Simulation Components in the Open Reflective Component Architecture

H.Hadler*, J.Treibig†, M.Kellner‡, T.Jung⋆
horst.hadler@informatik.uni-erlangen.de
*Department of Computer Graphics, University Erlangen-Nuremberg
Am Weichselgarten 9, 91058 Erlangen-Tennenlohe
†Department of System Simulation, University Erlangen-Nuremberg
‡Department of Material Science 6, University Erlangen-Nuremberg
⋆Crystal Growth Laboratory, Fraunhofer Institute IISB

Abstract

The Open Reflective Component Architecture (ORCAN) framework [2] is a collection of C++ libraries for building applications from run-time interchangeable components. It has been developed as part of a software system for the simulation of the heat transfer inside crystal growth furnaces and is primarily designed to assist a longterm distributed software development process. The software is using so-called *reflective components* [1] for the constitutive parts of the application. While reflective component frameworks are usually based upon interpreted programming languages, ORCAN is completely C++ based. The base component system is contained within a single, portable library that is available for all standard platforms. It is designed to be used with as little 'programming overhead' as possible and introduces an interface design that enables the software developer to change and exchange parts of the software without side effects.

While the framework itself can be used for arbitrary applications, a set of components named ORCAN/Sim has been defined to be used for standard simulation purposes. They decompose the simulation process into a set of inter-operating parts, each providing a specific functionality. Existing components provide mesh management, grid generation, linear equation solving, coupling, data visualization, etc. The development process is made up of connecting and configuring existing components and of introducing new component implementations when required by the simulation.

1 ORCAN/Sim

Components can be viewed as a means to decompose a complex computational model into manageable parts. The use of reflective components enforces the developer to think of the model's parts as replaceable entities. They can be added, removed and exchanged with respect to the problem being solved, the hardware available or the accuracy required. In the following the utilization of ORCAN/Sim's components is presented.

One principle of component-based applications can be described as "What You Need Is What You Get", meaning an application will at runtime acquire exactly the component

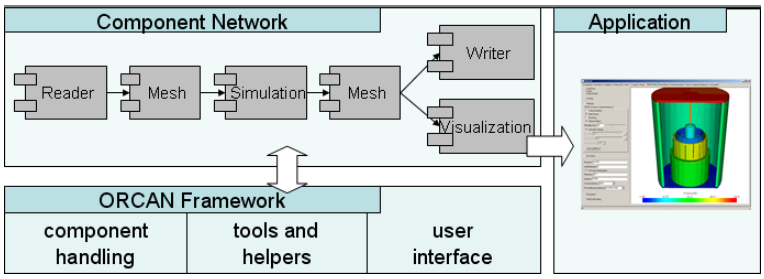

Figure 1: structure of application built from network of ORCAN/Sim components

implementations that it needs to perform its task. To do so, in contrast to conventional applications, two steps have to be performed: first, a component implementation has to be acquired and second, an *interface query* has to be performed, in which the implementation is queried for the functions needed by the application. The first step is delegated to a so-called *broker* which - from the application's point of view - creates an object that is used to access the component implementation. In the second step this object is used for the acquisition of interfaces, i.e. a set of functions, that can be called by other components or the main application to perform the requested task.

Likewise to conventional applications, the interface design of the program's modules is crucial to its maintainability and extensibility. In ORCAN/Sim each component's interface is naturally split into a generic part that is used to manage all parts that are likely to differ between implementations of the same component type and a fixed part defining the functionality common to all possible implementations. For example different grid generators for volume meshes can be subsumed to have a fixed interface with three functions. One function to set the input surface description, one function to specify the output volume mesh and another function that executes the grid generation process. All other variables and functions, that are specific to each grid generator implementation, are managed by a generic interface, that is available to all components by default. This separation enables keeping the interfaces as general as possible. Many of ORCAN/Sim's components have compact '*SetInput*, *SetOutput*, *Execute*'-style interfaces.

An exception to the components with very compact interfaces is the mesh management component, commonly used by other components for input and output. It stores mesh topology and data. The mesh component is not based upon a fixed 'per vertex' or 'per element' data layout, but it defines an interface for each possible way of storing data in the mesh. There is no limitation on the type and number of data that can be stored in the mesh, as long as a byte-wise copy is possible. For example, to associate a coordinate with each vertex, the 'vertex attribute' interface should be used to create an appropriate "3 floats"-type data element. The attribute is assigned a unique identifier. The coordinate of the vertex can then be accessed using this identifier and the vertex ID.

