Template-Based Semi-Automatic Workflow Construction for Gene Expression Data Analysis

Jiří Bělohradský[1], David Monge[2], Filip Železný[1], Matej Holec[1], Carlos García Garino[2]
[1] *Czech Technical University, Prague, Czech Republic*
[2] *Instituto para las Tecnologías de la Información y las Comunicaciones, UNCuyo, Argentina*
{belohji1,zelezny} @fel.cvut.cz {dmonge,cgarcia} @itu.uncu.edu.ar

Abstract

We propose a technique for semi-automatic construction of gene expression data analysis workflows by grammar-like inference based on pre-defined workflow templates. The templates represent routinely used sequences of procedures such as normalization, data transformation, classifier learning, etc. Variations of such workflows (such as different instantiations to specific algorithms) may entail significant variance in the quality of the analysis results and our formalism enables to automatically explore such variations. Adhering to proven templates helps preserve the sanity of explored workflows and prevents the combinatorial explosion encountered by fully automatic workflow planners. Here we propose the basic principles of template-based workflow construction and demonstrate their working in the publicly available tool XGENE.ORG for multi-platform gene expression analysis.

1. Introduction

Numerous software tools such as [11] exist facilitating the analysis of gene expression data. Typically they each follow some fixed computational workflow. The options for varying such workflows by the user are limited and usually manual. While significant improvements in analysis results (such as accuracy of a final predictive model) may be entailed by slight adaptations of a given workflow, such adaptations may not be possible with a conventional tool.

This problem is rectified partially by the currently vivid research [1,5,6] on automatic construction of data analysis workflows. Here, a space of admissible workflows (i.e., those leading from the specified inputs to the requested outputs) is systematically explored, usually relying on planning algorithms known from artificial intelligence. The problem of such approaches is the combinatorial explosion resting in the sheer number of possible admissible workflows, not all of which make intuitive sense to the analyst.

Here we propose a middle-way approach combining the benefits of both of the mentioned extremes. Fixing a specific data domain (in particular the gene expression analysis domain), it turns out that the experience-proven workflows share common patterns, and it would be an overkill to reinvent them from scratch. Therefore we develop an automatic workflow generator based on a system of pre-defined *templates* (directed graphs capturing the standard procedures) and *graph-rewriting* rules enabling meaningful substitutions of non-terminal nodes in the templates. The latter can be rewritten either into terminal nodes (specific algorithms) or into other (sub)templates retrieved from a library. As follows intuitively, edges in the produced graphs represent flow of data.

The grammar is generally non-deterministic, i.e. several rewriting rules are applicable to a given workflow, until it is fully instantiated. Thus we would normally end up with multiple different admissible workflows, each pertaining to a particular sequence of rewriting-rule applications. Whereas this satisfies our goal of exploring workflow variations, a sequential execution of such produced workflows (needed to establish their respective performances) would result in a large volume of repeated computations. This is because the workflows may share large substructures.

To remove this problem, we instead produce a single branched workflow, allowing multiple paths through it, each corresponding to a different version of an admissible analysis sequence. Such a workflow is generated from templates using a deterministic grammar. The latter relies on special rewriting rules that allow graph branching. The produced workflow is also naturally executable in a GRID environment.

We demonstrate the proposed approach within the public set-level [9], multi-platform [8] gene expression analysis software XGENE.ORG, in tasks involving

normalization, feature extraction (lifting features from genes to various kinds of gene sets), predictive model learning with different algorithms, and cross-validation of predictive accuracy. Here, the use-case serves to exemplify the proposed workflow generation principles. The empirical results collected from their application on specific data sets and their statistical analysis are reported in a separate paper [3].

2. Workflow construction

We first describe the formal principles of the workflow construction mechanism and then provide some technical details on its implementation.

2.1 Workflows as Graph Refinements

Formally a workflow $\mathbb{W}$ is generated by a function $\mathbb{t}(\mathbb{T}, \mathbb{A}) = \mathbb{W}$ where $\mathbb{T}$ is a directed graph defining a template and $\mathbb{A}$ is a set of graph rewriting rules. $\mathbb{A}$ and $\mathbb{T}$ are assumed to be defined by the user or simply retrieved from a library. More specifically, $\mathbb{T}$ is a tuple: $(\mathbb{E}, \mathbb{S}, \mathbb{i}, \mathbb{I}, \mathbb{O})$, where $\mathbb{E}$ is a set of template elements (vertices), $\mathbb{S}$ is a set of successions (edges), $\mathbb{i}$ is a function $\mathbb{i} : \mathbb{S} \rightarrow \mathbb{E} \times \mathbb{E}$ which maps successions to pairs of template elements, $\mathbb{I} \subseteq \mathbb{E}$ is a set of input template elements and $\mathbb{O} \subseteq \mathbb{E}$ is a set of output template elements.

