

Hybrid Scheduling in the DeVIDE Dataflow Visualisation Environment

Charl P. Botha and Frits H. Post
Data Visualisation^{*†}
Delft University of Technology

Abstract

Dataflow application builders such as AVS, OpenDX and MeVisLab are popular and effective tools for the rapid prototyping of visualisation algorithms. They enable researchers to build applications by graphically connecting functional modules together to form a network. A usually hidden yet important aspect of these tools is the scheduling of network execution: Most of these environments can be classified as employing event-driven or demand-driven scheduling. The scheduling strategy has important implications for the component developer. In this paper, we present our recently open-sourced dataflow application builder, called DeVIDE, for the rapid prototyping of medical visualisation and image processing techniques. Apart from the unique interaction possibilities and ease of integration that it offers, DeVIDE differentiates itself from similar environments by implementing a hybrid scheduling approach that adaptively applies demand- and event-driven scheduling to a single network. In this way, ease of component development and execution efficiency can be combined.

1 Introduction

A software platform that enables the rapid prototyping of new algorithms is an important component in the research and development of medical visualisation and image processing techniques. Applications where functional networks can be constructed graphically, enabling visual programming, are a popular and effective solution. AVS [UFK⁺89] and DX [AT95] are early examples of this type of application, which we will henceforth refer to as data-flow application builders. MeVisLab is a more recent example that focuses on medical visualisation and image processing.

An important aspect of data-flow application builders is the scheduling of the functional modules in the constructed network. The constructed network topology implies certain data-dependencies and hence a specific execution sequence. If the output of module A is connected to the input of module B, module A has to be executed, its output data has to be passed to module B, and then module B has to be executed.

There are two major scheduling models: event-driven and demand-driven. In the former, an explicit scheduling is performed where the complete network is analysed and modules

^{*}<http://visualisation.tudelft.nl/>

[†]This work was partly supported by the Virtual Laboratory for e-Science project (<http://www.vl-e.nl/>).

are centrally executed in the correct sequence. In the latter model, when data is requested at the output of any given module, that module requests data from its input module, which requests data from its input module and so forth.

Demand-driven execution makes efficient streamed execution relatively straight-forward. If a sub-block of data is requested at module B's output, module B need only request the data from module A that it needs to generate that specific sub-block. In addition, only the parts of the network that need to be executed to generate a specific block of data are executed without the need for an explicit network analysis. However, the component developer has to go to more trouble to implement a well-behaved module. During event-driven execution, an explicit analysis of the network can be performed in order to determine which parts need to be executed, but generally speaking, complete datasets are passed from module to module. This is less efficient, but has the important advantage that module implementations are far simpler, thus easing the burden of the component developer. Existing data-flow application builders choose either one of these scheduling models. MeVisLab uses demand-driven execution, AVS uses event-driven.

In this paper we present a newly open-sourced data-flow application builder called DeV-IDE, or the Delft Visualisation and Image processing Development Environment. DeV-IDE is a turn-key cross-platform application for medical visualisation and image processing that differentiates itself from similar applications in the following ways:

- Integrating new functionality is easier than with most other platforms of this kind.
- Pervasive interaction possibilities are offered: any object or variable in the system can be interacted with and modified by the user at run-time, via the graphical user interface and also via program code that can be inserted at any point.
- We have combined event-driven and demand-driven scheduling in a hybrid scheduling approach that adaptively offers the efficiency and scalability of demand-driven execution and the programming simplicity of event-driven execution.

The main technical contribution of this paper is the hybrid scheduling approach and its implementation. The freely available DeV-IDE platform is an ancillary contribution and will also be extensively discussed. This paper serves as the companion publication to the open-source release of the software.

The rest of this paper is organised as follows: In section 2 we briefly discuss related work. In section 3, we document DeV-IDE's functionality and architecture. We then detail the scheduling approaches available in the software in section 4. Conclusions and future work are discussed in section 5.

2 Related Work

A number of dataflow application builders, i.e. applications that enable the graphical construction of functional networks, are available for prototyping visualisation and or image processing algorithms. In this section, we discuss a small number of representative examples.

