A Toolkit for the Code Development in Advanced Computing

Atanas Atanasov, Hans-Joachim Bungartz and Tobias Weinzierl

TUM-I1330

A TOOLKIT FOR THE CODE DEVELOPMENT IN ADVANCED COMPUTING

ATANAS ATANASOV , HANS-JOACHIM BUNGARTZ , AND TOBIAS WEINZIERL*

Abstract. We propose to integrate the development of component-based simulation codes for computational sciences and engineering (CSE), the manufacturing of problem solving environments, and the experimentation in one toolkit in one place. Such a tool can cover the whole application lifecycle in advanced computing from the software design and development over simulation to data postprocessing and exploration. Our prototypical realisation based upon a simplified version of the Common Component Architecture shows that such a holistic approach's impact on the software development process is multifaceted: It simplifies and accelerates the traditional code development due to synergies of software engineering tools well-established in mainstream computing with scientific computing characteristics. It helps to make the CSE software development more agile due to the support of continuous integration and the bridging of different simulation activities. It finally fosters and stimulates to add features such as distributed simulation or computational steering facilities into simulation codes without a significant overhead. All these ingredients help to overcome the CSE software crisis becoming oppressing due to increasing hardware and software complexity as well as increasing functional complexity, requirements, and needs.

Key words. Component-based Programming, Integrated Development Environments, Advanced Computing, Computational Steering, Problem Solving Environments

AMS subject classifications. 68M14,68W05,68N19

1. Introduction.

Methodological improvements of the software development process in computational sciences and engineering (CSE) often are influenced by three considerations: First, software has to reflect a paradigm shift from pure forward and batch processing towards interactive usage with interacting software artifacts. The long-term impact of this shift is covered by the notion of the term computational steering. Second, software has to reflect a paradigm shift from single location runs to simulations distributed among computing centres. If the trend continues and if it is augmented by the fact that it often does not matter where a simulation runs, the long-term impact of this shift is covered by the term Grid computing. Finally, software has to pick up the experience that a complex artifact benefits from black-box components independently assembled, designed, developed, and tested before. Otherwise the software complexity is hardly manageable [15, 30, 32] and cannot master upcoming challenges [21]. Yet, software fragments often are written from scratch for each project—apart from some exceptions proving the rule. These three considerations are accompanied with the need for tools supporting software development, deployment, scientific experiments, and postprocessing as well as data exploration.

Component-based architectures transforming huge monolithic applications into an assembly of autonomous small pieces of software are one way to pick up the methodological considerations from above. Their "divide et impera" approach mirrors mainstream trends from service- and object-based programming: Components control and steer each other, they can be distributed among different computers, and the time-to-software-delivery is reduced, as the maintainability and maturity of prefabricated software artifacts are typically higher than those of a first-time shot. CSE tell several component success stories: Linear algebra libraries starting from BLAS up to PETSc or Trilinos are widely accepted and used, MPI is the de-facto standard for dis-

*Scientific Computing, Department of Informatics, Technische Universität München, Boltzmannstraße 3, 85748 Garching, Germany ({atanasoa,bungartz,weinzier}@in.tum.de).

tributed memory communication, and graph parallelisation libraries such as METIS are used frequently. In particular the visualisation community yields some shining stars of component architectures ranging from visualisation tools such as [1, 42] with a pipeline layout to problem solving environments (PSEs) such as [2, 3] with visual languages [43].

At the same time, component standards such as the Common Component Architecture (CCA) [4] that are not niched, do not restrict component communication to data flow, and promise a holistic and general-purpose architectural approach are neither mainstream ([27] or conclusion in [12]) nor used for the majority of scientific computing projects despite promising use cases [12, 27, 29, 30, 32]. Component architectures face multifaceted concerns ranging from language likes and dislikes to runtime overhead discussions. Different to mainstream computing where component-based software engineering is state-of-the-art, performance often is considered to be the fundamental different challenge, and endeavours such as CCA hence emphasise its importance [12, 29, 30, 32]. Severe though is also a chicken-and-egg problem. If a data-flow paradigm with fire-and-forget semantics [42] is not fitting to a developer's needs or if the functional requirements are exotic, often only few out-of-the-box components are available. At the same time, refactoring components out of an existing software base is time-consuming and involves reengineering while a-priori design towards components is more difficult than tailoring a piece of software to one concrete project. The upfront investment does not pay off. Additional effort to write for reuse and additional complexity due to the component paradigm are the enemy of a valuable idea. Consequently, it is both important to give the developers time to adopt component techniques and to allay performance concerns as well as to lower the component development threshold. Only a joint endeavour leads to a higher productivity in the field [26].

In this paper, we pick up the CCA and propose ideas how to make component development for computational sciences and engineering smoother, easier, and more agile. We follow the CCA notion of components, ports, and frameworks that is an extended method invocation paradigm. However, we restrict ourselves to a simplified variant of the standard, and then present a new component workbench for it. The workbench in turn is our basis to construct problem solving environments for concrete simulation and computational steering applications, and to propose ideas how to simplify and facilitate the component development and CSE workflow. The idea's unique selling point is the combination of five aspects: First, it seamlessly integrates into the development cycle, as it is an Eclipse plugin [5]. With such an integrated development environment (IDE) for formal component specifications, C/C++, Java, and FORTRAN at hand, the developer can switch forth and back from coding activities, refactoring, source control activities, and profiling to component deployment, simulation assembly, experiments, and result exploration. This is continuous integration [14]. Second, it is easy to write new components. Our component specification language also is a rigorous subset of the scientific interface definition language (SIDL) [20, 27], and our Eclipse plugin generates any glue code for C++, Java, or remote executables in the background, i.e. hidden from the user. Third, the workbench homogenises everything visually into one place, as the user interface follows the Eclipse look-and-feel and integrates for example VTK [1] and remote visualisation capabilities directly into the graphical user interface. Fourth, it offers the opportunity to deploy components to remote computers within the workbench seamlessly, i.e. to the workbench user it is hidden whether a component runs locally or at a different place.

This is particularly relevant for executables running in parallel in their own process on a specific remote cluster while the problem solving environment typically is hosted on a local workstation or laptop. Finally, it offers the opportunity to deploy a problem solving environment built on top of the applications to an application expert due to Eclipse's rich client platform facilities, as the Eclipse plugin already includes the complete CCA facilities. No additional middleware has to be installed on the user's computer. The present approach combines ideas of standardised component architectures and component servers, problem solving environments, and IDEs. It is designed to support, speedup, and agilise all phases of advanced scientific software engineering—from the design of simulation codes on the component level and coding, i.e. classical scientific computing, over assembling, and running the experiments (on the large scale) to visualisation, data postprocessing, and data exploration. We thus call it an advanced scientific computing development toolkit (ASCoDT).

