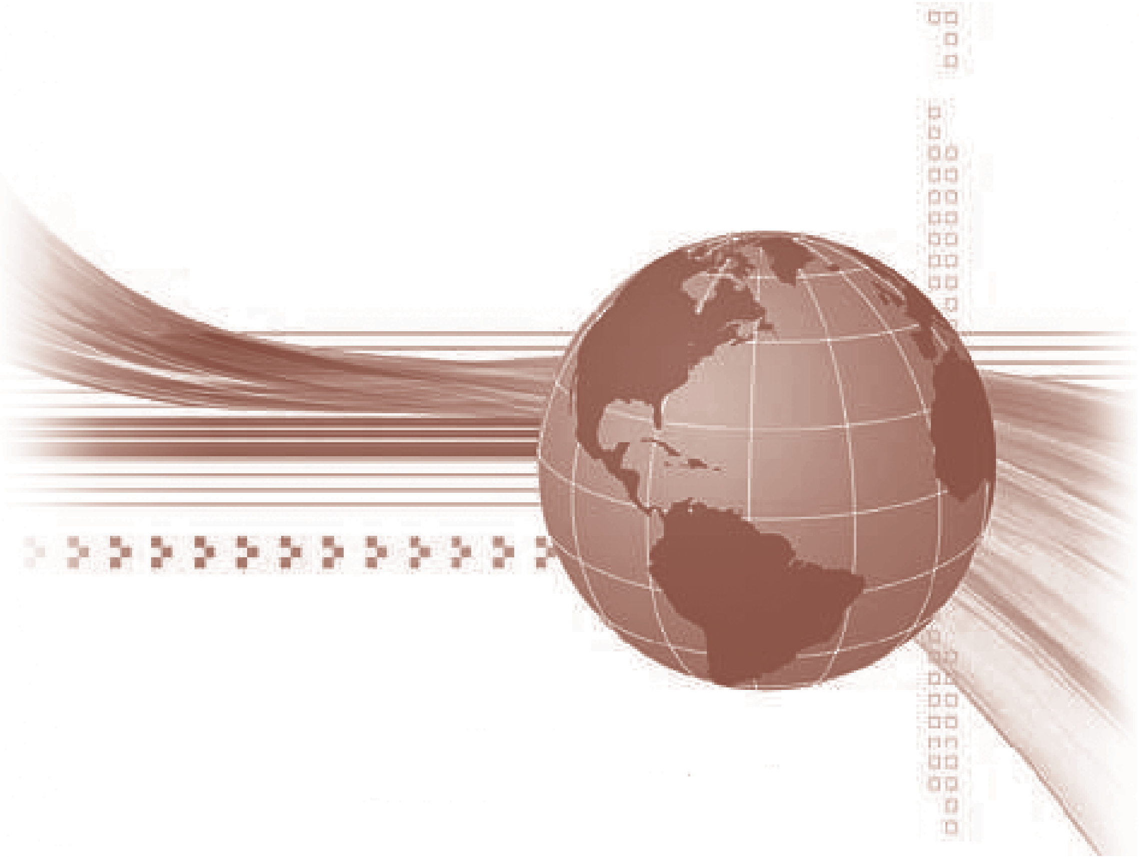
STUDIA UNIVERSITATIS
BABEŞ-BOLYAI



INFORMATICA

2/2008

Anul LIII **2008**

STUDIA
UNIVERSITATIS BABEŞ-BOLYAI

INFORMATICA

2

Redacţia: 400084 Cluj-Napoca, Str. M. Kogălniceanu nr. 1 Tel: 405300

SUMAR – CONTENTS – SOMMAIRE

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LII**, Number 2, 2008

CONSTRAINT OPTIMIZATION-BASED COMPONENT SELECTION PROBLEM

ANDREEA VESCAN AND HORIA F. POP

Abstract. Component-Based Software Engineering (CBSE) is concerned with the assembly of pre-existing software components that leads to a software system that responds to client-specific requirements. Component selection and component assembly have become two of the key issues involved in this process.

We aim at a selection approach that guarantees the optimality of the generated component-based systems, an approach that considers at each step the cost of the selected component and the set of requirements remaining to be satisfied. The dependencies between requirements are also considered. We have modeled the Component Selection Problem as a Constraint Satisfaction Optimization Problem and applied the Branch and Bound algorithm. The experiments and comparisons with the Greedy algorithm show the effectiveness of the proposed approach.

1. Introduction

Since the late 90's Component Based Development (CBD) is a very active area of research and development. CBSE [5] covers both component development and system development with components. There is a slight difference in the requirements and business ideas in the two cases and different approaches are necessary. Of course, when developing components, other components may be (and often must be) incorporated and the main emphasis is on reusability. Components-based software development is focused on the identification of reusable entities and relations between them, starting from the system requirements.

Building software applications using components significantly reduces development and maintenance costs. Because existing components can often be

Received by the editors: October 5, 2008.

2000 *Mathematics Subject Classification.* 68W01, 68N01.

1998 *CR Categories and Descriptors.* D.2 [**Software Engineering**]: Subtopic – *Reusable Software*; D. 1 [**Programming Techniques**]: Subtopic – *Object-oriented Programming* .

Key words and phrases. Component selection problem, Constraint Satisfaction Optimization Problem, Automatic assembly.

reused to build new applications, it is less expensive to finance their development.

In this paper we address the problem of automatic component selection. Generally, different alternative components may be selected, each coming with their own set of offered functionalities (in terms of system requirements). We aim at a selection approach that guarantees the optimality of the generated component-based system, an approach that considers at each step the component with the maximum set of offered functionalities needed by the final system. In our previous research, disseminated in [14], the dependencies between requirements were not taken into account. The current paper considers also the requirements dependencies during the selection process. The compatibility of components is not discussed here, as it will be dealt with in a future development.

We discuss the proposed approach as follows. Related work on Component Selection Problem is discussed in Section 6. Section 2 introduces our approach for Component Selection Problem: Subsection 2.1 presents a formal statement of the Component Selection Problem (CSP), the necessity of normalization in Subsection 2.3 and the modeling of the CSP as Constraint Optimization Problems (COP) in Subsection 2.4. A Greedy and a Branch and Bound approaches are considered. Section 3 presents the elements of the Greedy algorithm and the chosen selection function. The Branch and Bound algorithm is presented in Section 4. Using the example in Section 5 we discuss the two proposed approaches: Greedy and Branch and Bound. We conclude our paper and discuss future work in Section 7.

2. Constructing component-based systems by automatic component selection

In Component-Based Software Engineering, the construction of cost-optimal component systems is a nontrivial task. It requires not only to optimally select the components but also to take their interplay into account.

We assume the following situation: Given a repository of components and a specification of the component-based system that we want to construct (set of final requirements), we need to choose components and to connect them such that the target component-based system fulfills the specification. Informally, our problem is to select a set of components from an available set which may satisfy a given set of requirements while minimizing the number of selected components and minimizing the sum of the costs of the selected components. To achieve this goal, we should assign to each component a set of requirements it satisfies.

2.1. **Formal Statement of the Component Selection Problem.** Component Selection Problem (CSP) is the problem of choosing the minimum number of components from an available set such that their composition satisfies a set of objectives (variation of CSP, the cost of each component is not considered). The notation used for formally defining CSP (as laid out in [6] with a few minor changes to improve appearance) is described in what follows.

Problem statement. *Denote by SR the set of final system requirements (target requirements)* $SR = \{r_1, r_2, ..., r_n\}$, *and by SC the set of components available for selection* $SC = \{c_1, c_2, ..., c_m\}$. *Each component* c_i *may satisfy a subset of the requirements from SR,* $SR_{c_i} = \{r_{i_1}, r_{i_2}, ..., r_{i_k}\}$. *In addition* $cost(c_i)$ *is the cost of component* c_i. *The goal is to find a set of components Sol in such a way that every requirement* r_j $(j = \overline{1,n})$ *from the set SR may have assigned a component* c_i *from Sol where* r_j *is in* SR_{c_i}, *while minimizing* $\sum_{c_i \in SSol} cost(c_i)$ *and having a minimum number of used components.*

2.2. **Requirement dependencies.** In [13] we have introduced the matrix for the requirements dependencies.

In Table 1 the dependencies between the requirements r_1, r_2, r_3 are specified: the second requirement depends on the third requirement, the third requirement depends on the first and the second requirement.

