A FRAMEWORK FOR THE GRAPHICAL SPECIFICATION AND EXECUTION OF COMPLEX SIGNAL PROCESSING APPLICATIONS

A. Sicheneder, A. Bender, E. Fuchs, R. Mandl, B. Sick

University of Passau
Faculty for Mathematics and Computer Science
Innstrasse 33, 94032 Passau (Germany)

ABSTRACT

A framework with a tool-supported high-level specification technique is very important for the development of complex signal processing applications containing software-intensive parts (e.g. hybrid systems in automated production processes) in order to provide safe and reliable systems. In this paper we present the concept of a framework, which is an object-oriented CASE-tool offering a graphical specification ability to model and validate a given application and to control its execution. A variety of people having different programming skills is able to use this visual specification technique effectively. Especially users not being interested in implementation details can specify their application on a high abstraction level by connecting reusable and reliable components (modules representing basic algorithms). As a result, complex signal graphs representing the dataflow between the modules are created. The tool supports this software specification technique by automatic type checking for the connections between modules and by changeable module parameters. On the other hand it is easy for software engineers to integrate additional signal processing algorithms into the framework thus building suitable module libraries without considering a specific high-level application.

1. INTRODUCTION

Today's industrial production processes are distinguished by increasing automation to improve labour's working place on the one hand and enterprise's productivity and, therefore, competition advantages on the other hand. By means of automation high-quality products can be produced low-priced and rapidly, but computer-aided specification of automated, matured and reliable industrial production processes is often difficult and lengthy [4]. The proposed framework is a CASE-tool for the software specification of systems solving complex signal processing applications and for the exectution of the software realization to observe and control technical processes.

Who is developing which kind of signal processing algorithms? *Application engineers* want to focus on their specific signal processing problem and do not want to have a hard time with implementation details, programming language features, complex data structures and so on. This group has a lot of experience in the specific field of applications. Therefore they are able to propose an appropriate solution of the actual problem by analogy to previous solutions with different parameters or small changes or extensions. For the development on a high abstraction level, this kind of user needs a set of parameterized basic components, which may be composed to form a complex algorithm. Since this abstract composition of a specification should be as simple as possible, such a development of an algorithm should be supported with a graphical user interface (GUI). The GUI of the framework provides a graphical high-level language (HLL) programming facility.

Developers of complicated signal processing algorithms or *software engineers* on the other hand are familiar with the specific problems of algorithm and software development on a lower abstraction level. This group of users wants to implement basic components with respect to modularity, numerical stability, timing aspects and so on. Therefore these people need a flexible and easy maintainable object-oriented framework, which allows an integration of high sophisticated basic algorithms without changes of the framework itself. Graphical visualization modules serve as as a kind of HLL-debugger to test and validate new modules.

After implementing and testing specific signal processing applications, it should be possible to use the framework as well for the actual usage, that is for the execution of the complex algorithm f.i. in the monitoring of industrial production processes. This requires not only a tool for the offline analysis of previously recorded data sets but also a tool for online supervision and control. Hence a customizable front-end with hierachically regulated access to critical components for different user groups is needed.

In the following we present a framework which supports the described separation of abstraction levels of software development for signal processing applications. In section 2 the main components are introduced; the underlying data model is described in section 3 followed by the runtime control of the framework in section 4. A comparison with related tools (section 5) and a summary with the further developments conclude the paper.

2. COMPONENTS OF THE FRAMEWORK

Figure 1 shows the main components of the framework: the *development environment* with the graphical user interface and the module libraries, the *runtime system* with different kinds of interfaces and the *runtime control*.

Complex applications are specified by the application engineer in the GUI in a graphical way as described later in this section. For this reason appropriate parameterized modules are selected from the module libraries and connected to software prototypes which can be tested by means of the runtime control. The development environment supports the user with a compatibility check ('type-checking' facility). Special modules provide interfaces to files, database systems, graphical instruments, visualization tools, A/D- or D/A-converters, computer networks etc. The main task of the runtime control is to supervise the execution of complex algorithms.