Both visual languages and component-based architectures look back to a long tradition. Webservices and interoperability standards are mainstream in many application areas, and they influence any considerations with respect to inter-language operability in scientific computing. Workflow management systems are of value for some scientific applications such as embarrassingly parallel parameter studies or Monte-Carlo simulations (see [31] in the present paper's context). However, few scientific computing centres will, in the near future, run standardised, commercial web services meeting the low-overhead requirements of CSE. Our approach facilitates the development of tailored services. Babel [20] is the de-facto compiler and runtime environment for SIDL applications due to its extensive feature list and support for several languages. For our case studies, complex datatypes, interface version control, support of not-objected-oriented languages are not required. At the same time, we want to study communication aspects such as remote coupling in a trial-and-error prototype manner without overhead. We thus work in the present paper with a small subset of SIDL, a small in-house source-to-source SIDL compiler, and without any middleware—well-aware that this decision might not stand the test of time. Out-of-the-box CCA frameworks (cf. [27, 32] and references therein, or [4, 6, 12, 20, 44]) support the full functionality of Babel/SIDL, and each of these systems has its own unique selling points such as support of massively distributed memory parallelisation or Grid techniques. In the present paper, we propose to embed the component workbench into an existing integrated development environment (Eclipse) to reduce context and application switches and to increase the programmer's productivity due to synergies with other tools. To reach this goal prototypically fast and to study its impact on software engineering in CSE, we realised ASCoDT from scratch. Finally, the tools with the greatest maturity in the field are most probably visualisation-centred tools such as [2, 3, 19, 36]. On the one hand, competing with their functionality in term of visualisation and exploration is almost impossible due to long evolution and due to the enormous amount of available standard modules. On the other hand, these tools emphasise data-flow communication, i.e. fire-and-forget semantics. Because of the present focus on steering and component engineering, the more powerful, classical method invocation paradigm is the method of our choice. It still can mirror data-flow mechanisms.

ASCoDT is not a mature product but a case study. First, it demonstrates how to integrate a CCA-like workbench into existing development environments. To our knowledge, this has not been done before. Second, it generates all CCA glue code in the background. Such a behaviour could easily be realised due to Babel with a building deamon running in the background for a different framework. We just give a proof of concept, and we merge agile development and component-oriented scientific computing. Third, ASCoDT integrates visualisation devices into the programming environment. Such ideas have a long tradition, but have not been studied in the context of CCA. Here, we present a prototype covering the whole process from software specification and coding to postprocessing and visualisation. Fourth, the integration of (Grid) deployment mechanisms into the component workbench is a natural straightforward approach. It could also be realised within another workbench, but also usually is not integrated into the code development environment. Finally, the idea to deploy an environment tailored to a specific problem is a straightforward approach to bridge the gap from software development to problem solving environments and to smooth the scientific code development and numerical experiment process.

The remainder is organised as follows: We first introduce our simplified variant of CCA and generated glue code in Section 2. In Section 3, we present our graphical user interface and component management realised as Eclipse plugin. This presentation comprises the architecture, our idea of seamless integration and on-the-fly compilation, as well as the interplay of a virtual scientific computing laboratory with other software engineering tools such as source control, profilers, and auto documentation. The workbench acts as steering layer on top of the real components interacting with each other (Section 4) due to different communication objects. Two case studies with computational steering present our software ASCoDT in action (Sections 5.1 and 5.2), before a conclusion and an outlook summarise the paper and close the discussion.

2. Simplified CCA variant. A component in ASCoDT is an autonomous software entity with a state and an interface. Multiple instances of one component may exist. Hence, a component is similar to an object of a class, and we use class as synonym for component definitions. Our components can be distributed among different computers running in processes or applications, respectively, of their own. The static structure of our applications consists of interfaces and classes, i.e. components interacting due to these interfaces. Both are organised in packages mirroring the namespace concept of C++ and Java. Interfaces prescribe a signature, i.e. comprise methods, and they can extend other interfaces. A class implements interfaces and uses interfaces. Whenever a class implements an interface, the class has a provides port. The port's type is given by the interface. Whenever a class uses another interface, the class has a uses port. The port's type is given by the interface. This static structure in our case is written down in a simplified scientific interface definition language (simplified SIDL).

Our SIDL dialect comprises only three top-level keywords: `package`, `class`, and `interface`, and the dialect's basis blocks are framed by curly brackets. Interfaces may extend other interfaces due to an `extends` statement. They always are implemented completely, i.e. we do not support the concept of partial implementation. While the language provides the possibility to specify component signatures and relationships in terms of interfaces, concepts such as attributes, version numbers, collaboration cardinality constraints, functions tied to classes, and so forth are not supported. An interface prescribes a set of named operations. They have no return type, but they accept a sequence of arguments. Each argument either is a pure incoming (in) argument, i.e. the component does not change it, or can be modified by the component (inout). Furthermore, it has a name, a type, and a cardinality. The type is either integer, boolean, double, or string. Complex or user-defined datatypes are not supported, but overloading, i.e. having several operations with the same name but different argument

Algorithm 1 Simplified dialect of the scientific interface definition language (SIDL).

```
package pa {
  package pb {
    interface Interface1 {
      foo( in bool a, inout int b );
      bar( in bool a[], in string b );
    }

    interface Interface2 { ... }
    interface Interface3 { ... }
    interface Interface4 { ... }
  }
}

package pc {
  class MyClass
    implements-all pa.pb.Interface1, pa.pb.Interface2
    uses pa.pb.Interface3 as LogToTerminal,
         pa.pb.Interface3 as LogToFile,
         pa.pb.Interface4 as UserUsesPort {
    }
}
```

Algorithm 2 Plain C++ mapping of SIDL file. The developer than just has to implement the abstract types.

```
namespace pa {
  namespace pb {
    class Interface1;
    ...
  }
}

namespace pc {
  class MyClass: public pa::pb::Interface1, pa::pb::Interface2 {
    private:
      pa::pb::Interface3* LogToTerminal;
      pa::pb::Interface3* LogToFile;
      pa::pb::Interface4* UserUsesPort;
    public:
      virtual void foo( const bool& a, int& b ) = 0;
      virtual void bar( bool* a, const std::string& b ) = 0;
  };
}
```

numbers or argument types, is possible. The cardinality of an argument by default is one, but our SIDL variant also supports one-dimensional arrays of variable length identified by a `[]` postfix. We avoid discussions on row-major or column-major as only one-dimensional arrays do exist. In our simplified CCA, the set of interfaces used by a class and provided by a class is static and known a priori at the specification time. A class may have multiple uses ports of the same interface. They are distinguished in the SIDL file due to an `as` statement. In C++ and Java, uses ports consequently are mapped to plain attributes, and port registry mechanisms [12] are not required. A component has no control whether or to how many components (see discussion later on) a uses port is connected. We provide a source-to-source compiler generating C++ or Java from SIDL specifications. The source code mapping transcribes every concept one to one to the destination language (the Algorithms 1 and 2 illustrate a SIDL specification and the corresponding C++ code). Different to Babel [20], the generated code does not require a runtime system or additional libraries to link to.