We conceptualize the types of vertices and edges as follows. The domain of template elements $\mathbb{D}_\mathbb{E}$ ($\mathbb{E} \subseteq \mathbb{D}_\mathbb{E}$) is categorized through a subset structure:

$$\textit{PluginHolder} \subset \mathbb{D}_\mathbb{E};\ \textit{SubstructureSetProcessor} \subset \textit{SubstructureSet} \subseteq \textit{Substructure} \subset \mathbb{D}_\mathbb{E}.$$
$$\textit{PluginHolder} \cap \textit{Substructure} = \emptyset$$

In rewriting, an element of category *PluginHolder* may be substituted by either a specific algorithm ("plugin") or by an element of category *Substructure*, which corresponds to another defined template. An element of the *SubstructureSet* category is rewritten into a number of parallel branches each corresponding to a different parameterization of a single template. Finally, rewriting an element of category *SubstructureSetProcessor* (which must directly follow a *SubstructureSet* element) results in attaching a substructure or a plugin to each branch obtained by rewriting the immediate parent.

The domain of rewriting rules ("assignments", for short) $\mathbb{D}_\mathbb{A}$ ($\mathbb{A} \subseteq \mathbb{D}_\mathbb{A}$) has a structure of subsets:

$$\textit{DirectPluginAssignment} \subset \mathbb{D}_\mathbb{A};\ \textit{IterationAssignment}, \textit{SubstitutionAssignment} \subset \textit{SubstructureAssignment} \subset \mathbb{D}_\mathbb{A}.$$
$$\textit{DirectPluginAssignment} \cap \textit{SubstructureAssignment} = \emptyset,\ \textit{IterationAssignment} \cap \textit{SubstitutionAssignment} = \emptyset$$

A rule of category *DirectPluginAssignment* assigns to a plugin holder a concrete executable program and optionally stipulates parameters passed thereto. A *SubstructureAssignment* rule instead rewrites a plugin holder into a *Substructure* or a *SubstructureSet*. Finally, we have two special assignments. The first is called *SubstitutionAssignment* and is used for substituting the plugin holder by an adequate *Substructure* with coinciding input and output. This enables us to define different instances of procedures for one step inside the given template. The second is called *IterationAssignment* and simulates the multiple assignment of a *SubstructureSet*, each time with a different parameter of the specified plugin. The iteration can be based on an integer sequence of on a set of strings.

The **syntax** of the rewriting rules $\mathbb{a} \in \mathbb{A}$ of the individual categories is summarized below

1. $\mathbb{a} \in$ *DirectPluginAssignment*.
   PluginHolder → *plugin*(p_1=v_1,...,p_n=v_n)
2. $\mathbb{a} \in$ *SubstructureAssignment*.
   Substructure → (t_1,...,t_n)
   SubstructureSet → (t_{11},...,t_{1n}) || ... || (t_{m1},...,t_{mn})
   SubstructureSetProcessor → Π (t_1,...,t_n)
3. $\mathbb{a} \in$ *IterationAssignment*.
   SubstructureSet → I [*iter_conf*] (t_1,...,t_n)
4. $\mathbb{a} \in$ *SubstitutionAssignment*.
   PluginHolder → *Substructure*

Determinacy of the workflow-generator function $\mathbb{t}$ is guaranteed by requiring that rewriting rules are applied to template elements in a topological order. Thus for each $\mathbb{e} \in \mathbb{E}$ which is being rewritten, $\forall \mathbb{f} \in \mathbb{E}$, $\forall \mathbb{s} \in \mathbb{S}$, $\mathbb{i}(\mathbb{s})=(\mathbb{f},\mathbb{e})$, $\mathbb{f}$ must be terminal. The motivation for requiring this particular order is that the transformation of elements of *SubstructureSetProcessor* are dependent on the result of the transformation of preceding *SubstructureSet* elements.

