Product Line Architecture for a Family of Meshing Tools

María Cecilia Bastarrica[1], Nancy Hitschfeld-Kahler[1], Pedro O. Rossel[1,2]

[1] Computer Science Department, FCFM, Universidad de Chile
Blanco Encalada 2120, Santiago, Chile
[2] Departamento de Computación e Informática, Universidad Católica del Maule
Avenida San Miguel 3605, Talca, Chile
{cecilia|nancy|prossel}@dcc.uchile.cl

Abstract. Meshing tools are extremely complex pieces of software. Traditionally, they have been built in a one by one basis, without systematically reusing already developed parts. The area has matured so that we can currently think of building meshing tools in a more industrial manner. We propose a layered product line architecture for meshing tools that can be instantiated with different algorithms, ways of implementing basic concepts, or even for two or three dimensional meshing tools. We specify it formally using xADL and we show that the architecture is compatible with a series of already built tools.

1 Introduction

A mesh is a discretization of a domain. This discretization can be either composed by a unique type of element, such as triangles, tetrahedra or hexahedra, or a combination of different types of elements. Meshing tools generate and manage these discretizations.

Meshing tools are inherently sophisticated software due to the complexity of the concepts involved, the big amount of interacting elements they manage, and the application domains where they are used. Meshing tools need to accomplish specific sophisticated functionality while still having an acceptable performance. Managing thousands and even millions of elements with a reasonable use of computational resources –mainly processor time and storage– becomes a must if the tool is to be usable at all. Lately, however, other qualities related to modifiability have become relevant in meshing tool development.

There are many application domains where meshing tools are used, ranging from mechanics design to medicine [11]. Each domain requires slightly different functionality. For this reason, a variety of meshing tools have been built differing on the functionality included, the algorithms used for implementing their functionality, the way data is represented, or the format of the data used as input or output. Also depending on the application domain, it may be required to have one, two or three dimensional meshes, each one maybe using different types of basic modeling elements.

Developing any complex software from scratch in a one by one basis is expensive, slow and error prone, but this is the way meshing tools have traditionally been built. If this development task is not performed in a systematic way using good software engineering practices, it may easily get out of control making it almost impossible to debug and even more difficult to modify. There have been some efforts lately applying software engineering concepts in meshing tool development, mainly building general purpose libraries that facilitate reuse. Also object-orientation and design patterns have the potential of enhancing software reuse at the code and design levels, and there is some experience in using these techniques for developing meshing tools.

The software architecture is one of the main artifacts developed during the software life cycle [14] because it determines most of the non-functional characteristics the resulting software will have, and it is also one of the most difficult documents to change once the software is deployed [2]. Architectural patterns [7] are used as guidelines for architectural design by reusing design knowledge at a high level of abstraction. Different architectural patterns promote different non-functional characteristics. In this way, for example, by using component and connector patterns such as client-server or repository, runtime properties can be modeled. Or using module patterns such as decomposition or layers, properties related to maintainability can be modeled [8].

Software product lines is a trend for planned massive reuse of software assets [9]. The most typical reusable assets are software components, but we can also reuse the product line architecture (PLA), software requirement documentation, and test cases, among others. The PLA is an important reusable asset because all software products in the family share the same design [6]. Therefore, the PLA design should be carefully approached making sure it will produce software that complies with the desired requirements.

In this paper we present the product line architecture for a family of meshing tools. Its design is based on the general architecture of published meshing products, as well as our own experience in building this type of tools. We intended to provide a PLA that would promote flexibility and extensibility, so that different existing algorithms, data structures, data formats and visualizers could be combined in different ways to produce a variety of different meshing tools appropriate for different application domains, sharing the software structure. The PLA is modeled following the layered architectural pattern [7]. This module view type is used for promoting modifiability, reusability and portability. Sometimes it is argued that layered architectures may penalize performance, but we have found that performance does not necessarily degrade significantly using the proposed PLA [17].