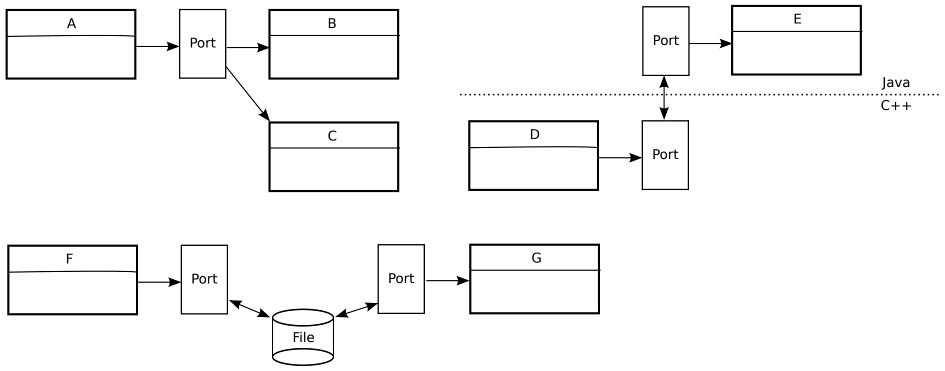


Fig. 2.1. *Ports are plain classes connecting one component with one or multiple other components (dispatcher port; components A,B, and C). Each non-Java port has its Java counterpart linked due to JNI (components D and E). There is also generated port variants in both C++ and Java that stream method calls to a file or parse method invocations from a file (components F and G).*

Each interface is mapped to a port interface in the destination language, and the compiler generates a couple of standard implementations of each port. While components could theoretically use other components directly, the ASCoDT architecture plugs at least one port object in-between two components for each component interaction (Figure 2.1). Cascades of different ports allow the component assembly to yield sophisticated behavioural patterns. The following standard implementations of a port are generated:

- A plain port that forwards each call to the destination component. It mirrors a one-to-one component composition. Depending on the implementation language of the component, this port either is a C++ or Java class.
- A Java version of this port if the destination language is not Java.
- A dispatcher port class holding several aggregates implementing the port's signature. Each time a port operation is called, the port distributes this call to all the connected components or ports, respectively. A dispatcher port is provided if and only if all the operations of the corresponding interface offer exclusively in parameters. It permits one-to-many component compositions.
- A Java native interface (JNI) variant of the port. It maps calls to the corresponding Java implementation and the other way round. This variant is relevant solely if the implementation language of the component is not Java.
- A file-based port offering a serialisation of outgoing calls to a file. It also can parse an input file for incoming calls or for the attributes marked with out.

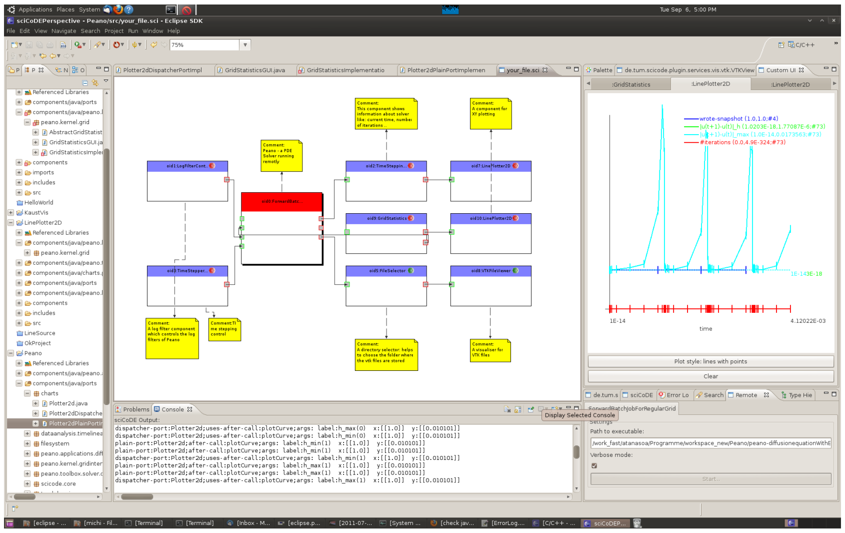
A standard workflow to develop a component reads as follows: First, we write a SIDL file specifying the component's interface and collaboration partners. Second, we make the SIDL compiler generate the corresponding class templates, port implementations, and makefiles. For this, we inform the compiler about the target. Target here comprises both the implementation language of the final component (Java or C++), whether the component runs remotely as stand-alone application or locally within the workbench, and whether a stand-alone application shall communicate due to files or via JNI. An extension of the compiler with new targets is straightforward, as the compiler is a plain Java application realised with SableCC [7]. Third, we write the component implementation, i.e. fill the generated class templates with code. All the underlying steps are embedded into the Eclipse plugin.

3. Eclipse plugin. ASCoDT is an Eclipse plugin and available due to Eclipse's update mechanism [8]. Its user interface (Figure 3.1) comprises four major parts per ASCoDT project: A palette, a component workbench, editors for SIDL files as well as the components' implementation in the target language, i.e. Java and C++, and user-defined or component-specific respectively windows or views. Besides the graphical elements, a component management system, i.e. a very lightweight component middleware, runs in the background.

The palette enlists the components available in the current project, i.e. all classes defined in the project's local SIDL files as well as imported components. From this list, the user can instantiate components via a drag-and-drop mechanism. The palette view offers a context menu where the user can import components cased into an external archive file. Archives later are subject of discussion. The user also can select a class from the project's palette and export it into a stand-alone component archive anywhere in the file system. This way, it is possible to share components among different projects.

The workbench shows a directed graph representing the application's assembly, i.e. the instantiated components, their ports, and which ports are connected to each other. Each instantiated component is represented by a rectangular box. Its uses ports are small rectangles on the right edge of the component's box, its provides ports are small rectangles on the left side. Instances can be moved around and scaled. Due to a tool bar, the user can inform the workbench that he would like to connect ports, select any uses port, and connect it to a provides port graphically. The result is a link from the uses port to the provides port and an instantiation of a well-suited port implementation. The choice is up to ASCoDT which port implementation to choose. The workbench by default chooses a dispatcher port. If no dispatcher port exists because of outgoing parameters in the interface definitions involved, the default port is a plain port. If the two components are written in different target languages but none of them is a stand-along application, it uses a JNI-based port implementation. If the target component is running as stand-alone application, ASCoDT instantiates file-based port implementations. The workbench's visual layout resembles object diagrams of the unified modelling language: Each component instance has an object name (an identifier), which is separated from the component's type (class name) by a colon. Each link between two ports has the name declared with `as` in the SIDL file, and, thus, illustrates the fact that the connection is realised as object attribute. The opportunity to give components an identifier enables the user to choose meaningful instance names explaining the component's purpose. Besides the object names, the workbench also allows the user to insert notes into the workbench that document the component assembly.

The SIDL editor is a standard text editor with syntax highlighting and auto completion. Two component usage models are available in ASCoDT. On the one hand, the user can import components from an archive, use them in the workbench, assemble applications visually, and conduct experiments with them. In this case, the underlying SIDL files are not manipulated by the user. On the other hand, the user can realise and tailor components, i.e. create and edit SIDL files and write and modify implementations. In this case, the underlying SIDL files are edited, and the plugin links the editor to the SIDL compiler with an autosave/autobuild semantics, i.e. whenever the user changes the SIDL file and saves these changes, all glue code and ports are automatically regenerated, and component instances on the workbench are reloaded. For the latter usage model, it is possible to open editors for the component implementations in Java or C++ which are also tied to the autobuild mechanism. Both variants of components, local ones of the project and imported archives, are available in the palette, can be used for the assembly in the workbench, and can be the foundation of scientific experiments.