Dependencies	r_1	r_2	r_3
r_1	√	√	
r_2			√
r_3	√	√	

TABLE 1. Dependencies specification table

Some particular cases are required to be checked: no self dependency (the first requirement depends on itself), no reciprocal dependency (the second requirement depends on the third and the third depends on the second requirements) and no circular dependencies (the second requirement depends on the third, the first depends on the second and the third depends on the first). All the above situations are presented in Table 1.

2.3. **Data normalization.** Normalization is an essential procedure in the analysis to compare data having different domain values. It is necessary to make sure that the data being compared is actually comparable. Normalization will always make data look increasingly similar. An attribute is normalized by scaling its values so that they fall within a small-specified range, such as 0.0 to 1.0.

As we have stated above we would like to obtain a system by composing components, a system that will have a minimum final cost and all the requirements are satisfied. The cost of each available component is between 0 and 100. At each step of the construction the number of requirements not yet satisfied is considered as a criterion to proceed with the search. We must normalize the cost of the components and also the number of requirements yet to be satisfied.

We have used two methods to normalize the data: decimal scaling for the cost of the components and min-max normalization for the requirements not yet satisfied.

Decimal scaling. The decimal scaling normalizes by moving the decimal point of values of feature X. The number of decimal points moved depends on the maximum absolute value of X. A modified value new_v corresponding to v is obtained using:

$$new\_v = \frac{v}{10^n},$$

where n is the smallest integer such that $max(|new\_v|) < 1$.

Min-max normalization. The min-max normalization performs a linear transformation on the original data values. Suppose that $minX$ and $maxX$ are the minimum and maximum of feature X. We would like to map interval $[minX, maxX]$ into a new interval $[new\_minX, new\_maxX]$. Consequently, every value v from the original interval will be mapped into value new_v using the following formula:

$$new\_v = \frac{v - minX}{maxX - minX}.$$

Min-max normalization preserves the relationships among the original data values.

2.4. **Constraint Optimization-based Component Selection Problem.**

Constraint Satisfaction Problems (CSPs) are mathematical problems where one must find objects that satisfy a number of constraints or criteria. CSPs are the subject of intense research in both artificial intelligence and operations research. Many CSPs require a combination of heuristics and combinatorial search methods to be solved in a reasonable time.

In many real-life applications, we do not want to find any solution but a good solution. The quality of solution is usually measured by an application dependent function called objective function. The goal is to find such solution that satisfies all the constraints and minimize or maximize the objective function respectively. Such problems are referred to as Constraint Satisfaction Optimization Problems (CSOP).

A Constraint Optimization Problem can be defined as a regular Constraint Satisfaction Problem in which constraints are weighted and the goal is to find a solution maximizing the weight of satisfied constraints. A Constraint Satisfaction Optimization Problem consists [4] of a standard Constraint Satisfaction Problem and an optimization function that maps every solution to a numerical value. The most widely used algorithm for finding optimal solutions is Branch and Bound and it can be applied to CSOP as well. The Branch and Bound algorithm was first proposed by A. H. Land and A. G. Doig in 1960 for linear programming. In Section 4 a more detail description is given.

3. Greedy Algorithm

Greedy techniques are used to find optimum components and use some heuristic or common sense knowledge to generate a sequence of sub-optimums that hopefully converge to the optimum value. Once a sub-optimum is picked, it is never changed nor is it re-examined.

The Pseudocode of the Greedy algorithm is illustrated in Algorithm 1.

Algorithm 1 Greedy algorithm

```
Require: SR; {set of requirements}
    SC. { set of components }
Ensure: Sol. { obtained solution }
Sol := $\emptyset$; RSR := SR;{RSR=Remaining Set of Requirements}
while (RSR <> $\emptyset$) do
    Choose a $c_i$ from $SC$, not yet processed;
    @ Mark $c_i$ as processed.
    if Sol $\bigcup$ { $c_i$ } is feasible then
        Sol := Sol $\bigcup$ { $c_i$ };
        RSR := RSR - $SRc_i$;
    end if
end while
```

The selection function is usually based on the objective function. Our selection function considers the sum of number of requirements to be satisfied (function f) and the cost of the already selected components plus the cost of the new selected component (function g) to be minimal ($(g + h)$ *is minimal*) and all the dependencies are satisfied.

4. Branch and Bound Algorithm

Branch and Bound algorithms are backtracking algorithms storing the cost of the best solution found during execution and use it for avoiding part of the search. More precisely, whenever the algorithm encounters a partial solution that cannot be extended to form a solution of better cost than the stored best cost, the algorithm backtracks, instead of trying to extend this solution.

The term Branch and Bound refers to search methods which have two characteristics that makes them different from other searching techniques:

(1) The method expands nodes from the search tree (this expansion is called *branching*) in a particular manner, trying to optimize the search.

(2) The search technique uses a *bounding* mechanism in order to eliminate (not expand) certain branches (paths) that does not bring any improvements.

The problem solving using *B&B* technique is based on the idea of building a search tree during the problem solving process. By a *successor* of a node n we mean a configuration that can be reached from n by applying one of the allowed operations. By *expansion* of a node we mean to determine all the possible successors of the node.

The selection of the successors of a node must also take into consideration the dependencies between requirements. The list of successors of a node is thus reduced.

Because by expanding the initial configuration some configurations can be repeatedly generated, and because the number of nodes can be large, we will not store the entire tree, but only a list with the nodes (configurations) that have to be processed (denoted $SOLUTION\_LIST$). At a given time a node from $SOLUTION\_LIST$ can have one of the following states: *expanded* or *unexpanded*.

The main problem is what node for the list should be selected at a given moment in order to obtain the shortest solution of the problem. Each node n from the list has an associated value (cost function),

$$f(n) = g(n) + h(n),$$

where:

- $g(n)$ represents the cost of the components that were used until now (from the root node to node n) to construct the solution;
- $h(n)$ represents the number of remaining requirements that need to be satisfied (to reach the final solution starting from the current node n). The function h is called heuristic function.

The *B&B* [7] algorithm is described using Pseudocode in Algorithm 2.

5. Case study

In order to validate our approach the following case study is used.

Starting for a set of six requirements and having a set of ten available components, the dependencies between the requirements of the components, the goal is to find a subset of the given components such that all the requirements are satisfied.

The set of requirements $SR = \{r_0, r_1, r_2, r_3, r_4, r_5\}$ and the set of components $SC = \{c_0, c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, c_9\}$ are given.

In Table 2 the cost of each component from the set of components SC is presented. We have used decimal scaling to normalize the cost of the components.

Algorithm 2 Branch and Bound algorithm

```
Require: SR; {set of requirements}
    SC. { set of components }
Ensure: Sol. { obtained solution }
Select a component (node) from the set of available components $SC$. The component (node) is added into the list $SOLUTION\_LIST$, initially assumed empty (hereby called “the list”). This component has the cost as the value of the function $g$ and the total number of requirements in the set $SR$, yet to be satisfied, as the value of the function $h$.
while (unexpanded nodes still exist in the list) do
  Select from the list the unexpanded node $n$ having the minimum value for the function $f = g + h$.
  Expand node $n$ and generate a list of successors $SUCC$.
  for (each successor $succ$ from $SUCC$) do
    Compute the function $g$ associated to $succ$.
    Compute the function $h$ associated to $succ$, i.e. the number of remaining requirements from the set $SR$ that need to be satisfied to reach the final solution (with all the requirements satisfied) starting from the node $succ$.
    if (the value of $h$ is 0 (a solution is found)) then
      $Sol$ will memorize the best solution between the previously obtained solution (if exists) and the current obtained solution.
    else
      if (component $succ$ does not appear in the list) then
        Add $succ$ into the list with its corresponding cost value $f(succ) = g(succ) + h(succ)$ and mark as unexpanded;
      else
        if (the value $g(succ)$ is $<$ the $g$ value of the node found in the list) then
          The node found in the list is directed to the actual parent of $succ$ (i.e. $n$) and is associated with the new value of $g$. If the node was marked as unexpanded, its mark is changed.
        end if
      end if
    end if
  end for
end while
```

Table 3 contains for each component the provided services (in terms of requirements of the final system).

Table 4 contains the dependencies between each requirement from the set of requirements.