The **semantics** of the rules is as follows**.** The *plugin* symbol is terminal, carrying information identifying the plugin to be executed including parameters (p_i – parameter name, v_i – parameter value). The symbol (t_1,...,t_n) is a tuple of non-terminal symbols which rewrite the substructure inner graph. The *Substructure* rewriting has two steps. Firstly, the element is rewritten by its substructure graph ($\mathbb{E}'$, $\mathbb{S}'$, $\mathbb{i}'$, $\mathbb{I}'$, $\mathbb{O}'$).

All incoming edges are redirected to input nodes I' and so the outgoing edges are redirected to output nodes O'. Edges are duplicated if there are more than one input node (or output node, respectively). Secondly, new nodes are renamed according to the tuple members ($t_1,...,t_n$). For example, consider the template in Figure 1 and a rule I: $C \rightarrow (L_1,M_1,N_1)$. The red rectangle (L) is the input node of the inner graph, the green one (N) is the output node of it.

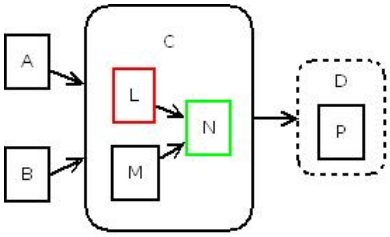

Figure 1. Example template definition

Then the resulting graph after rewriting has the form shown in Figure 2.

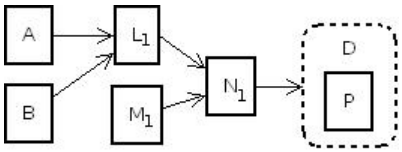

Figure 2. Graph after applying rewrite rule I.

The sign || indicates graph branching. If we consider C from the previous example a *SubstructureSet* and define the rule II: $C \rightarrow (L_1,M_1,N_1)||(L_2,M_2,N_2)$ we finally obtain a graph displayed in Figure 3.

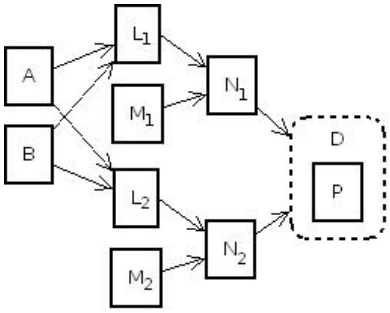

Figure 3. Graph after applying rewrite rule II.

The sign Π performs automatic branching according to the number of preceding *SubstructureSet* branches. In our example, we can have a rule III: $D \rightarrow \Pi\ (P_1)$ which entails the graph shown in Figure 4.

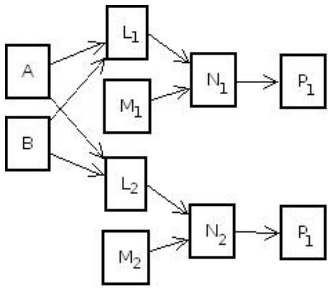

Figure 4. Graph after applying rules II. and III.

The iteration symbol I [*iter*_conf] branches the graph automatically according to the *iter_conf* pattern. It has the form $p = n_{start};step;n_{end}$ for integer value iteration or $p=\{s_1,\ldots,s_n\}$ for string value iteration. In both cases *Iteration assignment* automatically sets up the parameter p (of any plugin which supports it).

2.2 Implementation Remarks

Vertices, edges, rewrite rules as well as templates are formally specified in an ontological environment, in particular through the OWL language [4]. A dedicated algorithm is used to transform the OWL definitions into a *WorkflowDefinition* Java graph object structure which can be then easily translated into DAGMan format to be executed in Condor distributed environment [7]. Likewise, it can be executed locally just following the topological order of the tasks represented by graph nodes.
For each plugin, a set of inputs and set of outputs is defined. They are described by data types which determine the form of data which are accepted and produced by plugins. Data types are organized in a hierarchy where each child is a more specific version of its parent. It means that plugins accepting parent data type, accept all its children data types as well. There is a special data type called *Set* which can store a set of other data types, also including sets.

3. XGENE.ORG case study

The XGENE.ORG environment and example experiments are described in [2]. The following steps are involved:

1. GSM / GPL Data fetching from the public database NCBI GEO Database [10]
2. Normalization and scaling
3. Cross-platform gene-set data generation (using background knowledge – Pathways structure, Fully coupled fluxes, Gene Ontologies…)

4. Statistical, machine learning and visualization methods to obtain models distinguishing between defined sample classes

Here we first show how to represent the listed steps as workflow templates using our approach. Then we encapsulate them into a more complex template which additionally involves the cross-validation procedure for choosing the best instantiation of the workflow. Subsequently, we encapsulate it again in a template which automatically tests the workflow selection and builds up a histogram to visualize the results. Finally we show examples of rewriting rules used for such an experiment.