As mentioned in the introduction, AVS [UFK⁺89] and the open-source OpenDX [AT95] are some of the earliest examples. Both of these require the component developer to develop new modules in C and carry quite some programming overhead. Both employ event-driven scheduling.

SCIRun¹ [Par99] is a *Problem Solving Environment* that focuses on providing functionality for computational steering in a visual programming setting. SCIRun has an interesting scheduling approach where all modules are requested to execute at the same time in parallel threads. Because modules wait until they receive input data before executing, network execution occurs in the correct order. In its simplest form, this is equivalent to event-driven execution where independent branches are executed in parallel. SCIRun and its modules are implemented in C++ with some scripting support in Tcl/Tk.

MeVisLab² is a fully functional medically oriented example that has proven itself in a large number of successful research projects. MeVisLab makes use of demand-driven scheduling. Due to this and its page-based central memory management, it is able to process large datasets efficiently. However, developing new modules can be quite complex. This environment has good scripting integration in the form of JavaScript and Python, but much of this functionality is available only in the commercial version.

VisTrails [BCC⁺05, SVK⁺07] is at the forefront of research with and on dataflow for visualisation prototyping. Also based on Python, this environment makes use of functional network metadata to keep track of all experimentation with visualisation pipelines and to perform high-level reasoning with these pipelines. For example, visualisation networks can be compared and visualisations on new data can be created by analogy with existing visualisations. In contrast to this, DeVIDE focuses on more low-level interaction with the visualisation internals as will be explained in section 3.2. In addition, DeVIDE specialises on medical applications, and hence the modules operate at a slightly higher semantic level, as will be explained in the next section. VisTrails also employs an event-driven execution model.

3 System Description

We start the system description with a concise definition and overview of the DeVIDE system. We then describe the software from the user and component developer perspectives. The user interacts with the system mainly through the visual programming and black-box interfaces and these are detailed first. The pervasive interaction features are an important aspect of the visual programming interface and are discussed in a separate subsection. Subsequently, we document component development issues. The system description is concluded with a high-level architecture overview.

DeVIDE is a software environment for the rapid prototyping of medical visualisation and image processing algorithms. Its primary front-end is a graphical boxes-and-lines visual programming interface that can be used to rapidly construct and experiment with functional networks.

¹<http://software.sci.utah.edu/scirun.html>

²<http://mevislab.de/>

Python [vR01] has been used as the main implementation language in a mixed-language setup: Most processor-intensive processing has been implemented in more traditional low-level languages such as C++ or Fortran whereas all high-level functionality has been implemented in Python. The choice of a high-level dynamically typed language as main implementation language is one of the most important factors underlying DeVIDE's unique characteristics.

The software encapsulates a large spectrum of functionality, namely the complete VTK [SML99], ITK [ISNC03], numpy [Oli07] and matplotlib [Hun07] libraries, as well as various other smaller libraries. New external functionality can be easily added as module kits, an abstraction for modular extensions described in section 3.5. In general, the functional modules offered by DeVIDE operate at a higher level than the filters in the libraries that it encapsulates. For example, whereas the VTK abstraction requires at least four different classes to display a triangle mesh, the slice3dVWR in DeVIDE can take almost any data as input and will automatically construct the relevant visualisation pipeline behind the scenes.

DeVIDE is turn-key in the sense that the binaries are completely self-contained. They can be installed on an otherwise clean system via a single command and the user can immediately start working.

3.1 Visual Programming Interface

As will be seen in section 3.5, the DeVIDE architecture allows for interchangeable interfaces, both graphical and non-graphical. The default graphical interface is a boxes-and-lines visual programming interface.

Figure 1 shows a screenshot of the visual programming interface with a simple functional network. A small set of DICOM files have been dropped on the canvas, causing the software to create and configure automatically a dicomRDR module. The output of the reader module has been connected to a slice3dVWR, allowing the user to start visualising the data in about three mouse clicks.

To the left of the canvas there are three sub-windows: module search, module categories and the current module list. Each module can belong to multiple categories and in that sense it is more of a tagging system to facilitate finding the correct modules for any given theme. One or more categories can be selected, in which case only the modules that belong to those categories are shown. By typing in the module search box, only modules containing the search terms in their names, keywords or documentation will be shown in the module list. In this way, applicable modules can usually be found quite quickly.