Component	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
Cost	12	7	3	9	6	14	8	14	7	6
Cost Normalization	0.12	0.07	0.03	0.09	0.06	0.14	0.08	0.14	0.07	0.06

Table 2. Cost values for each component in the *SC*

	c_0	c_1	c_2	c_3	c_4	c_5	c_6	c_7	c_8	c_9
r_0	√		√	√						√
r_1					√				√	
r_2		√				√			√	
r_3	√						√			
r_4						√	√	√		√
r_5		√					√	√		√

Table 3. Requirements elements of the components in *SC*

Dependencies	r_0	r_1	r_2	r_3	r_4	r_5
r_0		√				
r_1						
r_2	√				√	
r_3		√				
r_4	√					
r_5		√				

Table 4. Specification Table of the Requirements Dependencies

Table 5 contains the normalization of the number of remain requirements to be satisfied.

5.1. **Results obtained by Greedy algorithm.** In the current section we discuss the application of the Greedy algorithm (presented in Subsection 3) to our problem instance.

The first step of the selection function is the computation of the functions g and h: g is the cost of the used components and h is the number of requirements yet to be satisfied. The component with the minimum value of the function $f = g + h$ is chosen to be a part of the solution. The ties are broken randomly. The dependencies must be also satisfied.

In the first iteration of the algorithm the c_4 component has the minimum value for the function f, i.e. 0.89 and has no dependencies. The set of requirements that are satisfied by choosing the c_4 component is: $\{r_1\}$. Next, only

No. of requirements to be satisified	Normalization	Value
0	0/6	0
1	1/6	0.16
2	2/6	0.33
3	3/6	0.50
4	4/6	0.66
5	5/6	0.83
6	6/6	1

TABLE 5. Normalization of the number of requirements to be satisfied.

the components that may improve the solution (by satisfying new requirements) are considered: $\{c_0, c_1, c_2, c_3, c_5, c_6, c_7, c_8, c_9\}$ but only three of them have all the dependencies satisfied, i. e. $\{c_0, c_2, c_3\}$. The c_0 component has the smallest value of the f function (0.68) and this component is selected to be considered into the solution.

The set of requirements that must still be fulfilled is $\{r_2, r_4, r_5\}$. Only three components may provide some of the remaining requirements and at the same time having all the dependencies satisfied: $\{c_6, c_7, c_9\}$. The c_9 component has the smallest value of the f function (0.40) and this component is the next to be considered for selection.

There is only one requirement to be satisfied, i. e. $\{r_2\}$. Only three components may provide this functionality and all of them have the dependencies satisfied: $\{c_1, c_5, c_8\}$. The component with the minimum value for the g (0.31) function is the c_8 component.

The set of the requirements to be satisfied RSR is empty and we have reached a solution with all the requirements satisfied by the selected components: c_4, c_0, c_9 and c_8. The cost of the final solution 0.31 is the sum of the cost of the selected components. Still, we will see in the next Section 5.2 that there are better solutions with the final cost 0.24: $\{c_4, c_2, c_6, c_1\}$ or $\{c_4, c_2, c_6, c_8\}$.

5.2. **Results obtained by Branch and Bound algorithm.** The Branch and Bound algorithm initialize the first used component in the solution list with the component c_4 (the only component with no dependencies). The set of satisfied requirements is: $\{r_1\}$. The first iteration of the Algorithm 2 adds the $\{c_0, c_2, c_3\}$ components (ordered by the value of the function f) to the list $SOLUTION\_LIST$ (n represents *not expanded node* and e represents *expanded node*).

$$SOLUTION\_LIST = \left\langle \frac{c_0\ c_2\ c_3\ c_4}{n\ \ n\ \ n\ \ e} \right\rangle .$$

The next step of the algorithm expands the first unexpanded node from the list, i.e. c_0. The components that may provide some functionalities from the set of requirements to be satisfied are: $\{c_1, c_5, c_6, c_7, c_8, c_9\}$. Only three components have the dependencies satisfied. The list of successors is reduced to: $\{c_6, c_7, c_9\}$. The new list of nodes is: $\{c_9, c_6, c_7, c_0, c_2, c_3, c_4\}$ with two expanded nodes, components c_4 and c_0.

The next node to be expanded is c_9. Three solutions are found but only the best one is memorized: $\{c_4, c_0, c_9, c_8\}$ with cost 0.31. Next expanded nodes are c_6 and c_7 but the obtained solutions have the cost greater that the previously best obtained solution.

The expansion of the c_2 node modifies the list of nodes. From the list of components that may provide new needed functionalities only four of seven components have the dependencies satisfied: $\{c_0, c_6, c_7, c_9\}$. All the successor are already part of the list but, except the c_0 node, the value of the g function is smaller than the value from the list. The list of nodes is updated according to the new values of f functions.

$$SOLUTION\_LIST = \left\langle \frac{c_6 \; c_9 \; c_7 \; c_0 \; c_2 \; c_3 \; c_4}{n \;\; n \;\; n \;\; e \;\; e \;\; n \;\; e} \right\rangle.$$

The next node that is expanded is the node c_6. The successors are: $\{c_1, c_5, c_8\}$. The new obtained solution considering the c_1 component is better then the current best solution: the cost is $0.24 < 0.31$. The other two obtained solutions (with components c_5 and c_8) have the cost greater or equal that the cost of the new solution, i.e. 0.31 and 0.24.

By expanding next the node c_9 four components may provide the needed functionalities (r_2 or r_3) and all have the dependencies satisfied. For the components c_0 and c_6 the new values for g are greater than those from the list. Therefore the values for the stated components is not going to be changed. The other components will be included into the final list:

$$SOLUTION\_LIST = \left\langle \frac{c_6 \; c_1 \; c_9 \; c_5 \; c_7 \; c_0 \; c_2 \; c_3 \; c_4}{e \;\; n \;\; e \;\; n \;\; n \;\; e \;\; e \;\; n \;\; e} \right\rangle.$$

With the next expanded node two solutions are found but with the cost greater than the best found solution, i.e. 0.34 and 0.30. By expanding the other nodes no new solution may be found and no new nodes mat be added to the solution list.

The solution obtained with the Branch and Bound algorithm (considering the g function as the sum of the cost of the used components and the h function as the number of requirements to be satisfied) consists of the components: $\{c_4, c_2, c_6, c_1\}$ having the cost 0.24.

6. Related work

Component selection methods are traditionally done in an architecture-centric manner. An approach was proposed in [12], where the authors present a method for simultaneously defining software architecture and selecting off-the-shelf components. They have identified three architectural decisions: object abstraction, object communication and presentation format. Three type of matrix are used when computing feasible implementation approaches. Existing methods include OTSO [10] and BAREMO [11].

Another type of component selection approaches is built around the relationship between requirements and components available for use. In [8] the authors have presented a framework for the construction of optimal component systems based on term rewriting strategies. By taking these techniques from compiler construction they have developed an algorithm that builds a cost-optimal component-based system. In PORE [2] and CRE [1] the same relation between requirements and available components is used. The goal here is to recognize the mutual influence between requirements and components in order to obtain a set of requirements that is consistent with what the market has to offer. The [6] approach considers selecting the component with the maximal number of provided operations. The algorithm in [3] considers all the components to be previously sorted according to their weight value. Then all components with the highest weight are included in the solution until the budget bound has been reached.

Paper [9] proposes a comparison between a Greedy algorithm and a Genetic Algorithm. The discussed problem considers a realistic case in which cost of components may be different.

7. Conclusion and future work

CBSE is the emerging discipline of the development of software components and the development of systems incorporating such components. A challenge in component-based software development is how to assemble components effectively and efficiently.

A proposal for the Component Selection Problem as a Constraint Optimization Problem is given. Two considered approaches are: Greedy and Branch and Bound. Further work will investigate different criteria for component selection: dependencies, different non-functional qualities. A real world system application will be considered next to (better) validate our approach. We have discussed the case when only the dependencies between the requirements from the set of requirements SR. A more real case should be also considered: a component could have other requirements that need to be satisfied before some of its provided services are used.

8. Acknowledgement

This material is based upon work supported by the Romanian National University Research Council under award PN-II no. ID_550/2007.

References

[1] C. Alves, J. Castro, *Cre: A systematic method for cots component selection*, Proceedings of the Brazilian Symposium on Software Engineering, IEEE Press, 2001, pp. 193–207.

[2] C. Alves, J. Castro, *Pore: Procurement-oriented requirements engineering method for the component based systems engineering development paradigm*, Proceedings of the Int'l Conf. Software Eng. CBSE Workshop, IEEE Press, 1999, pp. 1–12.