We formally define the PLA using xADL 2.0, an XML-based ADL specially designed to support the description of architectures as explicit collections of components and connectors [16]. There are graphical tools that make it easier to specify software architectures using xADL. xADL has also shown to be appropriate to specify product lines architectures [10].

We show how the proposed PLA can be instantiated for generating different meshing tools. In particular we show how already implemented tools can be seen as instantiations of our product family, independently of the methods followed for generating the meshes and the dimensions of the managed mesh.

The paper is structured as follows. In Section 2 we present and discuss concepts such as software architecture and software product lines and how they have been used in the development of meshing tools. Section 3 presents the proposed layered architecture for our product family of meshing tools, and Section 4 shows a series of different instantiations of this PLA to produce different meshing tools. Finally, in Section 5 we present some conclusions and describe our work in progress.

2 Related Work

The software architecture is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them [2]. Its building blocks are: components, connectors, and architectural configurations [18].

Software product lines (SPL) is a modern approach towards software development based on planned massive reuse. The idea is to provide a reuse infrastructure that supports a family of products, and to spend the resources in such a way that a high return of investment and reduce time-to-market can be achieved [25]. All elements subject to reuse are called core assets of the SPL. So, an SPL is a set of products that are built using core assets in a planned manner and that satisfy the needs of a market segment [9]. Opportunistic reuse does not usually work [6]; thus, assets in a SPL should be developed in such a way that reuse is promoted. This development process is longer and more expensive than developing one product at a time, but if assets are reused enough times, it is still cost-effective. Experience has shown that the costs of developing reusable assets is paid off after the second or third product is built [29]. The strategy for building software product lines is to identify commonalities, variabilities and optional modules.

There are several different architecture description languages (ADLs) [18], but not all of them are good for specifying PLAs.

In [5], an integrated notation for specifying software architectures in three levels of abstraction is proposed: structure, behavior and domain specific abstract data types. In [4] it is shown how to use this notation for defining a PLA. The notation helps in the process of identifying and localizing variations, but this it is not only non-standard for architecture specification, but also it has little tool support.

Koala is a software component model designed for creating software product lines for a large variety of products [27, 28]. Koala handles variation using composition, where selection of reusable components is bound in different ways to obtain different products. Koala was specifically created for modeling embedded systems. Mae is a technique, along with a supporting toolset, to formally spec-

ify, analyze, and evolve software architectures. It uses xADL 2.0 as an extensible notation to model the PLA as we do. We may use Mae in the future to face other development stages.

UML has become a standard notation for documenting software design. With the new UML 2.0 standard, some modeling elements specifically for software architectures were incorporated, but there are still no primitives for documenting connectors or architectural styles. However, there have been some efforts to extend UML in order to be able to use it as an ADL [22]. To our knowledge, UML has not been widely used for defining PLA. xADL improves on the UML approach in two significant ways: features and extensibility. With respect to features, xADL 2.0's type system and product-line support are abilities not present in UML 2.0 [10].

The use of software engineering principles in meshing tool design has spread only in the last five years. Some examples include the design of generic extensible geometry interfaces between CAD modelers and mesh generators [19, 21, 23, 26], the design of object-oriented data structures and procedural classes for mesh generation [20], and the computational geometry algorithm library CGAL [13]. Also recently it was publish a discussion on the usage of formal methods for improving reliability of meshing software [12]. There have also been some attempts in using software product family concepts for building meshing tools [3, 24].

To our knowledge, architectural patterns have neither been considered as a basis for designing PLAs nor for designing particular meshing tool architectures. Product line architectures must, by definition, be flexible to foster all products in the SPL, and promote modifiability so that variabilities could be incorporated. Therefore, it results natural to use module view type patterns [8], and more particularly a layered architectural pattern [7] as a guideline for designing the PLA.

3 Product Line Architecture

Independently of the application domain, any meshing tool may provide certain general functionality: read the domain geometry and physical values, generate an initial mesh, refine, derefine or smooth a mesh according to a quality criterion, and finally store the mesh into a file.