At the bottom left, the documentation of the currently selected module is shown. At the bottom right behind the 3-D visualisation is a sub-window with log messages.

A module can be placed on the canvas by dragging and dropping it from the module list, or by pressing the `Enter` key if the module has been selected by a search. Modules are connected by dragging from the output port of one module and dropping on the input module of another module. In general, modules have a view or configuration window which can be accessed by double-clicking on the module.

By default, DeVIDE makes use of explicit network execution, regardless of whether the

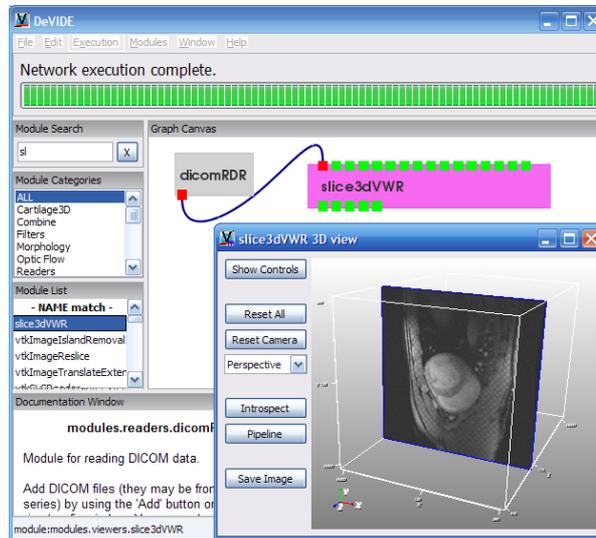


Figure 1: Screenshot of DeVIDE. A collection of DICOM files have been dropped on the canvas, at which point a dicomRDR is automatically created. Connecting the output of the dicomRDR module to a slice3dVWR is all that is needed to start visualising the data.

network is being executed in event-driven or hybrid mode. After a network has been constructed and configured, the user has to request execution explicitly. During network execution, the progress bar and message at the top of the interface as well as the log window at the bottom right give detailed feedback on network execution. If an error occurs at any point, either during the execution of a module or the transfer of data from one module to the next, the user is informed with a detailed error message, often containing information down to the specific line of code causing the error.

3.2 Pervasive Interaction

We start this section with an illustrative example. The superQuadric module generates both an implicit function and a triangle mesh with a user-configurable super-quadric function. Connecting a superQuadric to a slice3dVWR results in the visualisation shown on the right of figure 2. On the left the default configuration window of the superQuadric module is shown. Any of the parameters can be modified and the network can be instantaneously updated to see the changes in the visualisation. By experimenting with more complex setups in a similar fashion, visualisation approaches can be explored until a suitable solution is found.

In DeVIDE, the user is also allowed to interact with the internals of any functionality at run-time. In this case, a program code dialogue can be initiated with the superQuadric module via its introspection interface. Figure 3 shows this introspection interface. In the bottom



Figure 2: On the left the default configuration window of the superQuadric module is shown and on the right the resultant triangle mesh as visualised by the slice3dVWR.

subwindow, the user is experimenting with module internals, assisted by code-completion functionality. In the top subwindow, code has been entered that animates the superquadric surface visualisation over a range of one of its parameters. Besides the code-completion, the interface also shows method documentation wherever available.

Similar introspection interfaces are available in the main interface and also in a refined form in the CodeRunner module, briefly discussed in section 3.4. This functionality enables the advanced user to interact with and modify any code object or parameter and also to script complex program behaviour, all at run-time. The combination of this type of interaction, available at every point in the application, combined with more traditional dataflow application builder user interface functionality as well as instant visual feedback on all changes, makes possible an efficient form of prototyping where the user can rapidly converge on suitable solutions to visualisation problems.

