Computer Science Journal of Moldova, vol.**16**, no.1(46), 2008

Approaches to automated construction of graphical shells for computer algebra systems

Alexander Colesnicov Svetlana Cojocaru
Ludmila Malahova

Abstract

The paper proposes a calculator model of the graphical shell to be used for computer algebra systems. The calculator shell model is described. Then the techniques of semi-automated construction of such shell are discussed. The motivation of the approach based on the domain model is given. We describe also two possible component assembly methods, static and dynamic, and our experience with them. We motivate the selection of dynamic component assembly.

1 Introduction

We suppose the existence of programs executing symbolic computations in computer algebra *(engines)* whose developers need to provide modern graphical shell with their systems. Computer algebra is widely used in many areas, including pure and applied mathematics, theoretical physics, chemistry, engineering, technology, etc. Multitude of solved problems makes investigators to create specialized engines in the cases when use of general purpose systems is inefficient, or the necessary functionality is not implemented even in commercial systems. As a rule, creators of such systems have not enough time, resources, and qualification to develop shells for them. It isn't unusual that rich mathematical ideas implemented in an engine are enveloped in poorly designed interface. Our own experience with the Bergman computer algebra system (CAS) and review of other systems illustrate this [1].

The absence of the user-friendly standard shells makes such systems less popular because of requiring special knowledge and skills, e.g., in programming, to use them.

Another problem of computer algebra engines is multitude of their data formats and the implied difficulty in communication between different engines.

Investigations show that CAS interface developer provides some or all of the following features:

- 2-D presentation of mathematical expressions,
- Editing of mathematical expressions that includes sub-expression manipulation,
- Windows that model sheets of paper and combine texts, formulas, and graphics,
- Processing and presentation of long expressions,
- Simultaneous use of several CAS, which implies the necessity to solve problems of data conversion, configuration management, and communication protocols,
- Interface extensibility providing additions of new menus, new fragments of on-line documentation, etc.,
- Guiding of the user during the whole period of his/her problem solving,
- The system should be self-ex'planatory; its operational mode should be understandable directly from the experience of interaction with the system,
- Control over problem formulation correctness and over information necessary to solve it.

The primary scope of a shell is creation of a comfortable environment for a mathematician or another specialist that uses mathematical apparatus. It would be preferable for these users to input data and to

obtain mathematical results in their natural 2-dimensional form. The linear form of input can be used also as the linear input is faster but it imposes additional conventions to enter powers, indices, fractions, etc., or uses additional characters. It is necessary also to provide possibilities to edit expressions, integrate them with a usual text, and obtain results in a form suitable for publication of an article (e.g., LaTeX) or in Internet (e.g., MathML).

The syntactic check of the entered mathematical expressions and the spelling check of accompanying text would be also desired features.

We see that functions of the graphical shell are almost independent of the engine.

We propose therefore a universal shell implementing the calculator model and constructed from the ready-made components [2]. Moreover, we successfully used several engines at once with such shell. This solves many problems of incompatibility of data formats in different CASs, and solves partially the problem of their interconnection.

Sec. 2 defines and discusses the calculator model of the graphical shell. We describe there the details of its work and its interaction with the CAS engines.

There are two approaches to the automated construction of such shells. Both approaches are based on the component programming (CP) and differ mainly in the technique of component assembly that can be static or dynamic.

At the first approach (static component assembly), we successfully combined the CP and the aspect-oriented programming (AOP). This technique is described in Sec. 3.

It is possible to construct the shell using dynamic component assembly. This second technique was developed over the Eclipse platform. The details are described in Sec. 4.

In the Conclusion (Sec. 5) we compare both approaches, describe their advantages and shortcomings, and motivate our decision to use the second approach.

2 The calculator model of the user interface

A usual numerical calculator works step-by-step: you enter numbers, and select one of possible operations. The calculator executes the operation and shows the result that can be used as an operand for the next operation.

The calculator model of the CAS shell behaves quite similarly. You enter a mathematical object (e.g., an ideal that is presented as a list of polynomials with coefficients from some field), select a possible action (e.g., calculation of the Gröbner basis of the corresponding algebra) and start a CAS engine that executes the operation. The result is a new object (in our example, a new list of polynomials), and it can be used for further calculations (e.g., for reduction of polynomials).

The shell implements the input and output of mathematical object in the form suitable for the user; the engines implement all calculations.

Fig. 1 shows one of variants of our shell that supports two CASs: Bergman[1] and Singular[2].

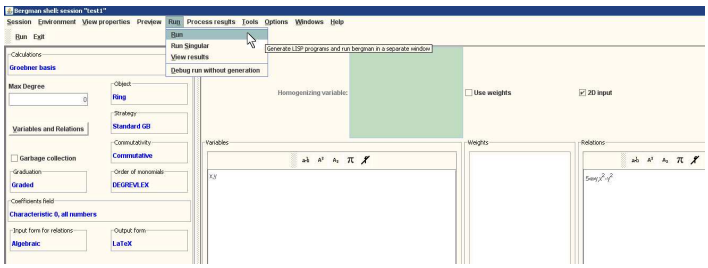

Figure 1. A graphical shell that supports Bergman and Singular

Both these engines support only the console interface. For Bergman, it is the underlying Lisp console. For Singular, it is a console with the Singular programming language. Our shell permits to enter mathematical objects, converts them to the Singular or Bergman input files, and runs the corresponding engine to execute calculations.