The PLA determines the product line scope limiting what products can be built, but at the same time it should be flexible enough to allow designers to build all desired tools. Flexibility and interchangeability are two of the non-functional requirements that guide our PLA design; this is why we chose a module view type architecture, and more precisely a layered architecture.

Figure 1 shows the structure of the meshing tool PLA. This architecture is specified using ArchStudio [1]. For simplicity we only include the connectors between layers and not those among modules within a layer even though they exist and they may be quite complex. Table 1 includes a general description of each type of component included in the PLA shown in Figure 1.

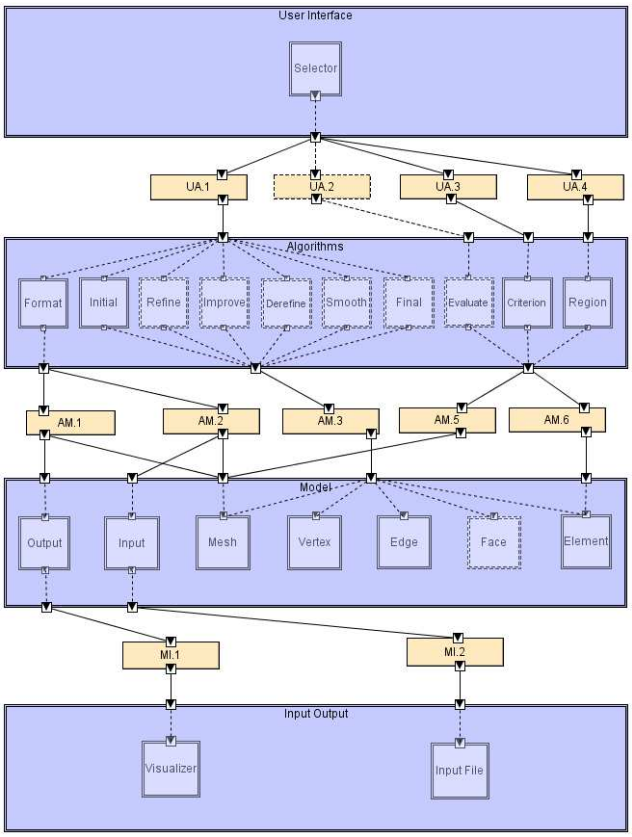

Fig. 1. Meshing Tool PLA

Refine and/or improve represent the core functionality of a meshing tool. In our PLA, both are presented as optional even though it may seem counter intuitive. Actually, at least one of them must be included in any tool instantiation. Though they represent different concepts, there are certain algorithms that perform both, so there are meshing tools that provide both functionalities only including one of them. There are other tools that prefer to use different algorithms for each one. This is why we give the opportunity of choosing different configurations. The `Face` module in the `Model` layer is also defined as optional. For all 3D tools there must exist a `Face` module, but it is meaningless for 2D tools.

The architecture is composed by four layers: *User Interface*, *Algorithms*, *Model* and *Input Output*. In xADL, each layer is defined as a structure. Figure 2 shows the xADL specification of the complete *Meshing Tool* PLA. Figure 3 shows the xADL specification of the `Refine` module. This module is included in the *Algorithms* structure.

Layer	Component Type	Description
User Interface	Selector	Menu for choosing the process to execute in the following step
Algorithms	Format	Translate the input geometry (domain) specified in any of the accepted formats in a normalized format.
	Initial	Generate an initial mesh of the domain
	Refine	Divide the mesh elements that do not satisfy the refinement criterion in the specified region
	Improve	Improve the mesh quality by dividing or reorganizing its elements according to the criterion in the specified region
	Derefine	Eliminate mesh elements according to a criterion in the specified region
	Smooth	Improve the quality of the elements by moving mesh points according to some criterion in the specified region
	Final	Apply a post-process to the complete mesh
	Evaluate	Generates statistics of the current mesh according to a quality criterion
	Criterion	Represents a geometric or physical quality that an element must fulfill. For example, the minimum angle of each element must be greater than 25° and/or the maximum edge length must be less than 2
	Region	Represents the part of the domain where the selected algorithm is applied to any element that does not fulfill the specified criterion
Model	Output	Gets the domain discretization and physical attributes and stores it in the required format
	Mesh	Contains the discretization of the domain. It is composed of elements, faces (only in 3D), edges and vertices
	Vertex	Represents a point of the discretization
	Edge	Represents a connection between two vertices
	Face	Represents the connections on an element surface. A triangular face is the one defined by three vertices or edges, and a rectangular face is defined by four vertices or edges
	Element	Represents a discretization cell. It can be a triangle or rectangle in 2D, or a tetrahedron or an hexahedron, among others, in 3D
	Input	Reads the domain description in a specific format and stores it as part of the mesh
Input Output	Visualizer	Tool that allows the visualization of the domain discretization and physical attributes
	InputFile	Contains the domain description in a format generated by a CAD tool