[3] P. Baker, M. Harman, K. Steinhofel, A. Skaliotis, *Search Based Approaches to Component Selection and Prioritization for the Next Release Problem*, Proceedings of the 22^{nd} IEEE International Conference on Software Maintenance, IEEE Press, 2006, pp. 176–185.

[4] R. Bartk, *Constraint Programming, In Pursuit of the Holy Grail*, Proceedings of the Week of Doctoral Students, Part IV, MatFyzPress, 1999, pp. 555–564.

[5] I. Crnkovic, M. Larsson, *Building Reliable Component-Based Software Systems*, Norwood: Artech House publisher, 2002.

[6] M. R. Fox, D. C. G. Brogan, P. F. Reynolds, *Approximating component selection*, Proceedings of the 36^{th} conference on Winter simulation, 2004, pp. 429–434.

[7] M. Frentiu, H. F. Pop, G. Serban, P*rogramming Fundamentals*, Cluj University Press, 2006.

[8] L. Gesellensetter, S., Glesner, *Only the Best Can Make It: Optimal Component Selection*, Electron. Notes Theor. Comput. Sci. 176 (2007), pp. 105–124.

[9] N. Haghpanah, S. Moaven, J., Habibi, M., Kargar, S. H., Yeganeh, *Approximation Algorithms for Software Component Selection Problem*, Proceedings of the 14^{th} Asia-Pacific Software Engineering Conference, IEEE Press, 2007, pp. 159–166.

[10] J. Kontio, *OTSO: A Systematic Process for Reusable Software Component Selection*, Technical report, University of Maryland, 1995.

[11] A. Lozano-Tello, A. Gómez-Pérez, *BAREMO: how to choose the appropriate software component using the analytic hierarchy process*, Proceedings of the 14^{th} International Conference on Software engineering and knowledge engineering, ACM, 2002, pp. 781–788.

[12] E. Mancebo, A. Andrews, *A strategy for selecting multiple components*, Proceedings of the Symposium on Applied computing, ACM, 2005, pp. 1505–1510.

[13] A. Vescan, *Dependencies in the Component Selection Problem*, Proceedings of the International Conference of Applied Mathematics, Baia–Mare, Romania, 2008 (accepted).

[14] A. Vescan, H. F. Pop, *The Component Selection Problem as a Constraint Optimization Problem*, Proceedings of the Work In Progress Session of the 3^{rd} IFIP TC2 Central and East European Conference on Software Engineering Techniques (Software Engineering Techniques in Progress), Wroclaw University of Technology, Wroclaw, Poland, 2008, pp. 203-211.

Department of Computer Science, Faculty of Mathematics and Computer Science,, Babeş-Bolyai University, Cluj-Napoca, Romania,

E-mail address: {avescan,hfpop}@cs.ubbcluj.ro

EXTENSION OF AN OCL-BASED EXECUTABLE UML COMPONENTS ACTION LANGUAGE

S. MOTOGNA, B. PÂRV, I. LAZĂR, I. CZIBULA, L. LAZĂR

Abstract. Executable models allow precise description of software systems at a higher level of abstraction and independently of a platform or a programming language. In this paper we explore the use of a Procedural Action Language based on OCL to specify executable UML components and we propose an extension that will include array types and corresponding operations.

1. Introduction

Model Driven Architecture (MDA) development represent a pertinent solution to design and control of large software systems, while UML establihed itself as a standard for software models. UML2 and its Action Semantics [6] provide the foundation to construct executable models. In order to make a model executable, the model must contain a complete and precise behavior description. But, creating a model that has a complete and precise behavior description is a tedious task or an impossible one because of many UML semantic variation points.

We have introduced COMDEVALCO a framework aimed to support definition, validation, and composition of software components, that allows the construction and execution of UML structured activities [3]. The framework refers only to UML structured activities because our first objective was to allow model transformation from PIM (Platform Independent Model) to procedural constructs of imperative languages. It includes a modeling language, a component repository and a set of tools. The object-oriented modeling language contains finegrained constructions, aimed to give a precise description of software components. Component repository is storing valid components,

Received by the editors: October 12, 2008.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification – *Formal methods, Model checking, Validation*; D.2.13 [**SOFTWARE ENGINEERING**]: Reusable Software – *Reuse models*; I.6.5 [**SIMULATION AND MODELING**]: Model Development – *Modeling methodologies* .

Key words and phrases. Software components, Executable models, OCL.

ready to be composed in order to build more complex components or systems. The toolset contains tools dedicated to component definition, validation, and composition, as well as the management of component repository.

Our approach uses Procedural Action Language (PAL) that is a concrete syntax for UML structured activities and defines graphical notations for some UML structured activity actions [7]. PAL simplifies the process of constructing executable models by simplifying the construction of UML structured activities.

The execution of the model is performed without any transformation, and by using this approach the time delay between the model changes and the model execution is minimized.

The repository should contain executable models, ready to be used in any further development. One aspect that can guarantee this principle is to use some extra conditions, such as preconditions, postconditions and invariants to the model definition that would describe in a formal way, the behavior of the model.

Object Constraint Language - OCL - has been extensively used for models of UML [6], representing a well suited specification language for defining constraints and requirements in form of invariants, pre- and post- conditions.

So, we add pre- and post-conditions to the model in the form of OCL expressions. In such a way, we obtain the desired descriptions in terms of OCL expressions, we then could use them in searching queries, and the layout of the repository can be standardized.

The repository will store different types of models, and in the initial phase, we have designed it for simple arithmetical and array problems. The OCL specification [5] doesn't contain array types, which are necessary in our approach. So, we have two options to tackle this problem: to express arrays using the existing constructions or to extend OCL Expressions.

The first approach has two main disadvantages: it restricts the type of the elements of the arrays and array specific operations should be re-written any time they are needed. We would prefer to work with a more generic construction, and do not worry about operations' implementations each time they are used. Array operations are defined once, and then called any time they are needed.

The rest of the paper is organized as follows: the next section presents some related works in the domain and compare them with our approach. Section 3 describes the action language defined as part of ComDeValCo framework and then section 4 presents the extension of PAL with array types and associated operations, and an example of an executable model that benefits from the use of our extension. The next section draws some conclusions and suggests some future development directions.

2. Related work

The xUML [8] process involves the creation of platform independent, executable UML models with the UML diagrams being supported by the action semantics-compliant Action Specification Language (ASL). The resulting models can be independently executed, debugged, viewed and tested. The action semantics extension to UML defines the underlying semantics of Actions, but does not define any particular surface language. The semantics of the ASL are defined but the syntax of the language varies. ComDeValCo is compliant with UML 2.0 and uses structured activities for models [3].

According to several domain experts, a precise Action Semantics Language (ASL) and a specified syntax are required. Unfortunately, actions defined in UML do not have a concrete syntax and OMG does not recommend a specific language, so there is not a standard ASL. Object Constraint Language (OCL) is a formal language used to describe expressions on UML models. The great overlap between ASL and OCL (large parts of the Action Semantics specification duplicates functionality that is already covered by the OCL) suggests that OCL can be used partly for ASL. OCL for Execution (OCL4X) [2] is defined based on OCL to implement operations that have side effects and provide the ability for model execution. By mapping from ASL to OCL, OCL is used to express some actions in ASL. This approach has identified some open problems when using OCL in specification of the executable models, and offered solutions based on extending OCL to include actions with side effects in order to model behavior. Our approach is, in many ways, similar to this one. We are also proposing some extensions of OCL, but based on identifying some other problems and suggesting more efficient approaches of executable model specification.

According to Stefan Haustein and Jorg Pleumann, since the OCL is a subset of the ASL, there are two options for building an action surface language based on OCL [1]: map all OCL constructs to actions, then add new syntax constructs for actions that are required, but not covered, or embed OCL expressions in new syntax constructs for actions.

The first option requires a complete mapping of the abstract OCL syntax to actions. This would mean to give up declarative semantics in OCL, or to have two flavours of OCL with different specifications that would need to be aligned carefully.

The second option can be implemented by referring to the existing OCL surface language, without modifying it, maintaining a clean syntactical separation between plain queries and actions that may influence the system state.

ComDeValCo is oriented on this second approach.