3.1. Basic templates definition

To enable visual understanding, we will use a graphical representation of the template definitions. Table 1 summarizes the graphical elements and their meaning in workflow templates.

Table 1. Semantics of template diagrams.

Graphical form	Meaning
Solid rectangle	Plugin Holder (grayed – supposed to be substituted by Structure)
Dashed rectangle	Substructure Template
Rounded rectangle	Substructure Set Template
Rounded dashed rect.	Substructure Set Processor
Solid arrow	Succession relation
Dashed arrow	Input or output from / for the parent structure template

Each Plugin Handler carries information about its name, as well as Input (I) and Output (O) data types. Inputs and output types of *Substructures* can be derived from the appropriate sub-elements regarding the rule that each *SubstructureSet* always produces the Set data type which carries the data from all generated branches.

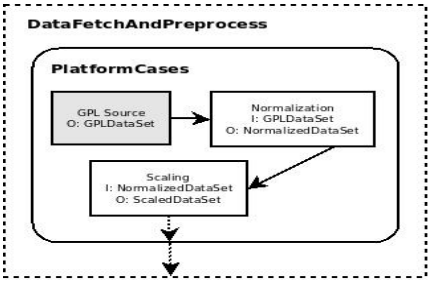

Figure 5. Substructure responsible for providing multi-platform data

The first two of the listed XGENE.ORG experiment steps are simply expressed in Figure 5. The *PlatformCases* is a SubstructureSet because we expect multiple platforms to do the cross-genome analysis.

Regarding Step 3, Cross-platform gene-set data generation is carried out by a Substructure shown in Figure 6. Elements signed with the BK abbreviation hold places for the Background Knowledge providers. These are plugins which provide the data needed for abstract working unit generation such as microarray probes to genes mappings or gene-sets definitions.

The other Plugin Holders follow the process of ranking, filtering and aggregating data measured on microarrays into abstract units which enable data fusion for cross genome analysis [2]. The output of the whole Substructure is the data transformed to the demanded form.

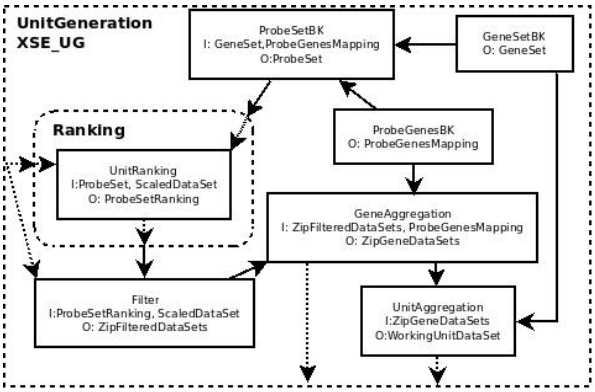

Figure 6. Substructure generating working units

Step 4 involves predictive classifier learning tasks in which predictive accuracy is the performance criterion. The corresponding Substructure, shown in Figure 7, takes two data sets on input annotated as TrainSet and TestSet which are distributed to appropriate Plugin Holders. The Train Plugin Holder is responsible for generating a classification model using train data set. The generated model is then evaluated in the Test Plugin Holder using the test data set.

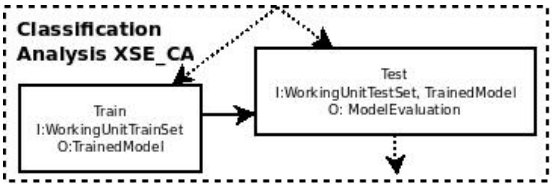

Figure 7. Classification Substructure

3.2. Inspecting performance of workflow paths

Templates defined in the previous section serve as building blocks for constructing more complex workflows. In this section we show, how to use template approach to define a workflow which collects classification accuracies, where cross-validation is used to determine the best (i.e. accuracy maximizing) path through the workflow (determining the configuration of plugins). The process is separated into two parts: validating of various combinations of working units and analysis algorithms, and collecting repeated validation results in order to build up the histogram. The first step is shown in Figure 8.