Table 1. Component types

As we can see in Figures 1 and 3, `Refine` exposes two interfaces, called *Refine.Top* and *Refine.Bottom*, respectively. The former has the direction *in*, and the latter has the direction *out*; this means that this component can be used by any component in the upper layer, and `Refine` may use other modules contained in the lower layer, following the rules of the layered architectural pattern [7].

According to Figure 1 where `Refine` is defined as an optional component, the xADL includes the *`options:optional`* tag indicating optionality.

4 Product Instantiation

In order to show the consistency of the proposed PLA, we present some products that may be part of the SPL.

The process of designing meshing products using the proposed PLA has two stages: component type selection and implementation selection. First, the component types that are to be included must be chosen; here some of the optional component types may not be included. In the second stage, a particular implementation needs to be chosen for every selected component type. In this

```
+ <types:archStructure types:id="Meshing Tool" xsi:type="types:ArchStructure">
+ <types:archStructure types:id="User Interface" xsi:type="types:ArchStructure">
+ <types:archStructure types:id="Algorithms" xsi:type="types:ArchStructure">
+ <types:archStructure types:id="Model" xsi:type="types:ArchStructure">
+ <types:archStructure types:id="Input Output" xsi:type="types:ArchStructure">
```

Fig. 2. Structures used in `Meshing Tool` Architecture

```
- <types:component types:id="Refine" xsi:type="types:Component">
    <types:description xsi:type="instance:Description">
        Refine module</types:description>
    - <types:interface types:id="Refine.Top" xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
            Top interface</types:description>
        <types:direction xsi:type="instance:Direction"> in</types:direction>
    </types:interface>
    - <types:interface types:id="Refine.Bottom" xsi:type="types:Interface">
        <types:description xsi:type="instance:Description">
            Bottom interface< /types:description>
        <types:direction xsi:type="instance:Direction"> out</types:direction>
    </types:interface>
    + <options:optional xsi:type="options:Optional">
</types:component>
```

Fig. 3. `Refine` Module Specification

way, different meshing tools may differ in their functionality (component types included) or in their implementation (concrete component implementation assigned to each component type).

Our SPL is oriented towards building tools for the generation of meshes required for solving problems numerically. The most widely used numerical methods for solving partial differential equations are: finite differences, control volumes, and finite elements. Typically mesh generators have been implemented using advancing front, octree or Delaunay algorithms. In Section 4.1 we present tools for generating finite element meshes and in Section 4.2 we present control volume meshes; in each case we present one example for 2D meshes and another one for 3D meshes.

4.1 Finite Element Meshes

For a large range of problems using the finite element method, isotropic meshes are required. The isotropy is measured in each mesh element geometrical properties, e.g. more equilateral elements are considered better than elements with too small or too large angles.

Simple 2D Triangulation Tool 2D triangulations require some of the component types identified as part of the Algorithms layer of the PLA in Figure 3.

In particular, a tool that generates Delaunay triangulations where all triangles have the minimum angle greater than a threshold value specified by the user, requires the component types described as part of Table 2. 2D triangulations do not require the `Face` component type, but all other component types in the Model layer must be included.