Comment: A log filter component which controls the log filters of Peano
Comment:Time stepping control
Comment: A directory selector: helps to choose the folder where the vtk files are stored
Comment: A visualiser for VTK files
Problems
Console
Display Selected Console
Plot style: lines with points
Clear

```
sciCoDE Output:
dispatcher-port:Plotter2d;uses-after-call:plotCurve;args: label:h_max(0)  x:[[1.0]]  y:[[0.010101]]
dispatcher-port:Plotter2d;uses-after-call:plotCurve;args: label:h_min(0)  x:[[1.0]]  y:[[0.010101]]
plain-port:Plotter2d;after-call:plotCurve;args: label:h_min(1)  x:[[1.0]]  y:[[0.010101]]
plain-port:Plotter2d;after-call:plotCurve;args: label:h_min(1)  x:[[1.0]]  y:[[0.010101]]
dispatcher-port:Plotter2d;uses-after-call:plotCurve;args: label:h_min(1)  x:[[1.0]]  y:[[0.010101]]
plain-port:Plotter2d;after-call:plotCurve;args: label:h_max(1)  x:[[1.0]]  y:[[0.010101]]
plain-port:Plotter2d;after-call:plotCurve;args: label:h_max(1)  x:[[1.0]]  y:[[0.010101]]
dispatcher-port:Plotter2d;uses-after-call:plotCurve;args: label:h_max(1)  x:[[1.0]]  y:[[0.010101]]
```

FIG. 3.1. *A typical ASCoDT workbench: Palette (left), workbench (middle), and user-defined views (right). A SIDL and a C++ editor are hidden behind the workbench.*

3.1. Architecture and component GUIs. ASCoDT implements a model-view-controller pattern [23] where the workbench, the SIDL editor, and the palette are views on one component repository holding the Java component instances, ports running locally, and representatives for remote components, components written in C++, and all remote ports. Due to the representative objects in the component repository, the workbench derives automatically which port implementations have to be used if new component connections are established. The representatives also keep track of component changes due to source code modifications and trigger makefiles or a Java rebuild automatically. As all component interactions are method invocations, the static repository is all component middleware required.

Besides elementary operations on the repository such as factory methods to create new components and establish port connections, ASCoDT's kernel offers a number of standard toolboxes or extension points, i.e. interfaces that might be implemented by a component realisation, to every component implementation to facilitate the realisation of user interfaces.

- There is an extension point enabling components to provide one or several

SWT-based GUIs to the user. The resulting GUI windows are displayed within ASCoDT in a view, i.e. small subwindow, of their own.
- There is an extension point enabling components to create VTK visualisation pipelines due to VTK's Java wrappers [1]. The resulting three-dimensional illustrations are displayed within ASCoDT's workbench in a view, i.e. small subwindow, of their own, and ASCoDT provides basic navigation mechanisms to manipulate the view point in the three dimensional space displayed.
- There is an extension point enabling components to write visualisation algorithms with Java 3D$^{\text{TM}}$. The resulting three-dimensional illustrations are displayed within ASCoDT's workbench in a view, i.e. small subwindow, of their own, and ASCoDT provides basic navigation mechanisms to manipulate the view point in the three dimensional space displayed.
- There is an extension point enabling components to connect to a remote Paraview server [9]. ASCoDT controls this server, it displays a small screenshot of the remote images that is permanently updated, and it provides simple navigation mechanisms such as zoom, rotate, and translate the data to the user.

3.2. Software development cycle. ASCoDT's vision covers all phases of the component development cycle. The state-of-the-art integrated SIDL editor facilitates the interface design of components. The autobuild mechanism generating all glue code in the background hides the routine jobs from the developer. Eclipse's plugins for Java, C++, Python, and FORTRAN allow the developer to write components in the same place where he later on assembles the applications with other components. These plugins can be augmented by domain-specific, i.e. scientific computing-specific language extensions [17], and thus facilitate the development work even futher. The standard interfaces for graphical user elements and visualisation foster permanent scientific experiments and the evaluation and analysis of results in the same place where the source code is developed. The palette finally offers the opportunity to deploy finished components into files of their own.

Exported ASCoDT components are zipped archives comprising the compiled component source code, i.e. class files or executables, its SIDL definition, additional resources required by the component such as icons and configuration files, and all SIDL definitions of provided and used ports. These components can be exchanged between different ASCoDT projects. Bigger projects with teams thus might introduce a marketplace managed by a source control system where the team members offer tested and mature software components to other developers and application experts. Also components from previous projects are integrated into the project due to this component archive mechanism. Whether premanufactured or new components are used and assembled is completely hidden from the user of the workbench.

The component development benefits significantly from the maturity of the Eclipse project. Tools such as source code editors, such as integrated source control management systems, or such as refactoring wizards assist the component developer. In particular the support of tuning and performance analysis tools is promising. Furthermore, the all-in-one-place paradigm fosters agile scientific computing where the user permanently switches from coding activities to experiments, result evaluation, and back [14].

4. Component interaction. The present paper emphasises the interplay of component development and software engineering in scientific computing. ASCoDT is one prototype to illustrate how component engineering and component-based numerical simulations can be embedded into state-of-the-art software development environments. The implementation realises basically a method invocation paradigm where components interact due to port objects. As longs as the component granularity is sufficiently coarse, i.e. the application is broken down into rather big, smart components not tightly coupled and not communicating permanently, such a method invocation scheme is sufficient. More sophisticated approaches coupling objects directly without intermediate ports however do exist and come along without or almost without performance penalties compared to subroutine or virtual C++ function calls [30, 32].

The individual components have single point of contact semantics: If a component is running in parallel (for example due to MPI or OpenMP), ASCoDT nevertheless treats it as a single entity, and all method invocations affect only one process, i.e. rank 0 or the primary thread. Different to CCA realising a single component multiple data (SCMD) model [12, 30] or data redistribution approaches [35], we leave it up to the user to deploy method calls among several MPI ranks or threads. This is a drawback for components of fine granularity, but providing more sophisticated communication schemes is on the one hand out of scope for this work and on the other hand an integration of well-established ideas.

4.1. Auto-parallelism and call semantics. The performance of component architectures is influenced by the decision whether parameters passed to a provides port are copied or passed by reference—the "to keep or not to keep" challenge [36]. For CCA, this discussion is solely relevant for arguments annotated as `in` arguments. Our implementation copies `in` arguments if their cardinality equals one. Otherwise, it works with pointers, i.e. it ranks the do-not-copy policy higher than the idea to map the unidirectional data-flow semantics to the source code. This way, components can, technically, violate the call semantics, but we circumnavigate running out of memory.

Whole operations with data-flow semantics enable the workbench to deploy such entities to threads of their own. ASCoDT protects each component with a boolean semaphore. Function calls comprising exclusively `in` parameters then are deployed to a thread of their own and the calling function returns immediately, i.e. it is a multithreaded non-blocking function call. While this automatic thread parallelism speeds up the computation, its fire-and-forget semantics introduces non-deterministic behaviour. We leave it up to the user to tackle this.

4.2. Component remote deployment. An important aspect of an integrated development environment in scientific computing is the capability to manage and handle distributed system. While components distributed among huge clusters are not considered here (a massively parallel MPI application still is considered to be one component), nevertheless the fact is taken into account that scientists often develop software on their local workstations with their tailored environment and well-chosen set of tools but run these components then on remote clusters.

ASCoDT supports components running remotely due to a file-based communication protocol. If the SIDL compiler is informed to generate component glue code for a remote component, it embeds the component into a stand-alone program. This program is started remotely by the workbench either due to SSH or due to an LoadLeveler [10] connection. All method calls are serialised and deserialised by the corresponding port implementations. Thus, the workbench solely has to take care to copy the corresponding files to and from the destination remote computer, while the generated glue code on the target machine comprises a polling mechanism permanently checking for new incoming port calls.