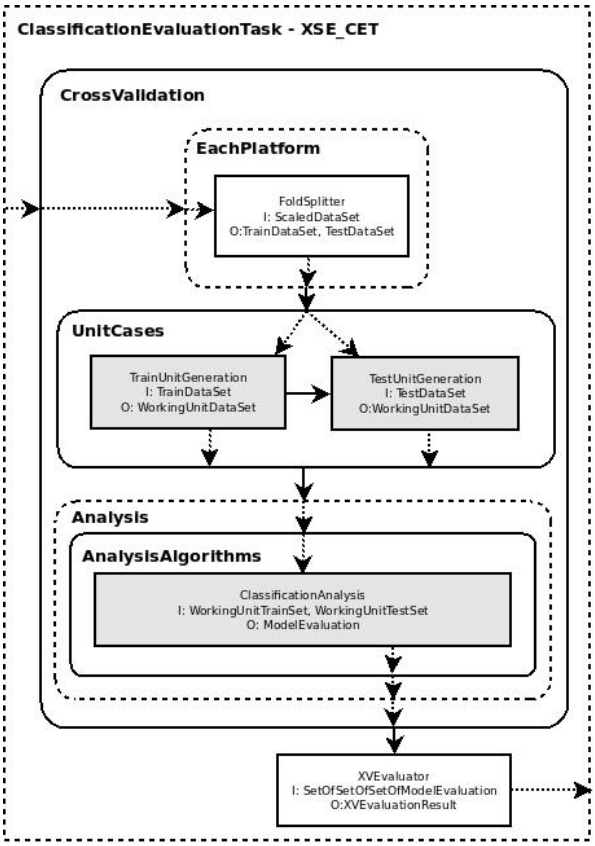

Figure 8. Template for cross-validation of combinations of working units and analysis algorithms

The *CrossValidation* block is supposed to be used with *IterationAssignment* to set the *FoldSplitter* plugin parameters automatically. The *TestUnitGeneration* is dependent on the *TrainUnitGeneration* because it has to use the same rankings for gene set filtering and the same aggregation matrices. The unit generation for test sets is realized by *Substructure* shown in Figure 9.

To achieve our goal we need to encapsulate the whole experiment into template responsible for generating the histogram (and thus repeatedly call experiments with various configurations and collect evaluation results). The template which can perform it is displayed in Figure 10.

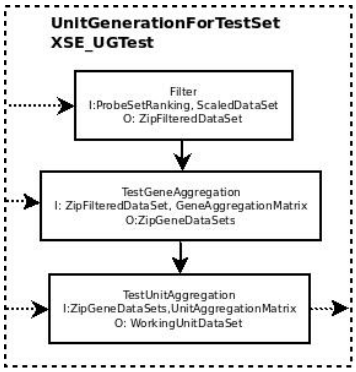

Figure 9. Unit generation for test sets

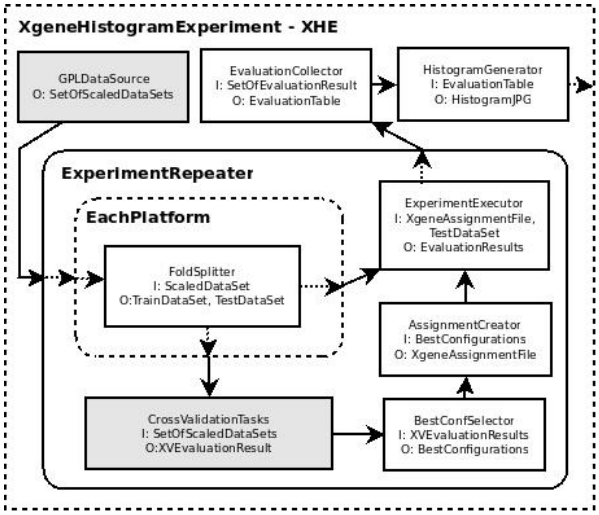

Figure 10. Histogram experiment template

ExperimentRepeater uses an *IterationAssignment* to repeat the experiment in order to collect enough data to build the histogram. Inside it we distinguish test and train data because we need to evaluate our configuration selection with the independent data. The *CrossValidationTasks* is to be substituted by *ClassificationEvaluationTask* proposed in the previous step. The *AssignmentCreator* and the *Experiment-Executor* are two plugins which compute results of the best selected configuration. It is performed by calling the workflow executing application externally from within the template.

The *EvaluationCollector* just collects results from all the iteration steps of the repeater. Finally the *Histo-*

gramGenerator takes this collection and generates the histogram.