A simple application, using mesh attributes for component interaction, can be constructed as follows (Figure 1): First, the input model is processed by a reader component

to produce a mesh, which is then passed on to a component, actually performing the simulation and storing its results as attributes on the mesh. For feedback the mesh can be input to a visualization component, allowing all mesh attributes to be displayed using standard visualization algorithms. Finally the results can be stored using a writer component. An interactive control of the simulation process is automatically available, as the framework provides a built-in mechanism for creating a user interface for each component implementation at run-time. This mechanism is based upon the generic interface.

In it's current configuration ORCAN/Sim defines components for a variety of simulation sub-tasks, ranging from basic mesh management to material databases. Table 1 shows an overview of the component types that are available in ORCAN/Sim. The components have been designed and implemented in parallel by the authors.

Component	**Function**
Geometry	geometry management; CAD data processing
SurfMesh	surface mesh management; represent surfaces and surface data
VolMesh	volume mesh management; represent volumes and volume data
Surf/VolMeshReader/Writer	read/write surface/volume meshes and data
LESSolver	solve linear equation systems
PDEDiscretizer	numerical solving of a PDE
Radiation	radiation solver
Coupling	physical correct transmission of data between meshes
OutputContext	device abstraction layer for visualization
Visualization	visualization of meshes and mesh data
Material & MaterialDB	material data & database for material data
VolMeshGen	3D grid generator

Table 1: ORCAN/Sim component types

The framework provides a set of convenience functions and macros for adding new component implementations and new component types. To introduce a new implementation, a class deriving from the component type's base class has to be defined and to be compiled into a shared library. This class must also derive from each interface it is offering. The interfaces are defined as standard C++ sets of pure virtual functions. Additionally the macro `OC_REGISTER_REALIZATION` has to be called once, anywhere in the implementation for assigning a "footprint" to itself that can at run-time be used by the broker to identify it's type and name. Likewise the macro `OC_REGISTER_INTERFACE` has to be called once for every interface to be able to query the implementation's functionality at run-time.

Within the main application the framework isolates the user from the details of accessing the broker for loading and creating component implementations. For example `VolMeshGen::New()` will return a `VolMeshGenRef` object encapsulating a *VolMeshGen* implementation. The interface query is directly performed with this object, which defines a member variable for each possible interface of the *VolMeshGen* component type, initialized with a non-zero value if it is defined by the implementation.

2 Example: A simulation tool for coupled heat conduction and radiation

We are developing a software for the simulation of crystal growth processes using 3D models of crystal growth furnaces. The software is intended to supplement an existing software package [3] for 2D axisymmetric models. In its present state the simulation process of the 3D software includes the input and conditioning of CAD data, the generation of unstructured volume meshes for the volumes forming the furnace, the assignment of material data and boundary values, the simulation of global heat transfer and the visualization of simulation data. Each of this sub-tasks is managed by a network of linked ORCAN/Sim components.

Within the heat transfer simulation network coupled heat conduction and radiation is computed using a loop-based iterative approach. The default conduction component computes heat transfer by stationary conduction

$$\nabla q(\vec{x}, T) = s \tag{1}$$

with source terms s and flux densities $q = -\lambda(T)\nabla T$. It uses a finite element approach to solve for temperature at the vertices of the finite element mesh, resulting in a linear equation system, solved by the *LESSolver* component. In a subsequent step the vertex temperatures are integrated for each boundary surface element to obtain the emitted flux q_i^{em} of each element i. These are input to the heat radiation solver.

The default radiation component computes isotropic surface-to-surface radiation. The emitted radiative flux of a surface element i is given as

$$q_i^{em} = \epsilon_i \sigma A_i T_i^4 \qquad 0 < \epsilon_i < 1. \tag{2}$$

ϵ_i is the emissivity, A_i the area and $\sigma = 5.66710^{-8} W m^{-2} K^{-4}$ the constant of Stefan-Boltzmann. The flux leaving element i is the sum of the emitted flux and the reflected part of the incoming flux

$$q_i^{out} = (1 - \epsilon_i) q_i^{in} + q_i^{em} \tag{3}$$

The incoming flux q_i^{in} is the sum over the fractions of the out-coming flux q_j^{out} of all surface elements j which reach the element i. The fractions are given by the view-factors f_{ij} between each pair of elements i and j.

$$q_i^{in} = \sum_j f_{ij} q_j^{out} \tag{4}$$

Basically heat conduction and radiation can be coupled by using a single global solver matrix for both effects. However, this approach is not feasible for general 3D geometries because of the $O(n^2)$ complexity of surface-to-surface radiation. Therefore the task of computing radiation is assigned to the radiation component, which may internally use any method (hierarchical, monte carlo, ...) to compute the radiation in the models cavities. This requires additional computations, in which the radiation values are transmitted forth and back between the internal representation and the conduction mesh.