[1] http://www.math.su.se/bergman/
[2] http://www.singular.uni-kl.de/

The CAS shell design can have two different starting points: the set of engine operations, or the set of processed mathematical objects. In [5, Sec. 3.3] these approaches are called correspondingly "noun-verb" and "verb-noun" (or "object-action" and "action-object"). V. López-Jaquero and F. Montero [4] refer to these variants as to "domain model" and "task model".

There is no common opinion on applicability of these two approaches. J. Raskin [5] shows several advantages of the first approach over the second one. V. López-Jaquero and F. Montero [4] motivate the advantage of the second approach, but their argumentation applies mainly to the case when the objects are containing in the databases. They note also that "the derivation of the user interface out of a task model adds an additional view to the design process: the user".

In the beginning we built shells for the Bergman CAS originating them from the engine operations ("the task model"). This seemed naturally for Bergman that has only a small set of objects, but a big and growing set of actions. Later we planned to use several CAS engines with the same shell. We wanted to support the usage of an engine that would not be even taken into account at the shell programming, i.e., to make the shell as independent of the engines as possible. The corresponding investigation permitted us to construct the calculator model of the shell. In this case, the shell actions are restricted by the object input, the result output, the object conversions between different representations, the selection of the engine and its operation, and several common tasks like the session support. We see that most tasks are defined by objects and that user tasks are quite restricted. In the meantime [1] the old "verb-noun" model began to hinder the shell development.

Therefore we decided to originate the CAS shell from the mathematical objects we want to proceed ("the domain model"). This approach permitted us to create a shell that is really independent of the used engines, and to provide researchers-mathematicians with a unified shell for several CASs.

Within the domain model, the CAS shell development begins with the listing of used mathematical objects. For each object, we provide

a procedure or procedures of its input that produces its internal XML representation. The entered object is dispalyed on a tab. The current set of tabs with objects, and the current parameter settings can be stored as a *session* and restored later.

The object is displayed on a tab in one of its external representations (e.g., LaTeX). A set of convertors from internal XML representation to different external representations exists for each object. The user selects a representation of an object through a menu. There is also a possibility to store an object in any of its representations in a file.

Except of input procedures and convertors to external representations, each object is associated with actions that can be performed over it by the existing engines. As the user opens object's tab, the associated actions became visible in a menu.

Some actions use more than one object. These additional objects should be entered and visible on other tabs. If there are several combinations of objects for one or several actions, the corresponding request is made to select one of these variants.

So the user selects object(s) and the action. The shell converts objects from their internal XML representation to the engine input files and starts the necessary engine.

After the engine run termination the calculated result is kept usually in a file. After the calculation is finished a shell module converts the result in its internal XML representation and shows it on a new tab. Being a mathematical object, this result can be converted in different external (visual) representations, saved in a file, and used for further calculations.

We see that a calculator CAS shell model supposes the object-independent part (session support, tab manipulations, dynamic menus support, etc.) and the object-dependent part. The object-dependent part contains input modules and convertors. Most convertors are engine-independent but the convertors used to generate input files before the engine start are engine-dependent. There are also engine-dependent convertors that scan output files after the engine finishes its calculations and produce internal XML representation of resulting objects.

Action menus are both object- and engine-dependent. To change these menus dynamically, the shell uses XML descriptions that exist for each type of object, each engine, and each allowed combination of those.

3 Static shell assembly

Our technique of the static shell assembly is described in details in [3]. We combined in it the component and aspect-oriented programming.

Having a set of ready-made modules described above we need three operations to assembly a shell:

1. to generate the *glue code* (the additional code that is necessary to assemble components together);
2. to generate code that tunes adaptable components;
3. to generate variable menus.

Aspect oriented programming (AOP) is a technique to add a new behavior to an existing program without changing its sources and even binaries. It is mostly used to handle cross-cutting concerns like logging or debugging. E.g., we need to add almost the same code in regularly selected places of the program to trace it. AOP concentrates templates of additional code and insertion points in *aspects*. Aspects are compiled separately, and the *code weaving* is performed during the execution of the program.

To apply AOP for the semi-automated assembly of a shell from components, we noted that the glue code is regular and repeating, and that it can be generated from a formal description of the shell. With AOP, we use an unchanged shell template and unchanged components, and generate only aspects containing the glue code or the code to tune adaptable modules. The menu is also generated as an aspect. We have checked this idea by implementing it.

A shell consists of the constant part and the variable part. The constant part contains, in particular, the session management: storing data for each session, their modification, etc. We also found useful a

notion of *environment*, or partially defined session [1]. Each session can be based on an environment where some data are already defined. The environment management is implemented like the session management.

Other features of the constant part of a shell are possibilities to create the list of engines, to start external programs, to check collected data, to show help, etc.