4. Conclusions and future work

The main goal of this study was to propose a method of constructing workflows semi-automatically using a system of templates. The formal method for template description was proposed and implemented using an OWL ontology. The method includes the whole life-cycle of the experiment preparation, starting in data type and plugin definitions, over workflow template definition, and finishing in the assignment which prescribes the form of the final workflow. An algorithm which translates the prescriptions into the abstract workflow has been designed, implemented and tested. We have exemplified the expressiveness of the method by designing a complex experiment in cross-genome set-level gene expression analysis empirically evaluated in [3].

In principle, there are two possible template-based approaches to automatically explore possible configurations of workflows and select those with the best performance. The first approach generates a high number of small single-case workflows while the second one generates only one complex (branched) workflow containing all the possible cases as different paths through it. We have explored the second option due to computational considerations. Results of most workflow steps can be shared by the succeeding branches and so do not have to be computed repeatedly.

Currently we work on a specialized scheduling method which further optimizes the produced workflows for their execution in a GRID environment. The price we had to pay for our design choice is that the necessity to maintain all inspected cases in a single large workflow makes it impossible to exploit the simple non-deterministic graph rewriting framework. Rather we needed to extend the expressiveness of conventional rewrite rules by establishing a number of symbols with special semantics.

The ongoing work is focused on developing various plugins for the gene expression analysis. We are currently working on plugins for the gene-set selection (SAM-GS, global test, GSEA), aggregation (SVD, SetSig) as well as plugins for fetching data from different sources (NCBI public databases, Bioconductor, Broad Institute). We also work on the framework which uses the proposed template system to optimize workflows for the Condor GRID environment.

Acknowledgements

JB, MH and FZ are supported by the Czech Science Foundation project 201/09/1665. Travel of JB, DM, and MH has been supported by the bilateral project MEB11105/RC0904 (MEYS (Cz)/MINCyT (Ar)). JB and MH are further suppported by the Czech Technical University internal student grant SGS10/071/OHK4/1T/13.

References

[1] M. Žáková, P. Křemen, F. Železný, and N. Lavrač, Automatic Knowledge Discovery Workflow Composition through Ontology-Based Planning, *IEEE Transactions on Automation Science and Engineering,* vol.8, no.2, pp.253-264, April 2011

[2] Holec, M., Klema, J., Zelezny, F., Belohradsky, J., Tolar, J.: Cross-Genome Knowledge-Based Expression Data Fusion, *Int.l Conf. on Bioinformatics, Computational Biology, Genomics and Chemoinformatics*, 2009.

[3] Klema J., Holec M., Zelezny F., Tolar J.: Comparative Evaluation of Set-Level Techniques in Microarray Data Classification. *ISBRA 2011: 7th Int. Sympos. on Bioinformatics Research and Applications*, Springer 2011

[4] Patel-Schneider P, Hayes P, and Horrocks I., "OWL web ontology language semantics and abstract syntax," W3C recommendation, 2004

[5] Bernstein A. and Deanzer M., "The NExT system: Towards true dynamic adaptions of semantic web service compositions (system description)," in Proc. 4th Eur. Semantic Web Conf. (ESWC'07)), 2007,

[6] Gil Y., Ratnakar V., Deelman E., Mehta G., and Kim J., "Wings for Pegasus: Creating large-scale scientific applications using semantic representations of computational workflows," in Proc. 19th Annu. Conf. Innovative Appl. Artif. Intell., 2007, pp. 1767–1774.

[7] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed Computing in Practice:The Condor Experience. Concurrency and Computation: Practice and Experience, 17(2-4):323-356, 2005.

[8] Holec M., Zelezny F., Klema J., Tolar J.: Integrating Multiple-Platform Expression Data through Gene Set Features. ISBRA 2009: the 5th International Symposium on Bioinformatics Research and Applications (Ft. Lauderdale, Florida 5/09

[9] Bild A. and George Febbo P.. Application of a priori established gene sets to discover biologically important differential expression in microarray data. PNAS, 102(43):15278–15279, 2005.

[10] Edgar R, Domrachev M, and Lash A.. Gene expression omnibus: NCBI gene expression and hybridization array data repository. Nucleic Acids Res., 30(1):207–10, 2002

[11] Wei Huang D., Sherman B., and Lempick R.. Systematic and integrative analysis of large gene lists using DAVID bioinformatics resources. Nature Protocols, 4:44 – 57, 2009.