over Cave2 Framework. In: ROBERT, M. et al. (Ed.). **SOC Design Methodologies**. Dordrecht: Kluwer Academic Publishers, 2002. p. 97-108.

[IND2001a] INDRUSIAK, L.S.; REIS, R.A.L. 3D integrated circuit layout visualization using VRML. **Future Generation Computer Systems,** Amsterdam, v.17, n. 5, p. 503–511, Mar. 2001.

[IND2002] INDRUSIAK, L. S. **A Review on the Framework Technology Supporting Collaborative Design of Integrated Systems.** Exame de Qualificação. 2002. (Doutorado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[IND2003] INDRUSIAK, L. S.; LUBITZ, F.; GLESNER, M.; REIS, R. A. L. Ubiquitous Access to Reconfigurable Hardware: Application Scenarios

And Implementation Issues. In: DATE 2003 – DESIGN AUTOMATION AND TEST IN EUROPE, DATE, 2003, München. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003.

[JAC95] JACOME, M.F.; DIRECTOR, S.W. Planning and Managing Multi-disciplinary and Concurrent Design Processes. In: RAMMIG, F.J.; WAGNER, F. R. (Ed.). **Electronic Design Automation Frameworks.** London: Chapman & Hall, 1995. p. 159-168.

[JAO92] JACOBSON, I. et al. **Object-Oriented Software Engineering - A Use Case Driven Approach**. [S.l.]: ACM Press/Addison Wesley, 1992.

[JER99] JERRAYA, A.A. et al. Multilanguage Specification for System Design and Codesign. In: JERRAYA, A.; MERMET, J. (Ed.). **System-level Synthesis**. Dordrecht: Kluwer Academic Publishers, 1999.

[JOH88] JOHNSON, R.; FOOTE, B. Designing Reusable Classes. **Journal of Object-Oriented Programming**, Chatsworth, v.1, n. 2, p. 22-35, 1988.

[KAT86] KATZ, R. H. A Version Server for Computer-Aided Design Data. In: DESIGN AUTOMATION CONFERENCE, 1986. **Proceedings**… Los Alamitos: IEEE Computer Society, 1986. p. 27-33.

[KAT91] KATZ, R. H. Towards a unified framework for version modeling in engineering databases. **ACM Computing Surveys**, New York, v. 22, n. 4, p. 375-408, Dec. 1990.

[KME2003] K-MED. **Knowledge-Based Multimedia Medical Education**. Available at: <http://www.k-med.org>. Visited on Jan. 27, 2003.

[KOB99] KOBRYN, C. UML 2001: A Standardization Odyssey. **Communications of the ACM**, New York, v. 42, n. 10, p. 29-37, 1999.

[KOS2003] KOSTIENKO, T. et al. Development of TRMS/GTLS - Global Tool Lookup Services. In: CHALLENGES IN COLLABORATIVE ENGINEERING, Poznan, 2003. **Proceedings…** Poznan: Publishing House of Poznan University of Technology, 2003. p. 18-19.

[KRA91] KRAFT, N. Embedded Tool Encapsulation. **Electronic Design Automation Frameworks**, Amsterdam, v. 2, p. 9-20, 1991.

[KRS88] KRASNER, G. E.; POPE, S.T. A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. **Journal of Object-Oriented Programming**, Chatsworth, v. 1, n. 3, p. 26–49, August/September 1988.

[KWE95] KWEE-CHRISTOPH, E.; FELDBUSCH, F.; KUMAR, R.; KUNZMANN, A. Generic Design Flows for Project Management in a Framework Environment. In: EUROPEAN DESIGN AND TEST CONFERENCE, 1995, Paris. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995.

[LAV2000] LAVANA, H. **A Universally Configurable Architecture for Taskflow-Oriented Design of a Distributed Collaborative Computing Environment**. 2000. PhD Thesis - Electrical and Computer Engineering, North Carolina State University, Raleigh.

[LEE2001] LEE, E.A. et al. **Overview of the Ptolemy Project**. Berkeley: UC Berkeley, 2001. (Technical Memorandum UCB/ERL M01/11).

[LEO2001] LEONG, P. H. W. et al. Pilchard - A Reconfigurable Computing Platform with Memory Slot Interface. In: IEEE SYMPOSIUM ON FIELD-PROGRAMMABLE CUSTOM COMPUTING MACHINES, 2001. **Proceedings**… Los Alamitos: IEEE Computer Society, 2001.

[LIB2002] LIBELIS. **Universal Java Objects/Data Mapping**. Available at: <http://www.libelis.com>. Visited on Dec. 8, 2002.