As mentioned before, the graphical representation of complex

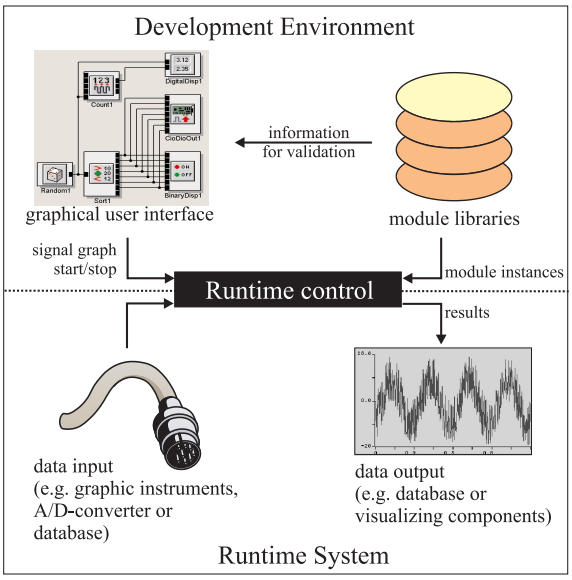

Figure 1: Components of the framework

algorithms abstracts from the module implementation, is clear and easy to understand, and allows an efficient fault detection. Within the GUI, boxes with ports are connected using directed edges (see figure 2). The result is a directed graph which may contain cycles. Nodes without predecessors represent data sources (inputs), nodes without sucessors data outputs and other nodes parameterized basic algorithms. The edges in the graph describe the dataflow between modules. By selecting a module, an instance of the algorithm is created and by wiring two modules a type-check is executed and an instance of a communication medium is created (e.g. a datastream). Modules and communication media are implemented in C^{++}.

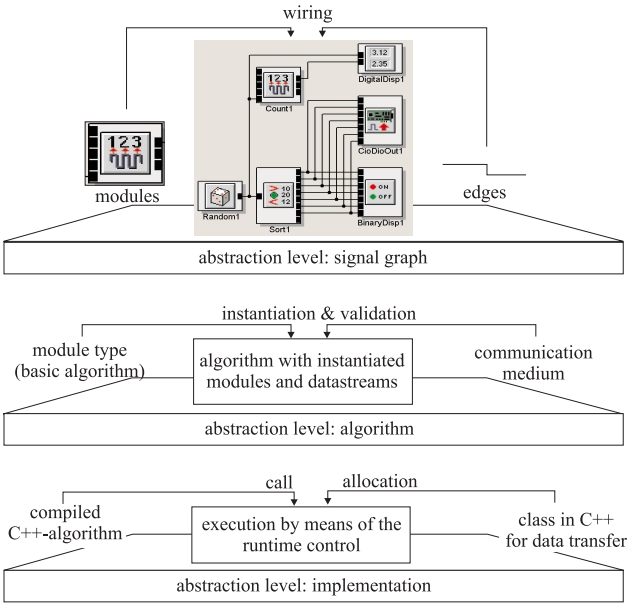

Figure 2: Different abstraction levels with their objects and actions

Some specific features are worth to be mentioned in detail:

- The icon-based notation facilitates the specification of very large programs through a hierarchical decomposition by macros. Macros and an orthographical wiring of the edges between the module ports make the solution clearer and help to avoid design bugs.
- Cycles within the graphical representation allow the specification of adaptive algorithms.
- The runtime system enables multiple asynchronous data sources, the parallel or distributed (over a network) execution of several signal graphs and it is responsible for the detection of the violation of (weak) realtime conditions.

The implementation of new modules is supported by templates which help to collect algorithms into a library in order to use them in the framework. Thus, a software engineer is only responsible for the algorithm's I/O-behaviour without integration aspects. In addition he has not to consider implementation details. Hence the system specification time is significantly accelerated and known solutions can easily be adjusted to similar applications. Expensive error tracing within the whole graph is dramatically reduced due to the modular character of the specification and special display modules within the runtime system. Using graphical specification techniques the documentation of the solution is simplified, because it suffices to describe the (high-level) graphical representation of the algorithm.
The application engineer can use graphical instruments (e.g. controllers and switches) and visualization modules to build a graphical front-end which is understandable and usable by an operator. In order to avoid undesirable usage of the GUI, editing functions can be blocked hierarchically for different user groups.