Component Type	Description
Selector	After generating the initial mesh, only the improvement algorithm can be selected letting the user to provide the minimum angle for the criterion to be applied
Initial	`Delaunay_algorithm` is used for generating the initial mesh
Improve	`Delaunay_improvement_algorithm` is used for improving
Criterion	`Minimum_angle` is used as a general criterion
Region	`Whole_geometry` is always used as the region where the improvement algorithm is applied

Table 2. 2D triangulation meshing tool (taken from [3])

Even though the `Format` component type is not optional, in this case it has a dummy functionality since the mesh is already read in its required format.

3D Tetrahedral Meshes In Table 3 the algorithms included in a particular 3D finite element mesh generator taken from [17] are described. This meshing tool allows the generation of 3D Delaunay and non-Delaunay meshes with a user specified point density and element quality. It also understands different input and output data formats. All component types included in the `Model` layer must also be realized as part of the tool, including `Face` since it is a 3D tool.

4.2 Control Volume Meshes

For the simulation of semiconductor devices using the control volume method, it is required to have anisotropic Delaunay conforming meshes where no part of a Voronoi region of an internal point is outside the domain. In 2D, this requirement is fulfilled if there is no obtuse angle opposite to boundary/interface edges. In 3D, for each boundary face the center of the smallest circumsphere must be inside the domain. In addition, too large angles in the interior of the domain and too high vertex edge connectivity must be avoided.

2D Triangulations In [3], a tool for the simulation of semiconductor devices is described. Here the mesh is read already in the format the tool is able to understand, so the `Format` component is assumed to have a dummy functionality. This tool is essentially used for improving and post-processing a mesh already generated and refined by another meshing generator. The specific component types chosen and their particular implementations are those described in Table 4.

Component Type	Description
Selector	After generating the initial mesh, the `Refine` and `Improve` components can be chosen several times
Initial	Generate a Delaunay mesh `GMVDelaunay`
Format	Translates the Off and Mesh formats into the appropriate format understandable by the meshing tool using `OffFormat` and `MeshFormat`, respectively
Refine	`LeppBisection` refines generally according to the longest edge criterion, or any other refinement criterion
Improve	`LeppDelaunay` improves the mesh with the `CircumRadiusEdgeRatio` criterion, or any other improvement criterion
Criterion	A set of different eligible criteria for refinement and improvement; e.g. `LongestEdge`, `CircumRadiusEdgeRatio`, `VolumeEdgeRatio`
Region	Region where the algorithm is applied; e.g. `WholeGeometry`, `Cube`

Table 3. 3D finite element mesh generator (taken from [17])

Component Type	Description
Selector	Allows to enter a specific improvement region and criterion, and also to choose the following algorithm to be applied (either `Improve` or `Final`)
Initial	Reads the already generated Delaunay mesh
Improve	Applies the `Delaunay_improvement_algorithm` to the specified region with a particular criterion
Final	Post-processes the mesh eliminating obtuse angles opposite to the boundary (`Non_obtuse_boundary_algorithm`)
Criterion	Improvement criteria such as `Maximum_edge_vertex_connectivity` and `Maximum_angle`
Region	Region where the improvement is applied; in the example only `Whole_geometry` is used, but it may also be `Circle`

Table 4. 2D control volume mesh (taken from [3])

3D Mixed Element Meshes A tool for semiconductor simulation in 3D is described in [15]. In this case, the mesh is composed of different types of elements, i.e. cuboides, prisms, pyramids and tetrahedra. The implementation is based on a modified octree approach. Even though this application was not developed with the product line concepts in mind, it fits within the PLA structure with little effort. The components included as part of the tool are described in Table 5.

5 Conclusion

Meshing tool construction has not always been approached using modern software engineering techniques, even though being sophisticated pieces of software makes them an appropriate application area.