This realisation is primitive and introduces bottlenecks if the component granularity is small, i.e. many messages are sent forth and back to and from the remote computer. However, it fits well to the security policies and firewalls of many computing centres, it already demonstrates how the handling of remote (super)computers is integrated into the workbench, and it fits to Grid application landscapes where method calls typically have to be serialised into messages to services [31]. Also the fact that a remote component runs on a massively parallel system is hidden from the user. ASCoDT's glue code comprises write to checkpoint file and read from checkpoint file operations. If they are overwritten with an implementation by the component, it is possible for the workbench to shutdown remote components, perform source code updates, trigger a recompile, and restart the application. The same way, it is possible for the workbench to shutdown remote components, transfer the checkpoints to another machine, and to restart the application there.

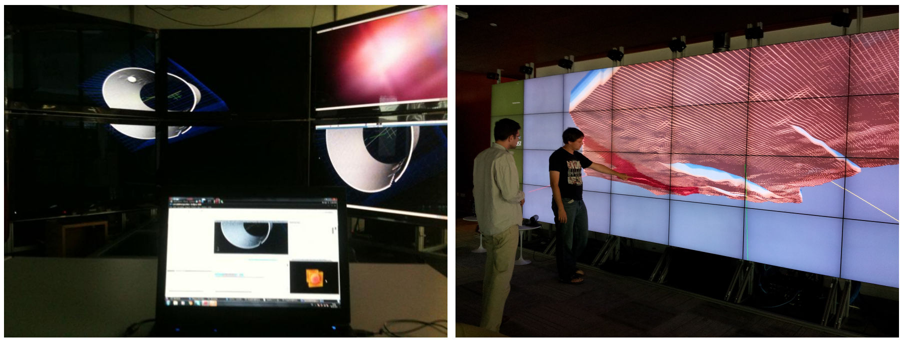


Fig. 4.1. *While the user's laptop/workstation runs sciCoDE in front (left) or not visible here (right), some of ASCoDT's visualisation views are deployed to a remote, CAVE-like system [41] or a powerwall in the back running the visualisation server.*

4.3. Remote visualisation. Permanent visual feedback is important for computational steering, for experiment control, and for debugging and analysis of new source code. ASCoDT on the one hand provides a toolkit for Java components to visualise data due to the VTK library or Java 3D$^{\mathrm{TM}}$ within the workbench. Multiple visualisation views are provided, i.e. the ASCoDT user is able to compare and merge different visualisations. Also, visualisation modifications immediately affect the visualisation window next to the editor. On the other hand, visualisation often itself requires lots of computing resources. In this case, it is reasonable to deploy visualisation tasks to dedicated nodes.

ASCoDT adopts Paraview's [9] remote rendering and server concept. If a Paraview server is running remotely (in a Cave Automatic Virtual Environment, a CAVE-like environment, or on a powerwall e.g.), ASCoDT can connect to and interact with this server. Components running inside the workbench than may command the server due to this connection, while a screenshot of the remote rendering is displayed locally in a downsampled resolution (Figure 4.1). This approach is similar to the ParaViewWeb project (cf. [9] and references therein).

With components scripting a remote Paraview server, it is for example possible for the user to work on his own laptop while sitting inside a CAVE visualising experimental data from the simulation controlled by the laptop running ASCoDT. Also, the local workstation running the component workbench neither has to have outstanding rendering capabilities, nor does it have to run Paraview—an important fact for example for non-UNIX systems. Such a remote visualisation not only is impressing and reveals simulation details probably not observable on the small scale, it also offers the opportunity to make multiple persons work both on the same simulation data and the same simulation software in parallel—the scientific computing counterpart of pair programming, and an important ingredient to make the simulation development workflow agile [14].

5. Case study. The present work explicitly addresses soft aspects of component development in scientific computing. It is hence important to study component-based engineering beyond technical aspects. The impact on the developers, i.e. the "program-and-feel", has to be studied, as well as the impact on typical developer tasks and software designs. A comprehensive empirical study of either issue is beyond the scope of this paper. In the present paper, we however present first results from a survey about the impact of the development workbench, and we examined one prototypical computational fluid dynamics. Here, we translated an existing code into a component-based design before we extended it into an interactive computational steering application.

5.1. A survey. We conducted our survey among Ph.D. developers from several scientific computing groups at our university. All of them have a strong numerics and computer science background, and they work on research codes with prototype character and new algorithms, i.e. maintainance activities, industry-relevant simulation work, and research where simulation is rather a tool to obtain insight than the subject of study are neglected. Altogether, 76 hours of development activities were tracked. The survey focuses on 15 developers working in teams of around 2.88 persons. Twelve of them use most of the time C++, only one prefers FORTRAN to C/C++, and two favour domain-specific languages such as Matlab and computer algebra systems. None of them used Java before we introduced ASCoDT, while all had experience with the non-Java plugins for Eclipse. Among the most popular helper tools were bash, gnuplot, make, and Python. On average, 2.64 applications were used simultaneously, i.e. were held open on the active desktop and invoked frequently throughout a programming session.

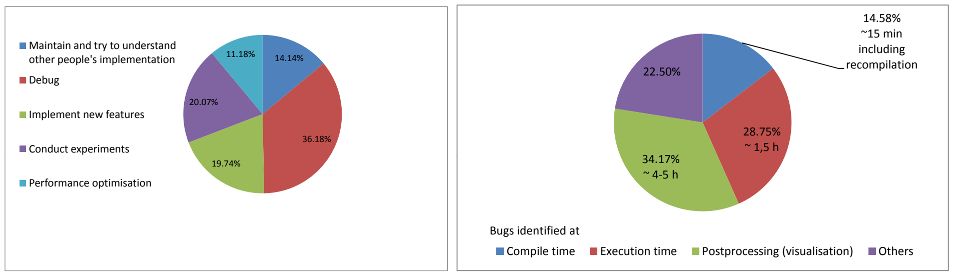


FIG. 5.1. *Left: development time spent on different activities; right: bugs identified throughout the development, and time needed to fix them.*

The participants spend more than one third of their time on debugging, and another third of their time on numerical experiments—even though these are development rather than insight-through-computing activities—and performance tuning (Figure 5.1). The latter activity first of all covers the execution of the software on a target machine in combination with analysis tools as well as the evaluation of the output of these tools. The implication is twofold. On the one hand, the figures are in perfect accordance with other studies: "[...] HPC productivity is also greatly influenced by several other less tangible factors, the most significant among them being the preparatory stages of application execution. Thus another strategy to improve HPC productivity is to reduce the effort involved in scientific application development. Research shows that in terms of productivity, the build process, i.e., compilation, linking, installation, deployment, and staging, may consume up to 30% of the development effort" ([38] citing [22]). While the authors conclude that conditioning techniques are essential to improve the productivity, also continuous deployment and compilation as meta-techniques are important. On the other hand, the figures reveal that debugging remains expensive. Techniques such as automatic testing reducing debugging efforts significantly seem not to be mainstream in scientific computing yet. Thus, there is a need for sophisticated debugging tools. They are integrated into ASCoDT's workbench due to the Eclipse foundation.

All test candidates follow an iterative programming style switching permanently from editing, over compilation, execution, to analysis and back. This basic cycle is followed around 3.34 times per hour. Per basic cycle, the developers switch from one application to another 3.08 times, i.e. the development tools are not integrated seamlessly. Only few developers however report that the seamless integration yields a time saving: The average developer saves only 8.75 percent of his time due to the integration. Yet, two report to have saved up to 20 percent. While individuals benefit from ASCoDT's seamless integration, the impact on the majority of developers is not severe. However, this could be different if the build process itself is more expensive, i.e. if bigger projects are studied [28].