3. DATA MODEL

Besides the measured or simulated data to be processed by the signal graph, the framework allows another kind of data within a signal graph, namely parameter data as inputs to modules. This feature allows dynamic changes of module parameters, which may have been determined from previous computations. Generally, parameter data are not distinguished from signal data by the framework; however, it is a module's task to ensure the correct interpretation of the data received at a parameter port. All the data flowing within a signal graph are processed blockwise due to the following benefits:

- Processing a block of data consisting of several values reduces the communication effort.
- Sometimes input data are already block-oriented (e.g. signal data from A/D-converters are stored in a buffer and transfered as a block (of data) for further processing).
- If the data to be processed are coming from a sensor recording different objects (e.g. quality control in production processes), the data recorded between two objects may be omitted; therefore the measured data of *one* object may be gathered in *one* block, thus leading to a data reduction on the one hand and to a logical grouping of data on the other.
- Several algorithms (e.g. FFT or approximation algorithms), are not meaningful on single values; therefore this kind of algorithm can easily be served with blocks of data having the amount of values they need for useful processing (e.g. powers of 2 for FFTs).

The problem of finding appropriate block lengths is often a trade-off between the overall execution time of the algorithm (long blocks to reduce the communication overhead) or fast reactions (short blocks).

Blocks are organized hierarchically in two levels (cf. figure 3). A *superblock* represents a set of data which belong together

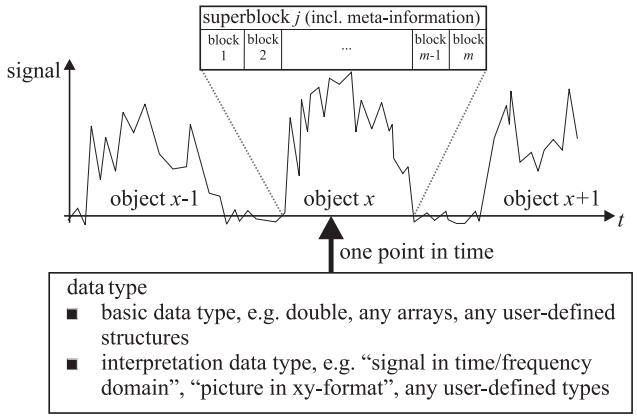

Figure 3: The data model

semantically (e.g. measured data of a single object). A superblock consists of one or more blocks, each containing a sequence of several single values. Additional meta-information is provided for every superblock (f.i. sample frequency, time stamps, unit, name of the signal), which is valid for all blocks within a superblock. This meta-information is used during the execution of the signal graph f.i. to perform conversions of physical units or to process correctly sensor data sampled with different sampling rates.

Figure 3 shows that the supported data types are also organized hierarchically in two levels: the *basic data type* describes the data format of the values in the blocks and the *interpretation data type* describes the context of the data. Basic data types are elementary types (double, int, char etc.), matrices, records, a special type for user-defined strucures, polymorphic types and references between types at different module ports. A type "all" is used to process all possible data types. Interpretation data types inform the user about the context of the signal ("signal in time domain", "picture in xy-format"). Only basic data types are examined in the compatibility check executed whenever two modules are connected in the GUI ('type-checking'). Interpretation data type violations produce warnings without stopping the specification step.