3.3 Black-box interface

As mentioned in section 3.1 and 3.5, the main DeVIDE interface can be easily replaced. A complementary interface to the visual programming interface is the black-box interface. This is a GUI-less mode where DeVIDE can be run on compute nodes without any kind of display, for example on a cluster or a grid. This is useful for performing parameter sweeps, where the same operation has to be performed a large number of times to determine the best combination of parameters for a particular problem, or for applying the same operation to a large number of datasets in parallel.

In this mode, DeVIDE can be configured to load dataflow networks that have been saved from an interactive session, and apply these networks to arbitrary datasets. This means that a complete dataflow network can function as a black-box component inside another dataflow application builder, or a coordination framework, such as Nimrod/G [BAG00]. This also enables DeVIDE to function in both the initial development and in the later tuning and evaluation stages of the medical image analysis research pipeline [OSBB07].

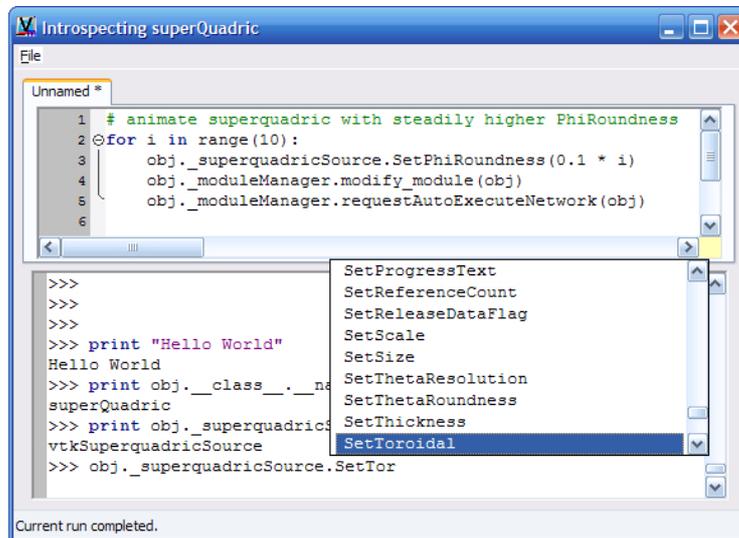


Figure 3: Introspection interface to the superQuadric module, invoked by selecting “Introspect the Module (self)” option from the module view. In the bottom window, the module is being queried via a line-by-line dialogue. In the top window, code has been entered that animates the superquadric visualisation over a range of one of its parameters. The code can be executed by selecting “Run current edit” from the “File” menu.

3.4 Component development

Component development refers to the extension of DeVIDE with new functionality. This can occur in the form of module kits, modules, the CodeRunner module or snippets. A common theme of DeVIDE component development is that care has been taken to alleviate the task of the developer. In all cases, development overhead is kept to a minimum.

3.4.1 Module Kits

The module kit is an abstraction for encapsulating and integrating external libraries such as VTK, ITK or the graphical user interface library wxPython³. The module kit programmer has to supply an initialisation function and an optional refresh function that can be used to reload a changed library. In general, this code is minimal compared to the functionality in the library itself, and is mostly responsible for making the library functions available to other DeVIDE components. During startup, all module kits are checked for loading errors and are removed from the list of available kits if necessary. The software can also easily be configured to start up with any subset of the available module kits, thus customising DeVIDE for different use scenarios and freeing up extra memory.

³<http://www.wxpython.org/>

3.4.2 Modules

Modules are the algorithmic components used to build up functional networks. Each module represents a different visualisation or image processing algorithm.

The canonical way to create a new module is to implement a Python class with the methods shown in the *Module* class in figure 4. The first two methods are called when input data pointers are transferred or when output data pointers are queried. The `execute` methods are called respectively during traditional event-driven execution and during hybrid execution if this is the last module in a streamable subset (see section 4.2, this method is optional). The last two methods, also optional, are responsible for module serialisation. `get_config()` should return a data structure so that when this data structure is used as a parameter to `set_config()`, the module is returned to the state it was in when `get_config()` was called. By making use of these calls and storing network topology, DeVIDE can save a complete functional network for later use.