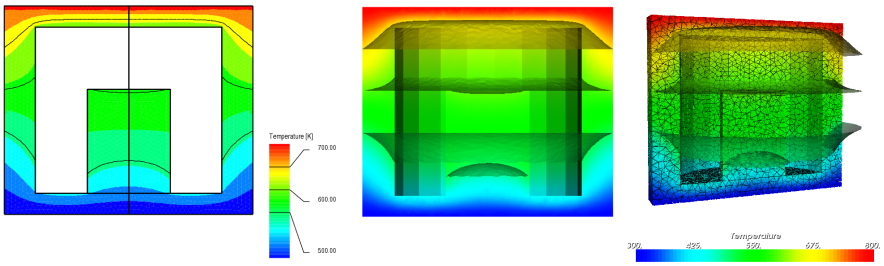

Figure 2: comparison of 2D axisymmetric and 3D computations for axisymmetric model

For coupling between radiation and conduction we are using the surface shared by the meshes of both solvers for exchanging boundary values in both directions in an iterative process. Starting with a solution from the conduction component, the matrix and the conduction mesh - with the resulting temperatures - are passed on to the radiation component. With the help of a coupling component, the input temperature values are converted into emitted flux density values at the surface elements of the radiation mesh. The radiation component then computes the radiation equilibrium. Finally the net-flux values at the radiation meshes surface elements are transfered to the conduction mesh as boundary values for the next iteration step and the matrix values are updated. This process is repeated until convergence i.e. the change in net-flux is below a user-given residual. Because convergence is slow when conduction and radiation are solved completely independent the radiation component is designated to make entries into the conduction matrix.

Our approach yields the expected results when compared to axisymmetric reference examples that can be solved using a single global matrix (Figure 2). It can however be used for general 3D geometries. The left part of figure 3 shows a cross section of a model of a Czochralski crystal growth furnace; from top to bottom: pulling rod, seed, crystal, melt, crucible, surrounded by heater and crucible support. The furnace is modeled from 35 volumes, representing regions of the furnace with different materials. Figure 3 shows the mesh and the result temperature distribution of a coupled conduction/radiation simulation of the furnace model. The mesh has a total of 200.000 volume elements and 80.000 surface elements used in radiation calculation, which results in 6.4 billion possible surface-to-surface interactions. The total computation time was 2 hours on Pentium 3GHz PC with a total of 300 conduction/radiation iterations.

3 Discussion

ORCAN/Sim provides a set of components for simulation sub-tasks that decompose complex simulation applications. By design, the component based approach implemented by the ORCAN framework enables the exchangeability of components, the rapid incorporation of external packages and, compared to conventional applications, an improved software maintenance. This is because a component implementation is unaware of other component

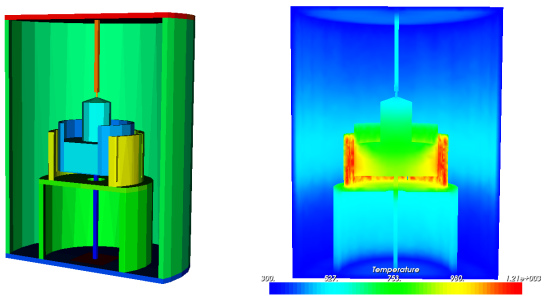

Figure 3: Czochralski crystal growth furnace and coupled radiation/conduction simulation

implementations, but provides its functionality as a ”black box”. It can be developed and compiled fully self-contained, which is made possible by the decomposition of the component’s interface into a fixed *general* part and a generic *implementation-specific* part. The component’s fixed interface, which is linking it to other components, can be kept very compact.

The utilization of components does however introduce extra levels of indirection to the application. Before an interface’s functions can be called, a component implementation has to be loaded and an *interface query* has to be performed. Furthermore the requested interface may not be available, requiring additional costs in the main application. By employing components the same data may be held several times in memory. However, as every simulation component creates its own internal data structure - regardless of other program parts - it can do so in the most efficient way.

The ORCAN framework and most of the ORCAN/Sim component implementations can be downloaded from *http://sourceforge.net/projects/orcan*.

4 Acknowledgements

This work was supported by the German Federal Ministry of Education and Research (BMBF) Grant Number 0327324A.

References

[1] *C. Szyperski:* ”Component Software, Second Edition”, New York, Pearsons Education Limited, 2002.

[2] *H. Hadler, M. Kellner, R. Grosso:* ”A Reflective Component Framework for a Large Scale Simulation Software”, ECTS2004 Proceedings, Lisbon, Portugal, 2004.

[3] *M. Kurz:* ”Development of CrysVUn++, a Software System for Numerical Modelling and Control of Industrial Crystal Growth Processes”, Dissertation, Erlangen, 1998.