Modules that enter the data and convertors form the variable part of a shell.

During the assembly of a shell its constant part is taken as the base. The developer prepares list of objects and defines how they have to be entered in the shell (by selection from several variants, by marking, by text editing, by 2D formula input, by entering parameters of a mathematical object using a wizard, etc.) Each possible method of the data input is implemented as a customizable component. The necessary modules pass the customization and are glued together with the constant part of the shell. Menus are also generated and included as an aspect.

The whole system consists therefore of a pre-implemented constant part, a set of data input components and convertors, and a shell generator that adapts and assembles all parts together producing CAS shells.

4 Dynamic shell assembly

The shell with dynamic module assembly is based on the open source Eclipse[3] platform.

An Eclipse-based application consists of the Eclipse *platform* and a set of *plugins*. Each plugin is a module that contains in itself its XML description as a resource. The system tunes itself (e.g., adds new menu items) using the XML description of the new module. To add a module, it is enough to copy its JAR archive in the plugin directory and to restart.

The visual part of the Eclipse platform is the Eclipse *workbench*.

[3] http://www.eclipse.org/

The workbench provides a window that contains tabbed *views* (e.g., lists of settings) and *editors*. Such window is called in Eclipse the *perspective* (corresponds to *session* as we defined before). Editors are plugins that edit texts; a usual text editor is already provided with the Eclipse platform. The workbench supports also *projects*; for a project, we can store and restore its current perspective.

The construction and work of the CAS shell based on Eclipse remains the same as described before.

The Eclipse platform will form the constant part of the shell.

Modules to input mathematical object should be implemented as plugins-editors. Convertors should be also implemented as plugins. Eclipse supports the dynamic change of menus at the activation of each editor.

A separate plugins are necessary to conduct engines. It includes engine list support, engine start, consoles, etc., and, especially, the support of correspondence between the engine functions and mathematical objects. This last feature is new for Eclipse.

5 Conclusions

The first approach permits to implement a platform-independent system. We work in Java with Swing graphics. The deployed shell consists of a single executable JAR that contains the compiled classes, resources, and additional libraries. This archive can be executed on any platform with the suitable version of the Java VM. At the second approach we use SWT graphic library from Eclipse that is platform dependent. We are in this case to deploy different archives for different platforms, or to require the user to install Eclipse or, at least, its libraries.

However the Swing graphics was criticized for its visual appearance that does not correspond the platform standards. Eclipse uses the native graphics on each platform that can slightly accelerate the graphical operations and guarantees the native appearance.

At the first approach we implement session support in the exact necessary volume. With Eclipse, we are to use the Eclipse framework

that is more general and may seem more complicated: some base features of Eclipse are superfluous for us.

Any shell expansion (adding a new mathematical object, new engine or new action of the existing engine, etc.) implies recompilation to add new features at the static assembly. Eclipse adds new plugins dynamically.

The factors listed till now balance one another; none of them is decisive. The main advantage of Eclipse is its richness, especially in the current Eclipse 3 "Europa". This version of Eclipse contains more than 900 ready-made plugins. A big part of common GUI functions is already implemented or can be adapted from existing plugins. After the appearance of Eclipse 3 we decided to use this approach in our project. However the general system structure and functions are common for both approaches.

6 Acknowledgements

The work was supported by the INTAS grant Ref. Nr. 05–104–7553 "Interface generating toolkit for symbolic computation systems".

References

[1] S. Cojocaru, L. Malahova, and A. Colesnicov. Interfaces to symbolic computation systems: Reconsidering experience of bergman. *Computer Science Journal of Moldova*, 13(2(28)):232–244, 2005.

[2] S. Cojocaru, L. Malahova, and A. Colesnicov. Providing modern software environments to computer algebra systems. In V.G. Ganzha, E.W. Mayr, and E.V. Vorozhtsov, editors, *Computer Algebra in Scientific Computing. 9th International Workshop, CASC 2006. Chişinău, Moldova, September 11–15, 2006. Proceedings*, number 4194, pages 129–140. Springer-Verlag, 2006.

[3] A. Colesnicov and L. Malahova. Aspect oritented programming and component assembly. *Computer Science Journal of Moldova*, 15(1(43)):38–53, 2007. ISSN 1561–4042.

[4] V. Lopez-Jaquero and F. Montero. Comprehensive task and dialog modelling. In J. Jacko, editor, *Human Computer Interaction, Part I. HCII 2007. Beijing, China, July 22–27, 2007.*, number 4550, pages 1149–1158. Springer-Verlag. ISSN 0302–9743.

[5] J. Raskin. *The Humane Interface. New Direction for Designing Interactive Systems*. Pearson Education, Inc. (Addison Wesley Longman), 2000. ISBN 0–201–37937–6.

A. Colesnicov, S. Cojocaru, L. Malahova,

Received February 19, 2008

Institute of Mathematics and Computer Science,
5 Academiei str.
Chişinău, MD−2028, Moldova.
E–mail: *kae@math.md*, *sveta@math.md*, *mal@math.md*