The `view()` should activate the module view or configuration window. The component developer has almost complete freedom in the definition of this method. Complete new graphical interfaces can be programmatically created, such as that of the slice3VWR, pre-defined module user interfaces can be used, or the programmer can request that DeVIDE create a configuration window automatically by specifying which parameters should be adjustable. DeVIDE takes care of all sanity checking and processing of user input in this case.

Any functionality that can be invoked from Python and that supports a dataflow architecture can be integrated as a module. For example, we have created modules that invoke shell commands behind the scenes, taking care of all parameter settings, file handling and so forth. This is useful for hiding the complexity of existing commandline-based simulation and processing software from the user. Modules could for example also query web services or websites.

Besides the canonical way to create new modules, a number of base classes and pre-defined methods are defined that ease component development. For example, VTK objects can be wrapped in five lines of Python code. These modules have automatically created user interfaces that gives access to all VTK class attributes and support configuration handling so that they can be saved and restored. As a proof of concept, a few hundred of the VTK filters were automatically wrapped using this base class.

The highly dynamic nature of Python enables an effective model of prototyping. When a new module is created and added to the necessary configuration file, it is added to the list of available modules at run-time and can be immediately combined with an existing network. If changes are made to the module source code, it can be reloaded in place, without exiting the application, and the effects of the change can be examined. Together with the pervasive interaction discussed in section 3.2, this enables a rapid code-run-view-code cycle during which new functionality can be created and fine-tuned for a specific problem.

3.4.3 CodeRunner

The CodeRunner is a special "blank" module that allows the user to create new module functionality with low programming overhead. The module can be integrated with a func-

tional network and will be scheduled with the rest of the network. Its interface is similar to the introspection interface shown in figure 3, except that it has three tabs in the top subwindow: scratch, setup and execute. Code entered into the setup tab is executed once for every time that it is modified by the user. This can be used for setting up VTK or ITK pipelines for example. Code entered into the execute tab is executed whenever the module is scheduled. Module inputs and outputs are easily accessible. The scratch tab is for experimentation and will not be executed during network scheduling.

By using CodeRunner modules, new ideas can be quickly packaged with minimal overhead in modular form and integrated with existing complex networks.

3.4.4 Snippets

Snippets represent another extension possibility. These are small scripts that can be loaded into any of the introspection interfaces introduced in section 3.2 and perform functions such as creating animations of evolving level sets, or changing the colour map of a slice of DICOM data.

3.5 Architecture overview

Figure 4 shows the high-level architecture of DeVIDE. The core acts as the communications hub of the other architectural components. The interface class should support basic logging, error handling and progress feedback methods. It is easily interchangeable. At the moment, DeVIDE has a visual programming and black-box interfaces, described in sections 3.1 and 3.3 respectively, but one could envision creating for example a web server interface. The scheduler class encapsulates all functionality explained in section 4. The network manager is responsible for loading and saving networks.

The module manager encapsulates quite a large fraction of DeVIDE functionality as it is concerned with the central activity of module housekeeping. It catalogues available modules, instantiates, destroys and queries them for documentation, all at the request of other components. It is also responsible for the lower level logic behind the saving and restoring of complete networks. Finally, the module manager has similar responsibilities with regard to module kits. It takes care of initialising, error-checking and making available the kits to other components of DeVIDE.

The module class is shown in more detail at the bottom right of figure 4. Its methods and working are described in more detail in section 3.4.2.

4 Scheduling

In this section, the scheduling functionality in DeVIDE is documented. First the default event-driven scheduling is explained, followed by the hybrid scheduling approach.