4. RUNTIME CONTROL

The runtime control is the central component of the framework (see figure 1). The execution of complex algorithms (from the module libraries) has to be controlled in a block-oriented and dataflow-driven manner with respect to their priority [5]. A static processing order depending on the graph's structure does not fulfill this task, because control data within the graph often disable subgraphs (e.g. demultiplexers). Another reason is a possible cyclic processing order of subgraphs causing multiple execution of modules in succession. Therefore the processing order has to be dynamic (online computed); it is based on the status of a module called "ready for execution" [6]. A module is ready for execution, if all the required information for the processing of the underlying algorithm is available, i.e. all mandatory input ports are filled with sufficient data. Altogether the framework distinguishes six module states: *instantiated, initialized, ready for execution, not ready for execution, executing* and *terminated*. Figure 4 shows all possible state transitions of the modules which are caused either by the modules or by the runtime control.

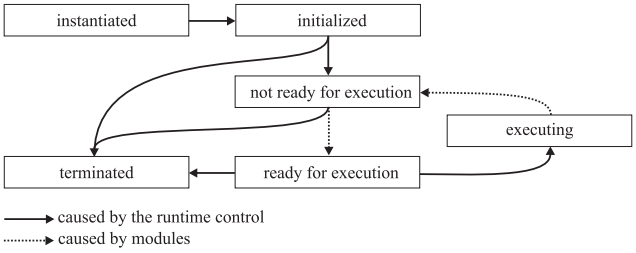

Figure 4: State transitions of the modules

The runtime control algorithm is based on these module states and consists of four phases (cf. figure 5):

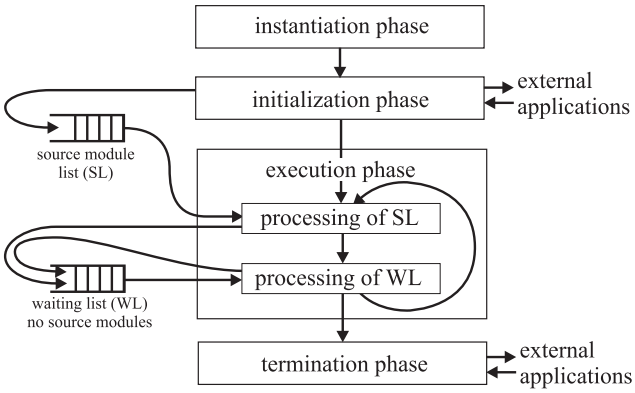

Figure 5: Algorithm of the runtime control

- Instances of modules and the communication media are generated my means of the module libraries in the instantiation phase. In addition, the values specified in the signal graph are assigned to parameters and macros are replaced by subgraphs.
- Within the initialization phase specific initiate actions like memory allocation are performed. Furthermore this phase builds a list containing all source modules with respect to the module's priority. External applications (e.g. a filter specification tool for FIR or IIR filters) can be executed for initialization purposes.
- The execution phase is divided into two subphases: first all the source modules, which are ready for execution, process their corresponding algorithm; after each execution of a module, the subsequent modules are checked whether they are ready for execution, and if so, they are inserted into a waiting list sorted by the modules' priority. A module is removed from the source module list if the module looses the feature "source", i.e. it does not produce data anymore. In the second subphase, the modules in the waiting list are processed according to their priority. The generated output is passed to the subsequent modules and if one of these becomes ready for execution, it is inserted into the waiting

list with respect to its priority. The module, who's algorithm was executed, is removed from the waiting list.
This alternate processing of source module list and waiting list is repeated until the source module list is empty. The processing is also stopped by an explicit request of a module to end the execution, by an external user request or after a runtime error.

- The termination phase, which is executed in any possible case of termination, closes data files, frees memory etc. Again it is possible to call external applications, e.g. tools for offline visualization of processed data.