3. Procedural Action Language - description and features

As part of ComDeValCo framework we have defined a procedural action language (PAL), that is a concrete syntax for UML structured activities, and graphical notations for some UML structured activity actions [7].

The framework also includes an Agile MDA approach for constructing, running and testing models. Debugging and testing techniques are also included according to the new released standards. In order to be able to exchange executable models with other tools, a UML profile is also defined. The profile defines the mapping between PAL and UML constructs and is similar to the profile defined for AOP executable models.

In order to develop a program we construct a UML model that contains functional model elements and test case model elements. Functional model elements correspond to the program and its operations and are represented as UML activities. Test case model elements are also UML activities and they represent automated tests written for some selected functional model elements.

The Procedural Action Language (PAL) is introduced to simplify the construction of UML structured activities. PAL defines a concrete syntax for representing UML structured activity nodes for loops, sequences of actions and conditionals. The PAL syntax is also used for writing assignment statements and expressions in structured activity nodes. PAL also includes assertion based constructs that are expressed using OCL expressions.

The syntax of the language is given in Appendix A.

The framework accepts user-defined models described in UML-style or using PAL, validates them according to UML metamodel and construct the abstract syntax tree, which is then used to simulate execution. For each syntactical construction of PAL there exists a rule corresponding to the construction of the abstract syntax tree.

4. Extending PAL with array type

The intention is to store different types of models in the repository, but, in the initial phase, we have considered small models for simple arithmetical problems, and we face the problem of dealing with arrays. As mentioned before PAL uses OCL-based expressions, but the OCL specification language does not allow arrays.

There are two things that should be taken into consideration when designing types for models [9]:

- Languages that manipulate and explore models need to be able to reason about the types of the objects and properties that they are regarding within the models.

- There is also a need to reason about the types of artifacts handled by the transformations, programs, repositories and other model-related services, and to reason about the construction of coherent systems from the services available to us. While it is possible to define the models handled by these services in terms of the types of the objects that they accept, we argue that this is not a natural approach, since these services intuitively accept models as input, and not objects.

At the first attempt, it would have looked simpler to add a new type array that could create arrays with elements of any existing type in the system, but taking a deeper look, creating an array of integers is totally different from creating an array of components. Consequently, we have though at the approach that is also taken in different strongly typed programming language (Java, .NET), and that will guarantee an easy extension of the type system.

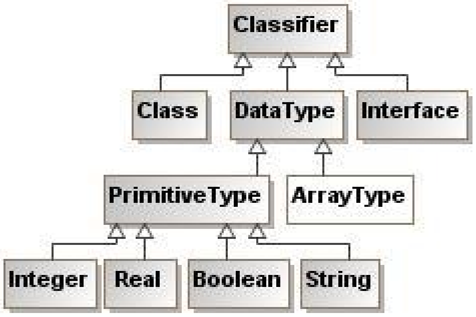


FIGURE 1. Type metaclasses

We have started from the type hierarchy from OCL [5] and refined it to integrate *ArrayType*, as depicted in Figure 1. A *Classifier* may be a *DataType*, a *Class* or an *Interface*. *ArrayType* and *PrimitiveType* are specializations of *DataType*. The most important feature of *DataType* is that a variable of this type can hold a reference to any object, whether it is an integer, a real or an array, or any other type.

We highlight only the modifications of the grammar such that our models will be able to handle arrays and records. Types can be arrays whose elements can be of any type. Records will be structures that will group together a

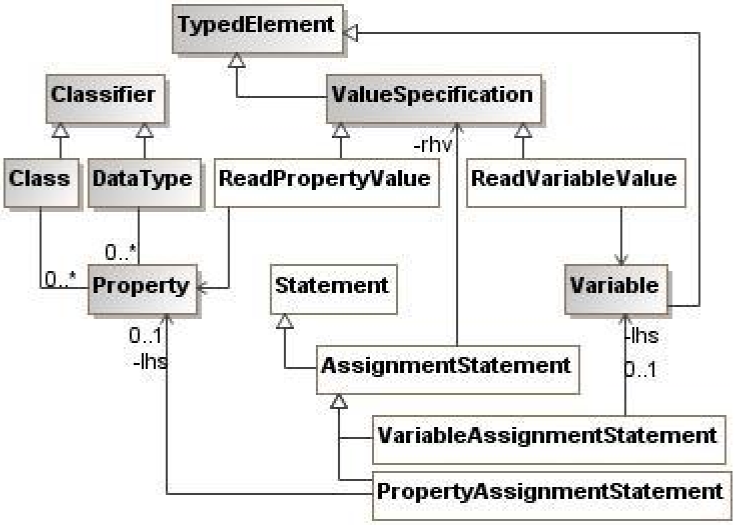


FIGURE 2. Assignment statements

number of fields, such that a field is similar to a variable declaration.

$$\begin{aligned}
TYPE &: DataType|Class|Interface\\
DataType &: PrimitiveType|ArrayType\\
PrimitiveType &: Integer|Boolean|String|Real|\\
ArrayType &: TYPE[DOMAIN]\\
DOMAIN &: INT..INT|INT..*
\end{aligned}$$

In the specification of the domain, first case describes an array of known size at declaration, and the second case specifies an array whose size is not known when declared.

Consequently, we will allow expressions to contain values of the newly introduced types, namely the value of an element of the array, and the value of a field from a record:

$$\begin{aligned}
atom &: ID|INT|STRINGLITERAL|\\
ID'('(expr)?(','expr)*endN=')'&|e=TRUE|\\
e=FALSE&|condition|ID'[INT']'
\end{aligned}$$

The statements that involve expressions need also to be revised. Figure 2 shows part of the syntax, without specifying all the statements. The complete syntax is presented in the Appendix. The dashed components are the ones defined in UML and the white-box components are the ones introduced in ComDeValCo. Assignment statement is further specialized in two categories, depending on its left-value: for variables or for properties. According to UML 2.1 *Property* can be associated to a *Class* or to a *DataType*.

The syntactical rules corresponding to these statements are:

$$\begin{aligned}
AssigmentStatement &: VarAssignStatement | PropAssignStatement \\
VarAssignStatement &: IDASSIGNexpr \\
PropAssignStatement &: Classname.PropASSIGNexpr | \\
&\quad ID[ID]ASSIGNexpr
\end{aligned}$$

We adopt the same approach as the OMG Specification of OCL: we consider that there is a signature $\Sigma = (T, \Omega)$ with T being a set of type names, and Ω being a set of operations over types in T. The set T includes the basic types *int, real, bool* and *String*. These are the predefined basic types of OCL. All type domains include an undefined value that allows to operate with unknown or null values. Array types are introduced to describe vectors of any kind of elements, together with corresponding operations. All the types and operations are defined as in OCL.

DataType: is the root datatype of PAL and represents anything. Therefore, its methods apply to all primitive types, array type and record type. It is defined to allow defining generic operations that can be invoked by any object or simple value. It is similar to *AnyType* defined in OCL, but we have preffered this approach since *AnyType* is not compliant with all types in OCL, namely Collection types and implicitely its descendants. Defining *DataType* and its operation *new* we can create uniformly any new value as a reference to its starting address.

Operations on the type:

isType(element : DataType) : Boolean Checks if the argument is of the specified type,

new() :DataType Creates a new instance of the type.

Operations on instances of the type:

isTypeOf(type) : Boolean Checks if the instance is of the specified type

isDefined() : Boolean Checks if the instance is defined (not null).

Array Type inherits from *DataType*.

Operations:

size() : Integer Returns the size of the array *isEmpty() : Boolean* Checks if the array has no items.

Operation *[]* takes an integer i as argument and returns the i-th element of the array.

The operations regarding the variable declarations are implemented in the *DataType*. In such a way, we may create new instances of array type, and we may check if the instance is not null.

A type is assigned to every expression and typing rules determine which expressions are well-formed. There are a number of predefined OCL types and operations available for use with any UML model, which we considered as given. For the newly introduced type constructions and its associated operations we will give typing rules. The semantics of types in T and operations in Ω is defined by a mapping that assigns each type a domain and each operation a function.