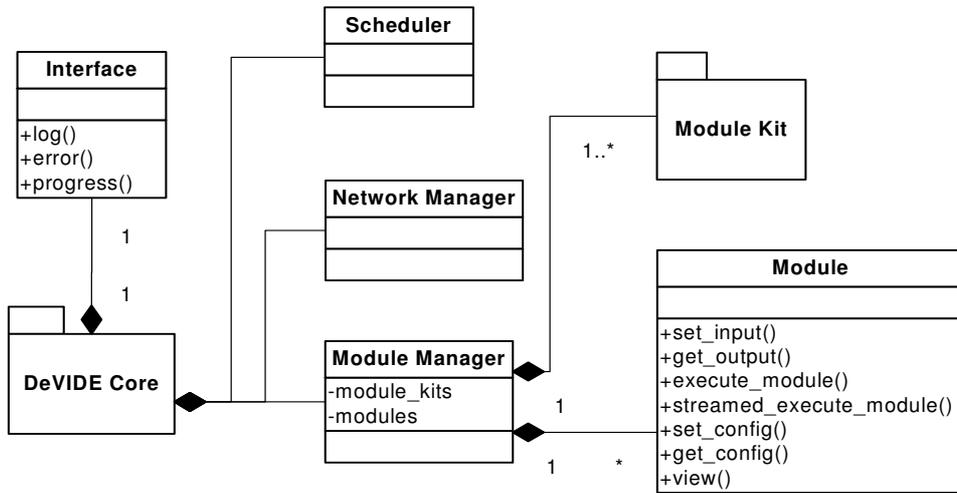


Figure 4: Diagram of the high-level DeVIDE architecture. Due to its important role, more detail on the module class has been included.

4.1 Default scheduling

By default, an event-driven scheduling is applied to networks when execution is requested. Before execution, the network is topologically sorted. The result of this operation, shown in algorithm 1, is a single list of modules that is ordered such that no module is dependent on any other module that comes after it. The topological sort is only applicable to directed acyclic graphs. DeVIDE does allow certain cycles, as they are useful for user interaction to be fed back into a functional network. These cycles are removed from the network before sorting and recorded for later use. If any remaining cycles are detected, sorting and execution are not performed.

Algorithm 1 Topological sort

```

V ← all modules
S ← {}
while V ≠ {} do
    Vf ← final_vertices(V)
    S ← S + Vf
    V ← V - Vf
end while
reverse(S)
S is the topological ordering of the original V
  
```

The topologically sorted list is traversed. At every iteration, pointers to datasets are trans-

ferred from module outputs supplying the current module, and then the module is executed via its `execute_module()` method. In this setup, each module may assume that complete datasets are available at its inputs and it has to produce complete datasets at its outputs. This greatly simplifies module development.

Special care is taken to work around libraries such as VTK and ITK where an update query on output data is able to initiate an execution of its producing filter. The data is disconnected from its producing filter before passing its pointer to the next module.

A straight-forward form of execution caching is employed. Modules are only executed if they have been modified since the previous execution, for example by parameter changes, or if data pointers have been transferred to their inputs. Data pointers are only transferred if the modules that produce them have been executed since the previous transfer. In this way, only the necessary parts of a network have to be re-executed after modifications. The scheduler keeps track of all data transfers, module modification and module execution times with three sets of timestamps.

4.2 Hybrid Scheduling

The idea behind hybrid scheduling is that the largest streamable subsets of a network are extracted and separately executed in streaming mode. The rest of the network is executed in the traditional way as explained in the previous section. Although the idea is rather straight-forward, its implementation is interesting and implications significant.

We have created algorithm 2 to find the largest subsets of streamable modules. A module is considered streamable if it offers a `streaming_execute_module()` method. The algorithm iterates through all streamable modules in topological order. If it finds a module that is not part of a streamable subset yet, it creates a new subset. It then uses a depth-first graph search to traverse the rest of the connected network, adding all streamable modules to the streamable subset. The search is terminated at any module where that module has one or more non-streamable consumers. All terminating modules are recorded.

When a network execution is requested, the network is traversed and executed as for the traditional event-driven execution. However, if any module that is iterated over belongs to a streamable subset, data pointers are transferred from its producer modules without the disconnection work-around. Importantly, the module is not executed, unless it is one of the terminating modules of a streamable subset, in which case its `streaming_execute_module()` method is called.

Separate sets of timestamp datastructures are maintained for execution and data transfer caching, similar to those mentioned in the previous section. During hybrid scheduling, timestamps for event-driven execution are invalidated, and vice versa, so that caching is also invalidated for streamable subsets when the user toggles scheduling modes.

Because only the terminating modules in streamable subsets are executed during hybrid scheduling, this is akin to demand-driven execution. In general, streaming execute methods make use of the streaming functionality of the underlying libraries. This allows one to combine adaptively the simplicity of event-driven scheduling, even with libraries that are natively demand-driven, with the efficiency of demand-driven scheduling.