The candidates commit source code changes 0.37 times per hour to the version control system. Each commit comprises 4.08 files if a new feature is committed. If the developers fix a bug, 2.58 files are involved. These numbers include solely source code files, and they are in accordance with commercial reports where less than 10 percent of commits affect only one file [34]. The developers report that having the experimental data and the experiment setup at hand for each commit saves up to 37.92 percent of the debugging time—it is easier to reconstruct why a commit had happened if the motivating experimental data is available. ASCoDT's holistic approach facilitates the debugging, as components, settings, and workflows are integrated and available to the source control system.

More than one third of all bugs are identified not before the simulation results are postprocessed (Figure 5.1), and a fix of such a bug induces four up to five hours of work. As the postprocessing is the last activity in the basic development cycle, these figures reveal one reason for debugging being that expensive. The test candidates report time savings around 13 percent if the effect of source code changes would immediately reveal in a visualisation. Consequently, the smooth integration and the visualisation in-place can speed-up development and debugging, but the savings are limited and do not eliminate the debugging hurdle completely. Hence, the integrated visualisation is a pro for the development process, but its impact on the realisation of new applications such as steering is more severe (Table 5.1).

Finally, the test candidates report that their software is typically installed on 2.45 machines simultaneously by them. With team sizes around 2.75 members, half of a dozen installations is instantaneously affected by each source code modification. The developers however switch from one machine to another place only 1.008 times per programming session, i.e. most developers log into one machine per work unit and

Table 5.1
The integration of different tools reduces the time spent on development and debugging.

	Activity	Savings	Total Savings
Integration of development tool chain	Development & Experiments	8.75%–20%	3.2%–7.2%
Integration of source control for all ressources	Debugging	37%	13 %
Integration of visualisation	Debugging	13%	4 %

conduct the whole work process there. ASCoDT's feature to integrate remote applications is important for Grid-like applications and deployment of computationally intense jobs. It is not important for the development workflow.

These results help to understand the perceived impact of ASCoDT on the development process. The prototype slightly reduces the time-to-numerical-experiment and it particularly reduces the time until a bug is identified due to its seamless integration and seamless visualisation or postprocessing, respectively, of results as well as the holistic approach merging source code with experimental settings, workflows, and results. Both impacts also stem from glue code all generated in the background. In combination with other tools such as performance analysis programs, we expect the workbench to yield also higher productivity for non-debugging tasks. The integrated approach adopts an existing IDE to advanced high performance computing and shows that the tools are mature enough to be integrated into each other to increase the productivity [26, 39]. The impact of seamless visualisation and remote deployment on the programming activities is negligibly though opens the door into Grid-based computing and computational steering.

With respect to recommendations from successful big CSE projects [37], several important pros of ASCoDT's ideas have to be highlighted. First, the merger of a development environment with an experiment workbench enables application specialists directly to work with the codes and workflows while developers still work on it. Such an early integration of application specialists that are the real customers of new CSE software is considered to be an important ingredient to successful CSE endeavours . Second, several successful CSE projects report that they were built on successful prototypes. ASCoDT provides a perfect environment to build such prototypes, combine them, and interchange them prior to a production code development. Third, object-oriented languages are considered to be a hurdle rather than a stimulus for many CSE projects, as their learning curve is steep and their performance—if not used without sufficient experience—poor. At the same time, many software components benefit from object-oriented techniques in terms of maintainability and time-to-delivery due to their power in expressiveness [25]. ASCoDT's multi-language concept allows, while not presented here, to combine FORTRAN and C codes with object-oriented components and thus leaves it open to the developer to select a proper implementation language and to combine the best of two worlds.

Despite a promising perception, these results have to be interpreted carefully. From the low ratio of FORTRAN (and in general non-object-oriented) codes compared to other scientific computing surveys [37], it becomes clear that the studied programming activities concern new code rather than legacy code. The combination of new code and researchers on the graduate level makes us expect that this community is more open to new ideas and improvements of their software development workflow than other communities [26]. We have to assume that a different test group would be more sceptical on any workflow or tool change. Also, the small teams imply that these projects are perfectly suited to agile development—in particular due to the fact that it is research codes, i.e. the requirements of the code change permanently and there is usually no such thing as a fixed implementation road map. With traditional workflows or bigger teams, the savings due to continuous integration might be smaller. On the long term, elaborate productivity metrics and productivity studies are necessary to draw a more detailed picture about the strengths and threats of the approach with respect to team sizes, application field, and software performance [39].

5.2. Computational fluid dynamics. Starting point of our computational fluid dynamics field study is a monolithic C++ solver merging two different physical models. It computes an instationary flow on complicated geometries and two or dimensions due to a Lattice-Boltzmann formulation on adaptive Cartesian grids. The solver runs in parallel, and it supports checkpointing. With the flow field at hand, it virtually inserts a particle into the flow and computes this particle's position due to Faxéns theorems of motion without changing the original flow field. The particle update operations are seamlessly integrated into the Lattice-Boltzmann time stepping. This code is extended in four ways throughout this case study. First, the existing Faxén implementation is complemented by a second implementation working derivative-free, as the computation of flow derivatives for some adaptivity patterns proved to be difficult. Second, there are two supercomputer budgets available for this problem. It has to be possible to transfer the simulation to the supercomputer of choice—typically the one with less workload. Third, multiple particles starting from different initial positions are to be simulated simultaneously. Finally, the particle trajectories are to be visualised on-the-fly. Due to these modifications, we were able to make an existing code base [18] capable to tackle problems from [16] on a previously impossible time scale [24].

As a preparatory step, we extract both the particle tracking and the time stepping loop from the original Lattice-Boltzmann code into two ASCoDT components (Figure 5.2). The communication pattern then reads as follows: First, the time stepping component informs the fluid solver to perform one time step. Second, particle tracking passes the particle's position on to the fluid solver. Third, the fluid solver accepts this particle position and returns the flow field around the (virtual) particle to the particle tracking component. Finally, the particle tracking component updates the particle's position. While the exact details of the data exchange are discussed in [13], it is obvious that this refactoring into components on a very coarse level does not introduce any performance penalty, as the time stepping of the fluid solver is the only computationally expensive step, and as the steps two and four can run in parallel to this time stepping. This holds even though the fact that the fluid solver is the only code running in parallel on a remote supercomputer. The other entities are local ASCoDT components.

For the second Faxén implementation, we create a common interface for the particle tracking and realise two different classes implementing this interface. Throughout this process, we also rewrite the original C++ Faxén simulation with Java. Next, we extend the time stepping component and decide before each time whether to continue the simulation. If we do not continue, we ask the Lattice-Boltzmann code to checkpoint and close the session. Afterwards, the checkpoint files are copied to another supercomputer, and we restart the simulation component there. With the particle tracking and the simulation code being two different components, it is straightforward to instantiate the particle tracking component multiple times to keep track of multiple particles simultaneously. Finally, these particle tracking components forward the current particle position to a VTK-based visualisation. As the particle tracking is refactored out from the original code, it is easy to extend this component and enable the user to interactively modify the virtual particles' positions and to add and remove particles.