5. RELATED TOOLS: THE STATE OF THE ART

Graphical specification techniques are well known in the area of control engineering, measurement technology and signal or image processing. Although there are various commercial products dealing with "visual programming", the presented framework offers a lot of advantages. Comparing related tools the following assessment criteria should be used:

- Considering the GUI, it is important how the tools support the specification process. Particularly the hierarchical decomposition facilities, the possibility to use control structures (e.g. loops) and the debugging support should be examined. Type-checking of connections should be carried out during development and not at runtime. Another criterion is a clear representation and an intuitive way of using the tool (look-and-feel).
- Another major criterion is the size of the module libraries and especially the simplicity to insert new modules.
- Looking at the data-types, it has to be examined which types are supported and if new types can be introduced and validated without changing the runtime environment.
- It should be allowed to use multiple (eventually asynronous) data sources with different sampling rates. Block sizes should be adjustable for each connection individually.
- To assess the runtime control the possibility to process cycles within the signal graph has to be investigated. In addition it is necessary to ensure a correct data synchronization of the data which are coming from different data paths (subgraphs). Other criteria are the ability to process several signal graphs in parallel and to ensure the observance of (weak) real-time conditions.

Following [7], some of the most popular software products in measurement technologies like National Instruments' *LabView* [3], Hewlett Packard's *HP Vee*, DIAdem from Gfs mbH [2], and in the area of image processing the tool *Khoros Pro* [1] from Khoral Research Inc. have been investigated.

As a result, it can be stated that particularly different sampling rates and cycles within a signal graph are supported by only a few tools. Other problems for most of these tools are the requirements for real-time execution (actions within the GUI often interfere with the runtime control) and the automatic type-checking during the specification process. The framework presented in this paper is the only tool which fulfills all the mentioned criteria. Therefore it can be used in nearby all applications, even if the solution requires e.g. very specific data-types or a cyclic specification.

6. CONCLUSION AND FUTURE DEVELOPMENTS

The presented framework with its graphical specification ability leads to considerable economic benefits and fulfills the requirements of different user profiles in the following way:

- Signal processing applications can be described, solved and documented in a single working cycle.
- Even very complex applications are understandable and known solutions can easily be adjusted for reuse in new applications.
- Well tested and extendable module libraries are provided; furthermore new libraries can be created and integrated into the framework on demand by software experts.
- The editing of signal graphs can be disabled to preserve the system from unauthorized use f.i. in a controlling application.
- Interfaces to databases or networks can be used to analyse data off-line.
- Several mechanisms help to recover and avoid bugs already in an early stage of the specification phase.

Up to now, the framework has been used in several industrial applications (e.g. online measuring of thin metal foils) and in different research projects at the University of Passau (e.g. tool condition monitoring in turning [8]). A demo version of the tool (working on a PC under Windows95/NT) can be obtained via anonymous ftp from ftp://ftp.uni-passau.de/pub/local/iconnect/files. Our future work deals with the extension of the libraries f.i. by image processing algorithms. A new version working with a real-time operating system is aspired.

7. REFERENCES

[1] D. Frieauff; *Kraftpaket-Bildverarbeitung mit Khoros 2.1;* In: iX- Magazin für professionelle Informationsverarbeitung, May, 1997

[2] Gfs mbH; *Informationen zu DIAdem;* Aachen, 1996

[3] R. Jamal, H. Pichlik; *LabView–Programmiersprache der vierten Generation*, Prentice Hall, 1997

[4] J. Kodosky, J. MacCrisken, G. Rymar; *Visual Programming Using Structured Data Flow;* In: Proceedings of the 1991 IEEE Workshop on Visual Languages, Kobe, 1991

[5] H. Nömmer; *Spezifikation und Implementierung einer Entwicklungsumgebung für Signalverarbeitungsalgorithmen mit Ablaufsteuerung zur datenflußgetriebenen Bearbeitung auf der Basis parametrisierter Module;* diploma thesis; University of Passau, 1997

[6] H. Nömmer, E. Fuchs, B. Sick, R. Mandl; *Entwicklung und Ablauf objekt-orientierter Echtzeitsoftware auf der Basis parametrisierter Algorithmenmodule;* In: Echtzeit97, Wiesbaden, 1997

[7] P.G. Schreier; *Users adopt new technologies, return to familiar suppliers;* In: Personal Engineering, January, 1997

[8] B. Sick; *Monitoring the Wear of Cutting Tools in CNC-Lathes with Artificial Neural Networks;* In: Proceedings of the 1997 International Conference on Acoustics, Speech and Signal Processing, vol. 4, 1997