Algorithm 2 Streamable subset finder

```
1:  $S \leftarrow$  topological sort of all modules  $V$ 
2:  $S_s \leftarrow$  all streamable modules in  $S$ , in order
3:  $V_{ss} \leftarrow \{\}$ 
4:  $M_{ss} \leftarrow \{\}$ 
5: for each  $m$  in  $S_s$  do
6:   if not  $m$  in  $V_{ss}$  then  $\{m$  is not in a streamable subset yet $\}$ 
7:     create streamable subset  $M$ 
8:     add  $m$  to  $M$ 
9:     if any consumer of  $m$  is non-streamable or  $m$  has no consumers then
10:      set  $m$  as terminating module
11:     else
12:      add all consumers of  $m$  to  $M$ 
13:      for each consumer of  $m$  do
14:         $m \leftarrow$  consumer
15:        goto line 9
16:      end for
17:     end if
18:     add  $M$  to  $M_{ss}$ 
19:     add elements of  $M$  to  $V_{ss}$ 
20:   end if
21: end for
22:  $M_{ss}$  is a list of streamable subsets of all modules  $V$ 
23:  $V_{ss}$  is a list of all modules in streamable subsets
```

5 Conclusions and Future Work

In this paper we have presented DeVIDE, a dataflow application builder for medical image processing and visualisation. Besides serving as the companion publication to the open source release of DeVIDE, the main technical contribution of this paper is the hybrid scheduling approach that has been implemented.

DeVIDE differentiates itself from similar platforms in the following ways:

Hybrid scheduling The scheduling approach in DeVIDE combines aspects of event-driven and demand-driven scheduling. We summarise this in the following paragraphs.

Pervasive interaction The software makes it possible to interact with the system at runtime even down to the code level, and to get immediate feedback on changes.

Ease of integration We have focused on making it as easy as possible for developers to integrate new components. By employing a high-level dynamically typed interpreted language, by supplying a number of different ways in which code can be added and through the pervasive interaction possibilities, we believe that we have simplified the process of adding new functionality.

Networks are usually scheduled via straight-forward event-driven execution. This greatly simplifies module implementation. However, in cases where some modules are capable of more complex demand-driven execution, the hybrid scheduling functionality is able to analyse such a network and adaptively execute it, applying demand-driven scheduling where possible and event-driven where it is not. The scheduling adapts to every new network and combination of event-driven and demand-driven modules. This is transparent to the user, who benefits from the simplicity of event-driven and the efficiency of demand-driven scheduling.

The software has been used for the segmentation of pathological shoulders from CT data for use in a pre-operative planning solution [KBV⁺06], analysis and biomechanical modelling of the pelvic floor muscles [vJ06], parts of the reconstruction of the visible orbit dataset [vZBW⁺06], running on the grid via Nimrod/G [MOB⁺07], retrobulbar fat analysis [BdGS⁺07] and various other studies that are still in progress. Due to its simplicity and interaction possibilities, it is also a useful educational tool.

DeVIDE has several limitations, first of which is an implication of the fact that it runs as a single process that allows the user access to all internals. By causing one of the underlying libraries to generate a program fault, such as a null pointer dereference, the whole application is halted. Although we would like to alleviate this problem, it is currently seen as an unavoidable side-effect of the flexibility of the software.

In future work, we are planning to integrate simple parallel processing capabilities via Parallel Python⁴, a robust multi-process based solution. As a first step we are planning to extend the scheduler to decompose functional networks into independent subsets. We are also planning to create a new interface whereby DeVIDE can be started up with a module view or configuration window as the primary and only interface. This will enable module developers to deploy simple graphical front-ends to their users, thereby using DeVIDE as a run-time environment.