The following rule states that we may create arrays of any existing type in the system:

$$\frac{G|-A:T}{G|-Array(A):T}$$

An array M is defined with elements of a type and an integer as index:

$$\frac{G|-N:Int, G|-M:A}{G|-array(M,N):Array(A)}$$

If i is an index of an array then i is of type $Integer$.

$$\frac{G|-M:Array(A)}{G|-indexM:Integer}$$

The following rule specifies the way we can infer the type of an element of an array knowing the type of the array:

$$\frac{G|-N:Int, G|-M:Array(A)}{G|-M[N]:A}$$

The last rule states the constraints imposed on assignment to an element of an array:

$$\frac{G|-N:Int, G|-M:Array(A), G|-P:A}{G|-M[N]:=P:array(A)}$$

In order to illustrate our workbench support for defining and executing platform-independent components we consider a simple case study that prints a given product catalog. The class diagram presented in Figure 3 shows an extract of an executable UML model developed using COMDEVALCO Workbench [7]. The *Product* entity represents descriptive information about products and the *ProductCatalog* interface have operations that can be used to obtain product descriptions as well as the product prices. The *CatalogPrinter* component is designed to print the catalog, so it requires a reference to a *ProductCatalog*. The model contains a *SimpleProductCatalog* implementation that has an array of products and an array of *prices*.

The model defined in Figure 3 uses the stereotypes defined by the iCOMPONENT UML profile for dynamic execution environments [4]. According to the iCOMPONENT component model, these model elements can be deployed

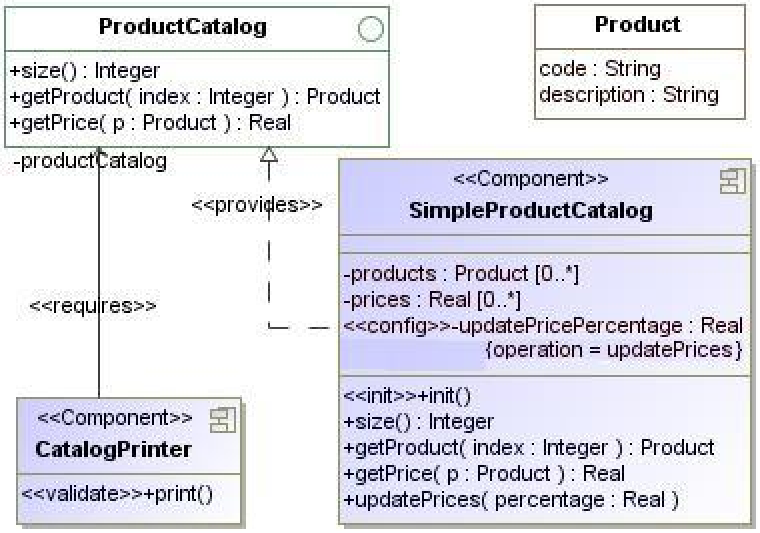

ProductCatalog
+size() : Integer
+getProduct(index : Integer) : Product
+getPrice(p : Product) : Real
Product
code : String
description : String
-productCatalog
<<provides>>
<<requires>>
<<Component>>
SimpleProductCatalog
-products : Product [0..*]
-prices : Real [0..*]
<<config>>-updatePricePercentage : Real {operation = updatePrices}
<<init>>+init()
+size() : Integer
+getProduct(index : Integer) : Product
+getPrice(p : Product) : Real
+updatePrices(percentage : Real)
<<Component>>
CatalogPrinter
<<validate>>+print()

FIGURE 3. Executable iCOMPONENTS

as three modules (units of deployments): a module that contains the *Product* class and the *ProductCatalog* interface, another one that contains the *CatalogPrinter* component, and finally a module containing the *SimpleProductCatalog* component. After deployment, the dynamic execution environment applies the dependency inversion principle in order to inject the *ProductCatalog* reference required by the *CatalogPrinter* component.

Using the *validate* stereotype the *CatalogPrinter* component register the *print* method as a callback method that is executed when the component acquire the required interface. The execution starts with this method.

The *init* method of the *SimpleProductCatalog* component is executed immediately after an instance of the component is created. The *updatePricePercentage* property is a configuration property that specifies that the *updatePrices* operation will be executed when a running component instance is reconfigured.

Figure 4 and 5 show the code written using the proposed extended OCL-based action language.

5. CONCLUSION AND FUTURE WORK

We have presented an Action Language based on procedural paradigm and defined an extension with arrays, that can be succesfully used in specifying executable UML components. The approach taken in extending the PAL,

```
operation print() {
  Integer productCount = productCatalog.size();
  for(int index = 0; index < productCount; index++) {
    Product product = productCatalog.getProduct(index);
    write(product.code);
    write(product.description);
    write(productCatalog.getPrice(product));
  }
}
```

Figure 4. *CatalogPrinter* operation

```
operation size(): Integer {
  return products.size();
}
operation getProduct(index: Integer): Product {
  assert (0 <= index) and (index < products.size());
  return products[index];
}
operation getPrice(product: Product): Real {
  return prices[product.code];
}
operation init() {
  products = new Product[] {
    new Product(0, "A"), new Product(1, "B")
  };
  prices = new Real[] {5, 7};
}
operation updatePrices(percentage: Real) {
  for(int index = 0; index < prices.size(); index++)
    prices[index] = (1 + percentage) * prices[index];
}
```

Figure 5. *SimpleProductCatalog* operation

can be used in adding new features to it, and integrating them in the framework. The main application of such specifications is to completely describe executable components for storing in a repository, as suggested in [4].

As future developments we intend to add, when necessary, further extensions to the PAL and integrate them in ComDeValCo workbench, and to use information from these specifications to validate the components.

6. ACKNOWLEDGEMENTS

This work was supported by the grant ID_546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

7. References

[1] S. Haustein and J. Pleumann. *OCL as Expression Language in an Action Semantics Surface Language.* OCL and Model Driven Engineering, UML 2004 Conference Workshop, 2004.

[2] K. Jiang, L. Zhang, and S. Miyake. *OCL4X: An Action Semantics Language for UML Model Execution.* Proc. of COMPSAC, pages 633-636, 2007.

[3] I. Lazar, B. Parv, S. Motogna, I. Czibula, and C. Lazar. *An Agile MDA Approach for Executable UML Structured Activities.* Studia Univ. Babes-Bolyai Informatica, 2:101-114, 2007.

[4] I. Lazar, B. Parv, S. Motogna, I.-G. Czibula, and C.-L. Lazar. *iCOMPONENT: A platform-independent component model for dynamic execution environments.* In 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing. submitted, 2008.

[5] Object Management Group. *Object Constraint Language Specification, version 2.0.* http://www.omg.org/cgibin/ apps/doc?formal/06-05-01.pdf, 2006.

[6] Object Management Group. *UML 2.1.1 Superstructure Specification.* http://www.omg.org/cgi-bin/doc?ptc/07-02-03/, 2007.

[7] B. Parv, I. Lazar, and S. Motogna. *COMDEVALCO Framework - the Modeling Language for Procedural Paradigm.* Int. J. of Computers, Communications and Control, 3(2):183- 195, 2008.

[8] C. Raistrick, P. Francis, J. Wright, C. Carter, and I. Wilkie. *Model Driven Architecture with Executable UMl.* Cambridge University Press, 2004.

[9] J. Steel and J.-M. Jezequel. *On model typing.* International Journal of Software and System Modeling (SoSyM), 2008.