Component Type	Description
Selector	Allows to enter a list of criteria and their associated regions, and then the whole process is invoked
Initial	Reads the device geometry and generates a first coarse mixed element mesh (`Fit_Device_Geometry`)
Refine	Divides element in order to fit physical and geometric parameter values (`Refine_Grid`)
Final	Improves elements in order to fulfill the Voronoi region requirement and generates the final mixed element mesh (`Make_Irregular_Leaves_Splittable`)
Region	Regions where the refinement is applied, e.g. cuboid or rectangle, among others
Criterion	`Doping_Difference` and `Longest_Edge` as the main refinement criteria
Format	Outputs the mesh in a format understandable by the visualizer (`Write_Geometrical_Information` and `Write_Doping_Information`)

Table 5. 3D control volume mesh (taken from [15])

The software product line approach intends to reuse all the artifacts that are built during software development in new products that fall within the SPL scope. The PLA is one of the most important assets in a SPL because it determines the non functional properties the resulting software will have. Having a well defined architecture allows us to integrate OTS components, either commercial or open source, such as the visualizer in our SPL case. This makes it evident that reusing is not only relevant for the software developed in house but also for software developed by third parties.

We proposed a layered PLA for a meshing tool SPL and we showed that a variety of diverse meshing tools could be considered to fit the proposed structure. By formally specifying the PLA using xADL, we were also able to iterate until we designed an architecture that was simple enough to be easily understood, while general enough to be able to capture the abstractions behind a wide variety of meshing tools. Having an integrated graphical and textual modeling tool such ArchStudio, helped greatly in this process.

The proposed PLA can be used as a road map to build almost any meshing tool. Different dimensions, algorithms, strategies and criteria will determine the concrete implementation of the component types identified as part of the PLA that will be part of each different meshing tool. We plan to build a more complete set of different implementations of the component types in the PLA and a software framework based on the PLA structure as a "meshing tool factory" for designing different tools that may be automatically built by combining the chosen implementation for each component type.

Acknowledgments

The work of Nancy Hitschfeld-Kahler was partially supported by Fondecyt Project N°1030672. The work of Pedro O. Rossel was partially supported by grant No. UCH 0109 from MECESUP, Chile.

References

1. ArchStudio 3. Architecture-Based Development Environment. Institute for Software Research, University of California, Irvine, 2005. http://www.isr.uci.edu/projects/archstudio/.
2. Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. SEI Series in Software Engineering. Addison-Wesley, 2^{nd} edition, 2003.
3. María Cecilia Bastarrica and Nancy Hitschfeld-Kahler. Designing a product family of meshing tools. *Advances in Engineering Software*, 37(1):1–10, January 2006.
4. María Cecilia Bastarrica, Marcelo López, Sergio F. Ochoa, and Pedro O. Rossel. Using the Integrated Notation for Defining a Product Line Architecture. In *Proceedings of the First Conference on the PRInciples of Software Engineering, PRISE'04*, Buenos Aires, Argentina, November 2004.
5. María Cecilia Bastarrica, Sergio F. Ochoa, and Pedro O. Rossel. Integrated Notation for Software Architecture Specifications. In *Proceedings of the XXIV International Conference of the Chilean Computer Science Society, SCCC'04*, pages 26–35, Arica, Chile, November 2004. IEEE Computer Society.
6. Jan Bosch. *Design and Use of Software Architectures. Adopting and Evolving a Product Line Approach.* Addison Wesley, first edition, May 2000.
7. Frank Buschmann, Regine Meunier, Hans Rohnert, and Peter Sommerlad. *Pattern Oriented Software Architecture: A System of Patterns.* John Wiley & Son Ltd., August 1996.
8. Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures. Views and Beyond.* SEI Series in Software Engineering. Addison Wesley, 2002.
9. Paul Clements and Linda M. Northrop. *Software Product Lines: Practices and Patterns.* Addison Wesley, first edition, August 2001.
10. Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. A Comprehensive Approach for the Development of Modular Software Architecture Description Languages. *ACM Transactions on Software Engineering and Methodology*, 14(2):199–245, 2005.
11. Rod W. Douglass, Graham F. Carey, David R. White, Glen A. Hansen, Yannis Kallinderis, and Nigel P. Weatherill. Current views on grid generation: summaries of a panel discussion. *Numerical Heat Transfer, Part B: Fundamentals*, 41:211–237, March 2002.
12. Ahmed H. ElSheikh, W. Spencer Smith, and Samir E. Chidiac. Semi-formal design of reliable mesh generation systems. *Advances in Engineering Software*, 35(12):827–841, 2004.
13. Andreas Fabri, Geert-Jan Giezeman, Lutz Kettner, Stefan Schirra, and Sven Schönherr. On the design of CGAL a computational geometry algorithms library. *Software - Practice and Experience*, 30(11):1167–1202, 2000.
14. Martin Fowler. Who Needs an Architect? *IEEE Software*, 20(5):11–13, 2003.