References

- [AT95] Greg Abram and Lloyd Treinish. An extended data-flow architecture for data analysis and visualization. In *Proc. IEEE Visualization '95*, page 263. IEEE Computer Society, 1995.
- [BAG00] Rajkumar Buyya, David Abramson, and Jonathan Giddy. Nimrod/g: An architecture for a resource management and scheduling system in a global computational grid. In *Proc. High Performance Computing in the Asia-Pacific Region*, pages 283–289, 2000.
- [BCC⁺05] Louis Bavoil, Steven P. Callahan, Patricia J. Crossno, Juliana Freire, Carlos E. Scheidegger, Claudio T. Silva, and Huy T. Vo. Vistrails: Enabling interactive multiple-view visualizations. In *Proc. IEEE Visualization*, pages 135–142, 2005.
- [BdGS⁺07] Charl P. Botha, Thijs de Graaf, Sander Schutte, Ronald Root, Piotr Wielopolski, Frans C.T. van der Helm, Huibert J. Simonsz, and Frits H. Post. *Visu-*

⁴<http://www.parallelpython.com>

alization in *Medicine and Life Sciences*, chapter MRI-based visualisation of orbital fat deformation during eye motion, pages 221–236. Mathematics and Visualization. Springer Verlag, 2007.

- [Hun07] John D. Hunter. Matplotlib: A 2D graphics environment. *Computing in Science & Engineering*, 9(3):90–95, May/June 2007.
- [ISNC03] Luis Ibanez, Will Schroeder, Lydia Ng, and Joshua Cates. *The ITK Software Guide*. Kitware Inc., 2003.
- [KBV⁺06] Peter R. Krekel, Charl P. Botha, Edward R. Valstar, Paul W. de Bruin, Piet M. Rozing, and Frits H. Post. Interactive simulation and comparative visualisation of the bone-determined range of motion of the human shoulder. In Thomas Schulze, Graham Horton, Bernhard Preim, and Stefan Schlechtweg, editors, *Proc. Simulation and Visualization*, pages 275–288. SCS Publishing House Erlangen, March 2006. Best Paper Award.
- [MOB⁺07] Ketan C. Maheshwari, Silvia D. Olabarriaga, Charl P. Botha, Jeroen G. Snel, Johan Alkemade, and Adam Belloum. Problem solving environment for medical image analysis. In *CBMS '07: Proceedings of the IEEE International Symposium on Computer Based Medical Systems*, pages 165–170, 2007.
- [Oli07] Travis E. Oliphant. Python for scientific computing. *Computing in Science & Engineering*, 9(3):10–20, May/June 2007.
- [OSBB07] Sílvia D. Olabarriaga, Jeroen G. Snel, Charl P. Botha, and Robert G. Bellman. Integrated support for medical image analysis methods: from development to clinical application. *IEEE Transactions on Information Technology in Biomedicine*, 11(1):47–57, January 2007.
- [Par99] Steven Parker. *The SCIRun problem solving environment and computational steering software system*. PhD thesis, The University of Utah, 1999.
- [SML99] Will Schroeder, Ken Martin, and Bill Lorensen. *The Visualization Toolkit*. Prentice Hall PTR, 2nd edition, 1999.
- [SVK⁺07] Carlos E. Scheidegger, Huy T. Vo, David Koop, Juliana Freire, and Claudio T. Silva. Querying and creating visualizations by analogy. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1560–1567, 2007.
- [UFK⁺89] C. Upson, T Faulhaber, D. Kamins, D. Laidlaw, D. Schlegel, J. Vroom, R. Gurwitz, and A. van Dam. The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics and Applications*, pages 30–42, July 1989.
- [vJ06] Štěpán Janda. *Biomechanics of the pelvic floor musculature*. PhD thesis, Delft University of Technology, 2006.
- [vR01] Guido van Rossum. *Python Reference Manual*. Python Software Foundation, April, 2001.
- [vZBW⁺06] Joris E. van Zwieten, Charl P. Botha, Ben Willekens, Sander Schutte, Frits H. Post, and Huib J. Simonsz. Digitisation and 3D reconstruction of 30 year old microscopic sections of human embryo, foetus and orbit. In A. Campilho and M. Kamel, editors, *Proc. 3rd Intl. Conf. on Image Analysis and Recognition (ICIAR 2006)*, volume LNCS 4142 of *Lecture Notes on Computer Science*, pages 636–647. Springer, 2006.