[LIN97] LINDHOLM, T.; YELLIN, F. **The Java Virtual Machine Specification**. Reading: Addison-Wesley, 1997.

[LIS2000] LI, SING. **Professional Jini**. Birmingham: Wrox Press, 2000.

[LTS2002] LTSC - LEARNING TECHNOLOGY STANDARDIZATION COMMITTEE OF THE IEEE. **IEEE P1484.12/D6.1 Draft Standard for Learning Object Metadata**. Available at: <http://ltsc.ieee.org/doc>. Visited on May 12, 2003.

[MAN98] MANGIONE, C. Performance Tests Show Java as Fast as C++. **JavaWorld**, [S.l.], v. 3, n. 2, Feb. 1998. Available at: <http://www.javaworld.com/javaworld/jw-02-1998/jw-02-jperf_p.html>. Visited on Dec. 14, 2001.

[MAY2000] MAYER, U.; BECKER, J.; GLESNER, M.; HOLLSTEIN, T.; INDRUSIAK, L. S.; REIS, R. A. L. An Internet-Capable CAD Suite for the Mulit-Level Design of Complex Microelectronic Systems. In: DESIGN, AUTOMATION AND TEST IN EUROPE CONFERENCE, 2000, Paris. **Proceedings…** Los Alamitos: IEEE Computer Society, 2000. p.303.

[MIC2000] MICROSOFT CORPORATION. **Universal Plug and Play Device Architecture**. v. 1.0. 2000. Available at: <http://www.upnp.org/download/UPnPDA10_20000613.htm>. Visited on June 24, 2002.

[MIG2002] MIGNOLET, J-Y.; VERNALDE, S.; VERKEST, D.; LAUWEREINS, R. Enabling hardware-software multitasking on a reconfigurable computing platform for networked portable multimedia appliances. In: ERSA, 2002, Las Vegas. **Proceedings…** [S.l. s.n.], 2002.

[MUE2000] MUELLER, G.; BRAEUTIGAM, F. **Tutorial: Getting started with ozone**. The Ozone Database Project, 2000. Available at: <http://www.ozone-db.org>. Visited on July 8, 2001.

[MUL88] MULLE, J. A.; DITTRICH, K. R.; KOTZ, A. M. Design management support by advanced database facilities. IFIP WORKSHOP ON TOOL INTEGRATION AND DESIGN ENVIRONMENTS, 1987, Paderborn, Germany. **Proceedings...** Amsterdam: North-Holland, 1988.

[MUN96] MUNSON, J.; DEWAN, P. A concurrency control framework for collaborative systems. In: ACM CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK, 1996. **Proceedings…** [S.l.: s.n.], 1996. p. 278-287. Available at: <http://citeseer.nj.nec.com/munson96concurrency.html>. Visited on Dec. 14, 2002.

[MUR2003] MUELLER, W.; SCHATTKOWSKY, T.; EIKERLING, H.J.; WEGNER, J. Dynamic Tool Integration in Heterogeneous Computer Networks. In: DESIGN AUTOMATION AND TEST IN EUROPE, 2003, München. **Proceedings...** Los Alamitos: IEEE Computer Society, 2003.

[MUR2003a] MUELLER, W.; SCHATTKOWSKY, T. Distributed Engineering Environment for the Design of Electronic Systems. In: CHALLENGES IN COLLABORATIVE ENGINEERING, Poznan, 2003. **Proceedings...** Poznan: Publishing House of Poznan University of Technology, 2003. p. 16-17.

[NEW99] NEWTON, A.R. WELD **Project** - Web-based Electronic Design, 1999. Available at: <http://www-cad.eecs.berkeley.edu/Respep/Research/weld/>. Visited on July 26, 1999.

[OBJ2002] OBJECT MANAGEMENT GROUP. **Common Object Request Broker Architecture** (CORBA). v. 3.0. 2002. Available at: <http://www.omg.org>. Visited on Nov. 26, 2002.

[OPE2002] OPENEDA. **EDA Open Source Community.** Available at: <http://openeda.org>. Visited on Dec. 5, 2002.

[OST2001] OST, L. C.; MAINARDI, M. L.; INDRUSIAK, L. S.; REIS, R. A. L. Jale3D - Platform-independent IC/MEMS Layout Edition Tool In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 14., 2001, Pirenopolis. **Proceedings**... Los Alamitos: IEEE Computer Society, 2001. p.174 – 179.