15. Nancy Hitschfeld, Paolo Conti, and Wolfgang Fichtner. Mixed Element Trees: A Generalization of Modified Octrees for the Generation of Meshes for the Simulation of Complex 3D Semiconductor Device Structures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 12(11):1714–1725, November 1993.
16. Rohit Khare, Michael Guntersdorfer, Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor. xADL: Enabling Architecture-Centric Tool Integration with XML. In *34^{th} Annual Hawaii International Conference on System Sciences (HICSS-34)*, Maui, Hawaii, January 2001. IEEE Computer Society.
17. Carlos Lillo. Analysis, Design and Implementation of an Object-Oriented System that allows to Build, Improve, Refine and Visualize 3D Objects. Master's thesis, Departamento de Ciencias de la Computación, Universidad de Chile, 2006. (in Spanish).
18. Nenad Medvidovic and Richard Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.
19. Silvio Merazzi, Edgar Gerteisen, and Andrey Mezentsev. A generic CAD-mesh interface. In *Proceedings of the 9^{th} Annual International Meshing Roundtable*, pages 361–370, October 2000.
20. Anton V. Mobley, Joseph R. Tristano, and Christopher M. Hawkings. An Object-Oriented Design for Mesh Generation and Operation Algorithms. In *Proceedings of the 10^{th} Annual International Meshing Roundtable*, Newport Beach, California, U.S.A., October 2001.
21. Malcolm Panthaki, Raikanta Sahu, and Walter Gerstle. An Object-Oriented Virtual Geometry Interface. In *Proceedings of the 6^{th} Annual International Meshing Roundtable*, pages 67–81, Park City, Utah, U.S.A., 1997.
22. Sunghwan Roh, Kyungrae Kim, and Taewoong Jeon. Architecture Modeling Language based on UML 2.0. In *Proceedings of the 11th Asia-Pacific Software Engineering Conference (APSEC 2004)*, pages 663–669, Busan, Korea, November 2004. IEEE Computer Society.
23. R. Bruce Simpson. Isolating Geometry in Mesh Programming. In *Proceedings of the 8^{th} International Meshing Roundtable*, pages 45–54, South Lake Tahoe, California, U.S.A., October 1999.
24. Spencer Smith and Chien-Hsien Chen. Commonality Analysis for Mesh Generating Systems. Technical Report CAS-04-10-SS, Department of Computing and Software, McMaster University, October 2004.
25. Anne Taulavuori, Eila Niemelä, and Päivi Kallio. Component documentation—a key issue in software product lines. *Information and Software Technology*, 46(8):535–546, 2004.
26. Timothy J. Tautges. The common geometry module (CGM): A generic, extensible, geometry interface. In *Proceedings of the 9^{th} Annual International Meshing Roundtable*, pages 337–347, New Orleans, U.S.A., October 2000.
27. Rob C. van Ommering. Building product populations with sofware components. In *Proceedings of the 22rd International Conference on Software Engineering, ICSE 2002*, pages 255–265, Orlando, Florida, USA, May 2002. ACM.
28. Rob C. van Ommering, Frank van der Linden, Jeff Kramer, and Jeff Magee. The Koala Component Model for Consumer Electronics Software. *IEEE Computer*, 33(3):78–85, 2000.
29. David M. Weiss and Chi Tau Robert Lai. *Software Product-Line Engineering: A Family Based Software Development Process*. Addison-Wesley, 1999.