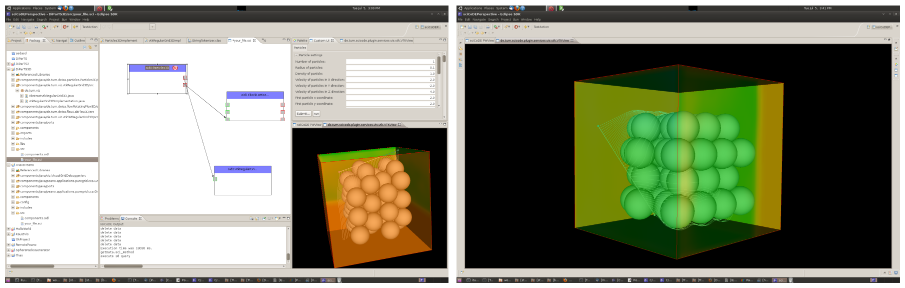


FIG. 5.2. *The monolithic application is initially split into three components. It serves as basis to track the trajectories of several particles in a flow field. On the right, the particles' initial positions are ordered along one line similar to the classical stream line visualisation.*

The experience of this case study reads as follows: Once the application is split up, an extension of the application is simpler, as the net lines of code for the Faxén simulation were reduced from 11665 C++ lines of code to 4994 in Java, while the latter figure already comprises the visualisation. The distribution of the component among the local workstation and the supercomputer as well as the migration of the fluid solver components from one supercomputer to the other are straightforward, too. Tricky is the checkpointing implementation and the parallelisation. However, both are due ot the component developer and cannot be covered by an environment. Our file-based communication, the splitting into separate components, and the distribution among the supercomputer, the local workstation, and even a visualisation server do not induce any performance penalty for this example. If a component however is decomposed more aggressively, or if the data exchange memory footprint is bigger, or if the data exchange is bidirectional and cannot be hidden behind the computation, performance would suffer.

6. Conclusion and outlook. With the increasing computing power also the software complexity grows. Component architectures and component-based designs are an important step towards big, heterogeneous, interactive, distributed scientific computing applications, and they are one ingredient to manage the ever-increasing complexity of today's simulation software. To enter mainstream, component approaches have to master multiple technical challenges, in particular the developers' concerns about the performance overhead. Otherwise, they will not be accepted by the majority of computational scientists. In the present paper, we however emphasise that the acceptance threshold is not only a technical one. It is also important to improve and simplify the software development and the computational experiment process, i.e. the improve the CSE developer's productivity. For this, we propose to integrate both the component development environment, the component management system, and the experiment workbench into one single piece of software. This software shall also offer well-established tools such as source control, testing, and tuning facilities. ASCoDT is a first prototype for such an integrated approach. It makes the development of complex software systems easier.

Beyond that, the integration has an impact on the meaning of computational steering. Steering traditionally circumscribes (real-time) visualisation in combination with an interactive algorithm and input data modifications. Our integration implies a shift from an experiment-centric towards a code-centric understanding of steering. No longer is solely the final artifact and data subject of steering, we also propose the interactive and seamless modification of source code parts and the algorithms. ASCoDT prototypically switches from a waterfall process to a holistic, agile starting point realising big picture thinking that covers all phases of scientific software development and experiment. This adds an additional flavour to the traditional computational steering challenges.

Our piece of software is based upon a dramatically simplified variant of the common component architecture and it covers only a very small subset of the facilities standard CCA-related tools such as Babel [20] offer today. This simplification brings along pros and cons. It makes the handling of the component idea simpler for the user, and allows us to study new ideas rapidly. It lacks features that are relevant for scientific computing. Future work thus comprises a cautious adaption of the CCA standard and integration of other tools. In particular bindings to C, FORTRAN, Python, other scripting languages, and domain-specific language extensions such as [17] are necessary. Furthermore, a support of complex data types is important. In this case, an integration of Babel and Babel-related work, in particular with respect of the type system [27], might be the only reasonable strategy. However, we do not consider the graphical user interface as prototyping and development layer that is later replaced by a script as soon as components are deployed to a supercomputer [12]. We consider ASCoDT to remain the central steering and interaction point for the application expert and supercomputing specialist, as, finally, the adaption of Grid and Cloud techniques in combination with remote deployment of components is promising [31, 33].

If the integration into the software development cycle is smooth, performance arguments finally again enter the discussion on component architectures. The straight next evolution step for ASCoDT with respect to this issue hence is two-fold. On the one hand, we replace the file-based data exchange pattern with a tighter coupling based upon direct remote method invocation or an abstraction of the file idea due to remote memory access [40]. This avoids the file overhead. On the other hand, we replace our one-to-many and single-point-of-contact communication scheme with a many-to-many approach. Here, several MPI processes distributed among a cluster of computing nodes communicate due to ASCoDT's ports directly with many MPI processes running on a different or the same cluster. For this, IO forwarding and array striding are essential techniques to be studied, and it carefully has to be analysed whether data distribution should be the responsibility of the component developer due to a single component multiple data paradigm or whether it does make sense to integrate the data redistribution into SIDL to preserve the language interoperability [35].

Acknowledgements. This publication is partially based on work supported by Award No. UK-c0020, made by the King Abdullah University of Science and Technology (KAUST), and on work supported by the Competence Network for Scientific High Performance Computing in Bavaria (KONWIHR II) through the project CompSteer. Visualisation resources for the present work have been provided by the FRAVE visualisation lab at Technische Universität München [41] and the KAUST Visualisation

Lab. Computing resources for the present work have been provided by the Gauss Centre for Supercomputing/Leibniz Supercomputing Centre under grant pr63no, and by the KAUST Supercomputing Laboratory.

The idea to develop ASCoDT arose throughout a research stay at the Scientific Computing and Imaging Institute (SCI) in Utah made possible due to the International Graduate School of Science and Engineering (IGSSE) at Technische Universität München. The support of the IGSSE is gratefully acknowledged. Thanks also are due to Mihaela Mihai and Denis Jarema for their work as student research assistants.

ASCoDT is based upon Eclipse hosted by the Eclipse Eclipse Foundation [5]. It uses Eclipse's Graphical Editing Framework (GEF) and can be linked with the J2SSH libraries [11] and the Java wrappers of VTK [1]. The underlying compiler frontends are realised with SableCC [7].

REFERENCES

[1] VTK—Visualization Toolkit. http://www.vtk.org.

[2] SCIRun—a Scientific Computing Problem Solving Environment, Scientific Computing and Imaging Institute (SCI). http://www.scirun.org.

[3] VisTrails—a scientific workflow management system that provides support for data exploration and visualization, Scientific Computing and Imaging Institute (SCI). http://www.vistrails.org.

[4] CCA—The Common Component Architecture Forum. http://www.cca-forum.org.

[5] Eclipse—software development environment. http://www.eclipse.org.

[6] Ccaffeine—a CCA component framework for parallel computing. http://www.cca-forum.org/ccafe.

[7] SableCC—an object-oriented compiler framework. http://www.sablecc.org.

[8] sciCoDE—a scientific Code Development Environment. http://www5.in.tum.de/scicode.

[9] Paraview—open source scientific visualisation. http://www.paraview.org.

[10] IBM LoadLeveler—parallel job scheduling system. http://www.ibm.com/systems/software/loadleveler.

[11] j2ssh—independent, extensible implementation of the SSH2 protocol. http://j2ssh.sourceforge.net.

[12] B. A. Allan, R. C. Armstrong, A. P. Wolfe, J. Ray, D. E. Bernholdt, and J. A. Kohl. The CCA core specification in a distributed memory SPMD framework. *Concurrency and Computation: Practice and Experience*, 14(5):323–345, 2002.