8. Appendix A - PAL grammar

```
prog: program (operation)* | (operation)+
program: PROGRAM ID pre_post_conditions statement_block
operation : OPERATION ID operation_parameter_list
    (':' TYPE)? pre_post_conditions statement_block
operation_parameter_list : aux='(' (operation_parameter)?
    (','operation_parameter)* ')'
operation_parameter : (PARAM_TYPE)? aux=ID ':' TYPE
pre_post_condition:(precondstatement)?(postcondstatement)?
statement_block : startN='{' statement* endN='}'
statement: asignstatementstandalone | callstatement | ifstatement |
    declstatement | whilestatement | forstatement | assertstatement |
     readstatement | writestatement | returnstatement
callstatement : CALL expr endN=';'
```

```
readstatement : READ ID endN=';'
writestatement : WRITE expr endN=';'
asignstatement : ID ASSIGN expr
asignstatementstandalone : ID ASSIGN expr endN=';'
declstatement :VARDECLR ID':'TYPE(':='expr)?endN=';'
ifstatement : IF '(' expr ')' b1=statement_block
    ( ELSE b2=statement_block )?
whilestatement : WHILE '(' expr ')' loop_statement_block
forstatement : FOR '(' e1=asignstatement ';' e2=expr ';'
    e3=asignstatement ')' loop_statement_block
assertstatement : ASSERT ':' expr endN=';'
returnstatement : RETURN expr? endN=';'
precondstatement : PRECOND ':' oclexpr endN=';'
postcondstatement : POSTCOND ('(' ID ')')? ':' oclexpr endN=';'
loopinvariant : LOOPInv ':' expr endN=';'
loopvariant : LOOPVa ':' expr endN=';'
loop_statement_block : startN='{' (loopinvariant)?
    (loopvariant)? statement* endN='}'
condition : '(' expr ')'
oclexpr : expr
expr: sumexpr
sumexpr : (conditionalExpr ) (OP_PRI0 =conditionalExpr)
conditionalExpr : (multExpr ) (OP_PRI1 e=multExpr )
multExpr : (atom) (OP_PRI2 e=atom )*
atom: ID | INT |STRINGLITERAL | ID '(' (expr)? (',' expr)* endN=')'|
     e=TRUE | e=FALSE | condition
PARAM_TYPE: 'in'| 'out' | 'inout'
TYPE : 'Integer' |'Boolean' | 'String' | 'Real'| DataType |
     TYPE[DOMAIN ]
DOMAIN : INT..INT | INT.. *
OP_PRI0 :('and' | 'or' | 'not' |′ <′ |′ >′ |′ <=′ |′ >=′ |
    ′ ==′ |′ <>′)
OP_PRI1 : ('+' |'-' )
OP_PRI2 : ('*' |'/' | 'div')
ID : ('a'..'z' |'A'..'Z' ) ('a'..'z'|'A'..'Z' | '0'..'9' )*
INT : '0'..'9' +
STRINGLITERAL : ' " ' ( options {greedy=false;} : . )* ' " '
BOOLEAN_CONST : 'true' | 'false'
```

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu, Cluj-Napoca 400084, Romania

E-mail address: bparv,motogna,ilazar,czibula@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

SOFTWARE QUALITY ASSESSMENT USING A FUZZY CLUSTERING APPROACH

CAMELIA SERBAN AND HORIA F. POP

Abstract. Metrics have long been studied as a way to assess the quality and complexity of software, and recently this has been applied to object-oriented software as well. However one of their shortcomings is the lack of relevant result interpretation. Related to this, there is an aspect that has a decisive influence on the accuracy of the results obtained: the issue of software metrics threshold values.

In this paper we propose an alternative approach based on fuzzy clustering analysis for the problem of setting up the software metrics threshold values. Measurements are used to evaluate the conformance of an object oriented model to well established design heuristics.

1. Introduction

In time, software systems become very large and complex due to repeated modifications and updates, needed to meet the ever changing requirements of the business. The code becomes more complex and drifts away from its original design. The result is that the system becomes practically unmanageable. A small change in one part of it may have unforeseen effects in completely other parts, leading to potential disasters. In order to prevent this, we need proper quantification means in order to assess the quality of software design during its development lifecycle.

A good object-oriented design needs design rules, principles and practices that must be known and used [11]. In this way, software metrics are very useful being a mean for quantifying these aspects and identifying those design entities that capture deviations from good design principles and heuristics.

Although a large number of metrics have been proposed by researchers to assess object-oriented design quality, they pose some problems of their own, the most important being the ability to give relevant interpretation of the

Received by the editors: October 10, 2008.

2000 *Mathematics Subject Classification.* 68N19, 68T37.

1998 *CR Categories and Descriptors.* D.2.8 [**Software Engineering**]: Metrics – *Performance measures*; I.5.3 [**Pattern recognition**]: Clustering – *Algorithms*.

Key words and phrases. Software quality, Software metrics, Fuzzy clustering.

measurement results which in turn is due to the fact that threshold values for the metrics are difficult to set. This problem is far from being new and characterizes intrinsically any metrics-based approach. A threshold divides the space of a metric value into regions. Depending on the region of the metric value, we may make an informed assessment about the measured entity. For example, if we measure the reusability of a design entity with possible values in the [0..1] range and we define 0.7 as being the threshold with good reusability, then all measured components whose reusability values are above that threshold may be quantified as being reusable. This simple example raises a set of questions: how did we come up with a threshold of 0.7 in the first place? Why not 0.5? And, is a component with a reusability value of 0.68 not reusable compared to a component having a reusability value of 0.7? Would such a threshold still be meaningful in a population where the largest reusability value is 0.5?

As a conclusion, the accuracy of the results obtained is questionable. In order to overcome this limitation, we propose an alternative approach for the problem of setting up the software metrics threshold values using fuzzy clustering analysis. This allows us to place an object in more than one group, with different membership degrees.

The remainder of this paper is organized as follows. Section 2 describes the theoretical background for an object-oriented design quality assessment system while Section 3 presents the fuzzy-clustering approach used in the quality evaluation of a system design. Section 4 describes in details our proposed approach for detecting design flaws in an object-oriented system. Section 6 presents and discusses the experimental results obtained by applying the proposed approach on an open source application, called *log4net* [3]. Section 7 reviews related works in the area of detection design flaws. Finally, Section 8 summarizes the contributions of this work and outlines directions for further research.

2. Theoretical framework

Object oriented design quality evaluation implies identification of those design entities that are relevant for the analysis of their properties and of the relationships that exist between them and the software metrics that best emphasize the aspects (design principle/heuristics) that we want to quantify. So, our theoretical framework, consists of three groups of elements:

- a meta-model for the object-oriented systems;
- design principles/heuristics;
- relevant suites of software metrics.

Thus, our object oriented design quality evaluation system will be associated with a 3-tuple, $ES = (MModel, Aspects, Metrics)$. In what follows, all the above mentioned elements will be briefly described.

2.1. **A meta-model for object-oriented systems.** A meta-model for object-oriented systems consists of design entities together with their properties and the relations between them [11]. Thus, a meta-model is a 3-tuple $MModel = (E, P, R)$ where,

- $E = \{E_1, E_2, ..., E_n\}$, represents the set of *design entities* of the software system, E_i, $1 \leq i \leq n$ may be a class, a method from a class, an attribute from a class, a parameter from a method or a local variable declared in the implementation of a method. We also will consider that:

- $Class(E) = \{C_1, C_2, ..., C_l\}$, $Class\,(E) \subset E$ is a set of entities that are classes;
- Each class, C_i, $1 \leq i \leq l$ has a set of methods and attributes, i.e. $C_i = \{m_{i1}, m_{i2}, ..., m_{ip_i}, a_{i1}, a_{i2}, ..., a_{ir_i}\}$, $1 \leq p_i \leq n$, $1 \leq r_i \leq n$, where $m_{ij} (\forall j, 1 \leq j \leq p_i)$ are methods and a_{ik} $(\forall k, 1 \leq j \leq r_i)$ are attributes from C_i;
- $Meth(E) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} m_{ij}, Meth(E) \subset E$, is a set of methods from all classes of the software system;
- Each method m_{ij}, $1 \leq i \leq l$, $1 \leq j \leq p_i$, has a set of parameters and local variables, i.e., $m_{ij} = \{p_{ij1}, p_{ij2}, ..., p_{ijp_{ij}}, v_{ij1}, v_{ij2}, ..., v_{ijv_{ij}}\}$ $1 \leq p_{ij} \leq n$, $\quad 1 \leq v_{ij} \leq n$, where $p_{ijk}(\forall k,\ 1 \leq k \leq p_{ij})$ are parameters and $v_{ijs}(\forall s,\ 1 \leq s \leq v_{ij})$ are local variables;
- $Param(E) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} \bigcup\limits_{k=1}^{p_{ij}} p_{ijk}, Param(E) \subset E$;
- $LocVar(E) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} \bigcup\limits_{s=1}^{v_{ij}} v_{ijs}, LocVar(E) \subset E$;
- $Attr(E) = \bigcup\limits_{i=1}^{l} \bigcup\limits_{j=1}^{r_i} a_{ij}, Attr(E) \subset E$, is the set of attributes from all classes of the software system.

- P represents the set of *properties* of the aforementioned design entities, $P = ClassP \bigcup MethP \bigcup\ AttrP \bigcup ParamP \bigcup LocVarP$. Where,

- $ClassP$ represents the properties of all classes in E (e.g. abstraction, visibility, reusability);
- $MethP$ represents the properties of all methods in E (e.g. visibility, kind, instantiation, reuse, abstraction, binding);
- $AttrP$ represents the properties of all attributes in E (e.g. visibility);

- $ParamP$ represents the properties of all parameters in E (e.g. type, aggregation);
- $LocVarP$ represents the properties of all local variables in E (e.g. type, aggregation);

- R represents the set of *relations* between the entities of the set E. These relations are described in detail in [11].