[PIE96] PIERRE, G.; MAKPANGOU, M. A Flexible Hybrid Concurrency Control Model for Collaborative Applications in Large Scale Settings. In: ACM SIGOPS EUROPEAN WORKSHOP, Connemara, 1996. **Proceedings**… [S.l.: s.n.], 1996.

[PRE94] PREE, W. Metapatterns: A Means for Capturing the Essentials of Object-Oriented Design. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMMING, 10., 1994, Bologna, Italia. **Proceedings**... Berlin: Springer-Verlag, 1994. p.150-164.

[PRS96] PRESSMAN, R. S. Software **Engineering**: A Practitioner’s Approach. [S.l.]: McGraw-Hill, 1996.

[REI2000] REIS, R. et al. **Sistemas Digitales**: Síntese Física de Circuitos Integrados. Bogotá: Uniandes, 2000. 374 p.

[REI2000a] REIS, R. **Concepção de Circuitos Integrados**. Porto Alegre: Sagra Luzzato, 2000. 252 p.

[ROO2003] ROOS, R. M. **Java Data Objects**. London: Addison-Wesley, 2003. 264 p.

[ROW97] ROWSON, J. A. SANGIOVANNI-VICENTELLI, A. Interface-based Design. In: DESIGN AUTOMATION CONFERENCE, Anaheim,

1997. **Proceedings…** Los Alamitos: IEEE Computer Society, 1997. p. 178-183.

[RUB94] RUBIN, S. M. **Computer Aids for VLSI Design**. 2nd ed. [S.l.]: Static Free Software, 1994. Available at: <http://www.rulabinsky.com/cavd/>. Visited on May 28, 2001.

[RUM91] RUMBAUGH, J. et al. **Object-Oriented Modeling and Design**. [S.l..]: Prentice Hall, 1991.

[SAN2000] SANGIOVANNI-VICENTELLI, A. et al. System Level Design: Orthogonalization of Concerns and Platform-Based Design. **IEEE Transactions on Computer-Aided Design of Circuits and Systems**, v. 19, n. 12, December 2000.

[SAT2000] SATZINGER, J.W.; JACKSON, R.; BURD, S.D. **Systems Analysis and Design in a Changing World**. [S.l.]: Course Technology, 2000.

[SAW2002] SAWICKI, S.; BRISOLARA, L.B.; INDRUSIAK, L.S.; REIS, R.A.L. Collaborative Design using a Shared Object Spaces Infrastructure. In: SYMPOSIUM OF INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2002, Porto Alegre. **Proceedings...** Los Alamitos: IEEE Computer Society, 2002.

[SAW2002a] SAWICKI, S. **Projeto Cooperativo no Ambiente Cave Baseado em Espaço Compartilhado de Objetos**. Dissertacao de Mestrado. 2002. Dissertação (Mestrado em Ciência da Computação) – Instituto de Informática, UFRGS, Porto Alegre.

[SCH2002] SCHNEIDER, A.; IVASK, E.; MIKLOS, P.; RAIK, J.; DIENER, K.H.; UBAR, R.; CIBÁKOVÁ, T.; GRAMATOVÁ, E. Internet-Based Collaborative Test Generation with MOSCITO. In: DESIGN, AUTOMATION AND TEST IN EUROPE, Paris, 2002. **Proceedings**… Los Alamitos: IEEE Computer Society, 2002. p. 221-226.

[SCU95] SCHUBERT, J.; KUNZMANN, A.; ROSENTIEL, W. Reduced Design Time by Load Distribution with CAD Framework Methodology Information. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1995, Brighton. **Proceedings...** Los Alamitos: IEEE Computer Society, 1995.

[SHE93] SHERWANI, N.A. **Algorithms for VLSI physical design automation**. [S.l.]: Kluwer Academic Publishers, 1993.

[SHI2002] SHIN, S. **Jini(tm) Network Technology**. Available at: <http://www.plurb.com/misc/jini>. Visited on Sept. 7, 2002.

[SHN2002] SHEN, H.; SUN, C. Flexible notification for collaborative systems. In: ACM CONFERENCE ON COMPUTER SUPPORTED COOPERATIVE WORK, 2002. **Proceedings**… [S.l.: s.n.], 2002. p. 77-86.

[SIA99] SEMICONDUCTOR INDUSTRY ASSOCIATION. **International Technology Roadmap for Semiconductors**: 1999 edition. Austin: International SEMATECH, 1999.

[SII2002] SILICON INTEGRATION INITIATIVE INC. Available at: <http://www.si2.org/>. Visited on Oct. 3, 2002.

[STE2001] STEINMETZ, R. et al. Ein Java-basiertes Werkzeug für transparente Kollaboration über das Internet. **thema Forschung**, Darmstadt, v. 1, p. 56-61, Mar. 2001.