[13] A. Atanasov and T. Weinzierl. Query-driven multiscale data postprocessing in computational fluid dynamics. In M. Sato, S. Matsuoka, G. D. van Albada, J. Dongarra, and P. M.A. Sloot, editors, *Proceedings of the International Conference on Computational Science, ICCS 2011*, volume 4 of *Procedia Computer Science*, pages 332–341, 2011.

[14] K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[15] M. R. Benioff and E. D. Lazowska. *PITAC—Report to the President. Computational Science: Ensuring America's Competitiveness.* President's Information Technology Advisory Committee, 2005.

[16] M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, and T. Weinzierl. Numerical simulation of particle transport in a drift ratchet. *SIAM Journal of Scientific Computing*, 30(6):2777–2798, October 2008.

[17] H.-J. Bungartz, W. Eckhardt, T. Weinzierl, and C. Zenger. A precompiler to reduce the memory footprint of multiscale pde solvers in C++. *Future Generation Computer Systems*, 26(1):175–182, January 2010.

[18] H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids. *Computational Mechanics*, 46(1):103–114, June 2010. published online.

[19] S. P. Callahan, J. Freire, E. Santos, C. E. Scheidegger, C. T. Silva, and H. T. Vo. Vistrails: visualization meets data management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, SIGMOD '06, pages 745–747, New York, NY, USA, 2006. ACM.

[20] T. Dahlgren, T. Epperly, G. Kumfert, and J. Leek. Babel user's guide. Technical Report babel-0.99.0 edition, CASC, Lawrence Livermore National Laboratory, 2006.

[21] J. Dongarra, P. Beckman, P. Aerts, F. Cappello, T. Lippert, S. Matsuoka, P. Messina, T. Moore, R. Stevens, A. Trefethen, and M. Valero. The international exascale software project: a call to cooperative action by the global high-performance community. *Int. J. High Perform. Comput. Appl.*, 23:309–322, 2009.

[22] P.F. Dubois, G.K. Kumfert, and T.G.W. Epperly. Why johnny can't build. *Computing in Science and Engineering*, 5(5):83–88, 2003.

[23] E. Gamma, R. Helm, R. E. Johnson, and J. Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, 1st edition, 1994.

[24] D. Jarema, P. Neumann, and T. Weinzierl. A multiscale approach for particle transport simulation in low reynolds number flows. Technical report, Technische Univeristät München, 2012. 12th Copper Mountain Conference on Iterative Methods, Student paper competition, to be submitted.

[25] K. Kennedy, C. Koelbel, and R. Schreiber. Defining and measuring the productivity of programming languages. *International Journal of High Performance Computing Applications*, 18(4):441–448, 2004.

[26] J. Kepner. Hpc productivity: An overarching view. *International Journal of High Performance Applications*, pages 393–397, 2004.

[27] G. Kumfert, D.E. Bernholdt, T.G.W. Epperly, J.A. Kohl, L.C. McInnes, S.G. Parker, and J. Ray. How the common component architecture advances computational science. *J. Phys.: Conf. Ser.*, 46:479–493, 2006.

[28] G. Kumfert and T.G.W. Epperly. Software in the doe: The hidden overhead of "the build". Technical report, Lawrence Livermore Nat'l Lab., 2002.

[29] J. W. Larson, B. Norris, E. T. Ong, D. E. Bernholdt, J. B. Drake, W. R. Elwasif, M. W. Ham, C. E. Rasmussen, G. Kumfert, D. S. Katz, S. Zhou, C. Deluca, and N. S. Collins. Components, the common component architecture, and the climate/weather/ocean community. In *In 84th American Meteorological Society Annual Meeting*. American Meteorological Society, 2004.

[30] S. Lefantzi, J. Ray, and H. N. Najm. Using the common component architecture to design high performance scientific simulation codes. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, IPDPS '03, pages 52.1–, Washington, DC, USA, 2003. IEEE Computer Society.

[31] M. Malawski, J. Meizner, M. Bubak, and P. Gepner. Component approach to computational applications on clouds. *Procedia CS*, 4:432–441, 2011.

[32] L. C. McInnes, B. A. Allan, R. Armstrong, S. J. Benson, D. E. Bernholdt, T. L. Dahlgren, L. Freitag Diachin, M. Krishnan, J. A. Kohl, J. W. Larson, S. Lefantzi, B. Norris, S. G. Parker, J. Ray, and S. Zhou. Parallel pde-based simulations using the common component architecture. In *Numerical Solution of PDEs on Parallel Computers*, volume 51 of *Lecture Notes in Computational Science and Engineering (LNCSE*, pages 327–384. Springer-Verlag, 2006.

[33] I. L. Muntean. *Efficient Distributed Numerical Simulation on the Grid*. Dissertation, Institut für Informatik, TU München, München, October 2008.

[34] G. C. Murphy, M. Kersten, M. P. Robillard, and D. Ćubranić. The emergent structure of development tasks. In *ECOOP–European Conference on Object-Oriented Programming*, pages 33–48, 2005.

[35] E. T. Ong, J. W. Larson, B. Norris, R. L. Jacob, M. Tobis, and M. Steder. Multilingual Interfaces for Parallel Coupling in Multiphysics and Multiscale Systems. In Y. Shi, G. D. van Albada, J. Dongarra, and P. M.A. Sloot, editors, *Computational Science - ICCS 2007, 7th International Conference Beijing, China, May 27-30, 2007, Proceedings, Part I*, volume 4487 of *Lecture Notes in Computer Science*, pages 931–938. Springer, 2007.

[36] S.G. Parker and C.R. Johnson. SCIRun: A scientific programming environment for computational steering. In *Supercomputing '95*. IEEE Press, 1995.

[37] D. E. Post and R. P. Kendall. Software project management and quality engineering practices for complex, coupled multiphysics, massively parallel computational simulations: Lessons learned from asci. *Int. J. High Perform. Comput. Appl.*, 18:399–416, November 2004.

[38] M. Sńska, J. Sński, and V. Sunderam. Enhancing productivity in high performance computing through systematic conditioning. In *Proceedings of the 7th international conference on Parallel processing and applied mathematics*, PPAM'07, pages 341–350, Berlin, Heidelberg, 2008. Springer-Verlag.

[39] M. Snir and D. A. Bader. A framework for measuring supercomputer productivity. *International Journal of High Performance Computing Applications*, 18(4):417–432, 2004.

[40] J. Soumagne and J. Biddiscombe. Computational steering and parallel online monitoring using RMA through the HDF5 DSM Virtual File Driver. *Procedia CS*, 4:479–488, 2011.

[41] M. Tönnis, A. Benzina, and G. Klinker. Utilizing Consumer 3D TV Hardware for a Flexibly Reconfigurable Visualization System. Tech report, Technische Universität München, 2011.

[42] C. Upson, Jr. T. Faulhaber, D. Kamins, D. H. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The application visualization system: A computational environment for scientific visualization. *IEEE Comput. Graph. Appl.*, 9:30–42, July 1989.

[43] C. Williams, J. Rasure, and C. Hansen. The state of the art of visual languages for visualization. In *Proceedings of the 3rd conference on Visualization '92*, VIS '92, pages 202–209, Los Alamitos, CA, USA, 1992. IEEE Computer Society Press.

[44] K. Zhang, K. Damevski, and S. G. Parker. Scirun2: A CCA Framework for High Performance Computing. In *In Proceedings of the 9th International Workshop on High-Level Parallel Programming Models and Supportive Environments (HIPS*, pages 72–79. IEEE Press, 2004.