[SUN97] SUN MICROSYSTEMS INC. **Java(TM) Core Reflection** - API and Specification. 1997. Available at: <http://java.sun.com>. Visited on Apr. 21, 1999.

[SWA2001] SWAN, S. et al. **Functional Specification for SystemC 2.0**. 2001. Available at: <http://www.systemc.org>. Visited on Jan. 3, 2002.

[TEN91] TEN BOSCH, K.O.; BINGLEY, P.; VAN DER WOLF, P. Design Flow Management in the NELSIS CAD Framework. In: DESIGN AUTOMATION CONFERENCE, 1991. **Proceedings**… Los Alamitos: IEEE Computer Society, 1991. p.711-716.

[TRI90] TRIMBERGER, S.M. **An Introduction to CAD for VLSI**. San José: Domencloud Publishers, 1990.

[VAD88] VAN DER WOLF, P.; VAN LEUKEN, T.G.R. Object Type Oriented Data Modeling for VLSI Data Management. In: DESIGN AUTOMATION CONFERENCE, 1988. **Proceedings**… Los Alamitos: IEEE Computer Society, 1988. p. 351-356.

[VAD90] VAN DER WOLF, P.; BINGLEY, P. DEWILDE, P. On the Architecture of a CAD Framework: The NELSIS Approach. In: EUROPEAN DESIGN AUTOMATION CONFERENCE, 1990. **Proceedings**… [S.l.: s.n.], 1990. p. 29-33.

[VAE2000] VAN DER AALST, W.M.P. et al. Advanced Workflow Patterns. In: IFCIS INTERNATIONAL CONFERENCE ON COOPERATIVE INFORMATION SYSTEMS, 7., 2000. **Proceedings…** [S.l.: s.n.], 2000.

[VAN2001] VANBEKBERGEN, P. **CoDesign Strategies For SoC**. Available at <http://www.coware.com/ppt/ESC2001/sld001.htm>. Visited on Aug. 9, 2002.

[VAR2002] VANDERPERREN, Y. et al. A Design Methodology for the Development of a Complex System-on-Chip Using UML and Executable System Models. In: EUROPEAN SYSTEMC USERS GROUP MEETING, 6., 2002, Lago Maggiore. **Proceedings**… Available at: <http://www-ti.informatik.uni-tuebingen.de/~systemc/>. Visited on Dec. 5, 2002.

[VER2000] VERSANT CORPORATION. **Getting Started with Versant** - From UML to DBMS Using Java. White Paper. 2000. Available at: <http://www.versant.com>. Visited on Oct. 15, 2000.

[WAG91] WAGNER, F.R.; LIMA, A.H.V. Design Version Management in the GARDEN Framework. In : DESIGN AUTOMATION CONFERENCE, 1991. **Proceedings**… Los Alamitos: IEEE Computer Society, 1991. p. 704-710.

[WAG94] WAGNER, F.R. **Ambientes de Projeto de Sistemas Eletrônicos**. [S. l.. s.n.], 1994.

[WEN2001] WENTWORTH, S; LANGAN, D. D. Performance Evaluation: Java vs. C++. In: ANNUAL ACM SOUTHEAST CONFERENCE, 39., 2001, Athens, Georgia, 2001. **Proceedings…** [S.l.: s.n.], 2001. Available at <http://webster.cs.uga.edu/~jam/acm-se/review/referee/spw98.doc>. Visited on Aug. 13, 2002.

[WID88] WIDYA, I.; VAN LEUKEN, T.G.R.; VAN DER WOLF, P. Concurrency Control in a VLSI Design Database. In: DESIGN AUTOMATION CONFERENCE, 1988. **Proceedings**… Los Alamitos: IEEE Computer Society, 1988. p. 357-362.

[WIE2000] WIDENIUS, M.; AXMARK, D. **MySQL Reference Manual**. MySQL AB, 2000. Available at: <http://www.mysql.com/documentation>. Visited on July 26, 2000.

[WIL2000] WILLIAMS, L.; KESSLER, R. R. All I Really Need to Know about Pair Programming I Learned In Kindergarten. **Communications of the ACM**, New York, v. 43, n. 5, p. 108-114, 2000.

[ZHU2002] ZHU, X.; MALIK, S. A Hierarchical Modeling Framework for On-Chip Communication Architectures. In: IEEE/ACM INTERNATIONAL CONFERENCE ON CAD, 20., 2002, San Jose. **Proceedings…** Los Alamitos: IEEE Computer Society, 2002. p. 663-671.