2.2. **Design principles and heuristics.** The main purpose of our evaluation is to identify those design entities that capture deviations from good design principles and heuristics. Object-oriented design principles are mostly extensions of general design principles in software systems (*e.g.*, abstraction, modularity, information hiding). Samples of principles for good design in software systems are: *high coupling*, *low cohesion*, *manageable complexity*, *proper data abstraction*. Design heuristics [17] are stated as the rules of thumb or guidelines for good design. These rules are based on design principles and their ultimate goal is to improve quality factors of the system and avoid occurrence of design flaws. These rules recommend designers and developers to "do" or "do not" specific actions or designs. A sample of such heuristics is "minimize the number of messages in a class".

A literature survey showed a constant and important preoccupation for this issue: several authors were concerned with identifying and formulating design principles [14, 12] and heuristics [17, 9]. Riel [17] presents a set of heuristic design guidelines and discusses some of the flawed structures that result if these guidelines are violated. In the recent years, we found various forms of descriptions for bad or flawed design in the literature such as *bad-smells* [7]. In the same manner, Martin [12] discusses the main design principles of object-orientation and shows that their violation leads to a *rotting design*.

2.3. **A catalog of design metrics.** As we mentioned earlier, the quantification of object-oriented design principle needs a relevant metrics catalog. Thus, the third element of the proposed framework is the set of design metrics. These metrics have to be selected based on the definitions and classification rules of each design principle/heuristics. We do not intend to offer an exhaustive list of design metrics in this section, but to emphasize their relevance in quantifying some rules related to good object oriented design.

Thus, in the following we make a short survey of the most important object-oriented metrics defined in the literature. These metrics capture characteristics that are essential to object-orientation including *coupling, complexity and cohesion.*

Coupling Metrics. We selected Coupling Between Objects(CBO) [6] as the primitive metric for coupling. CBO provides the number of classes to which a given class is coupled by using their member functions and/or instance variables. Other metrics related with CBO are Fan - Out [19], Data Abstraction Coupling(DAC) [1] and Access To Foreign Data(ATFD) [11]. A second way of measuring coupling is: when two classes collaborate, count the number of distinct services accessed (the number of distinct remote methods invoked). One measure that counts the number of remote methods is RFC (Response For A Class)[6]. Another important aspect that has to be taken into account when measuring coupling is the access of a remote method from different parts of the client class, each access being counted once. This is the approach taken by Li and Henry in defining the Message Passing Coupling(MPC) metric, which is the number of send statements defined in a class [1] (also proposed in [10]). A similar type of definition is used by Rajaraman and Lyu in defining coupling at the method level. Their method coupling MC measure [16] is defined as the number of non-local references in a method.

Cohesion Metric. LCOM (Lack of Cohesion in Methods) [6] is not a significant cohesion indicator as discussed in [8, 5]. In [5] the authors propose two cohesion measures that are sensitive to small changes in order to evaluate the relationship between cohesion and reuse. The two measures are TCC (Tight Class Cohesion) and LCC (Loose Class Cohesion) TCC is defined as the relative number of directly connected methods. Two methods are directly connected if they access a common instance variable of the class. TCC refers the relative number of directly connected methods in a given class. LCC is the relative number of directly or indirectly connected methods. Two methods are considered to be indirectly connected if they access a common instance variable through the invocation of other methods.

Complexity Metric. In order to measure the structural complexity for a class, instead of counting the number of methods, the complexities of all methods must be added together. This is measured by WMC (Weighted Method per Class) metric [6]. WMC is the sum of the complexity of all methods for a class, where each method is weighted by its cyclomatic complexity. The number of methods and the complexity of the methods involved is a predictor of how much time and effort is required to develop and maintain the class.

Several studies have been conducted to validate these metrics and have shown that they are useful quality indicators [20].

After computing the metrics values, the next step is to give a relevant interpretation of the obtained measurements results. Following a classical approach we have to set threshholds values for metrics that we use. As we mentioned before, the problem of setting up the threshholds is not simple and the accuracy of the results obtained is questionable. In order to overcome this limitation, we propose an alternative approach based on fuzzy clustering analysis for the problem of setting up the software metrics threshold values. Thus, an object may be placed in more that one group, having different membership degree.

3. Fuzzy clustering analysis

Clustering is the division of data set into subsets (clusters) such that, similar objects belong to the same cluster and dissimilar objects to different clusters. Many concepts found in real world do not have a precise membership criterion, and thus there is no obvious boundary between clusters. In this case fuzzy clustering is often better, as objects belong to more that one cluster with different membership degrees.

Fuzzy clustering algorithms are based on the notion of fuzzy set that was introduced in 1965 by Lotfi A. Zadeh [21] as a natural generalization of the classical set concept. Let X be a data set composed of n data items. A fuzzy set on X is a mapping $A : X \to [0,1]$. The value $A(x)$ represents the membership degree of the data item $x \in X$ to the class A. Fuzzy clustering algorithms partition the data set into overlapping groups based on similarity amongst patterns.

3.1. **Fuzzy Clustering Analysis – formalization.**

Let $X = \{O_1, O_2, ..., O_n\}$ be the set of n objects to be clustered. Using the vector space model, each object is measured with respect to a set of m initial attributes $A_1, A_2, ..., A_m$ (a set of relevant characteristics of the analyzed objects) and is therefore described by a m-dimensional vector $O_i = (O_{i1}, O_{i2}, ..., O_{im})$, $O_{ik} \in \Re$, $1 \le i \le n$; $1 \le k \le m$;

Our aim is to find a fuzzy partition matrix $U = (C_1, C_2, ..., C_c)$, $C_i = (u_{i1}, u_{i2}, ..., u_{in})$, $1 \le i \le c$, that best represents the cluster substructure of the data set X., i.e. objects of the same class should be as similar as possible, and objects of different classes should be as dissimilar as possible. The fuzzy partition matrix, U has to satify the following constraints:

- *membership degree:* $u_{ik} \in [0..1]$, $1 \le i \le c$, $1 \le k \le n$, u_{ik} represents the membership degree of the data object O_k to cluster i;
- *total membership:* the sum of each column of U is constrained to the value $1(\sum\limits_{i=1}^{c} u_{ik} = 1)$.

The fuzzy clustering generic algorithm, named Fuzzy c-means clustering, is described in [4]. This algorithm has the drawback that the optimal number of classes corresponding to the cluster substructure of the data set, is a data entry. As a result in this direction, hierarchical clustering algorithms, produce not only the optimal number of classes (based on the needed granularity), but also a binary hierarchy that show the existing relationships between the classes. In this paper we use the Fuzzy Divisive Hierarchic Clustering algorithm (FDHC) [22].

4. Our Approach

The main objective of this paper is to use fuzzy clustering technique in order to offer an alternative solution to the problem of setting up the software metrics thresholds values, metrics applied for object-oriented design quality investigation. In other words, we aim at identification of those design entities that violate a specified design principle, heuristics or rule. These entities are affected by some design flaw. Thus, our problem can be reduced at identification of those design flaws that violate a specified design principle or heuristic. In fact, design flaws are violations of these heuristics/principles.

Let us consider the theoretical framework proposed in Section 3. In addition, we adopt the following notations:

- DP denotes the set of design principles, heuristics or rules that we want to quantify;
- DF denotes the set of design flaws that violate the entitites from DP;
- $R \subseteq DP \times DF$, the associations set between DP and DF;

Definition 1. *The 3-tuple $GPF = (DP, DF, R)$ is a bipartite graph, called principles-design flaws.*

For each element from the DP or DF set we have to identify a set of relevant metrics. The set of all these metrics will be denoted by M. Let also consider R_1 to be the set of associations between the entities from DP and their corresponding metrics from M and R_2 to be the set of associations between the entities from DF and their coresponding metrics from M.

Definition 2. *The 3-tuple $GPM = (DP, M, R_1)$ is a bipartite graph, called principle metrics.*

Definition 3. *The 3-tuple $GFM = (DF, M, R_2)$ is a bipartite graph, called flaw metrics.*

With these considerations our problem stated in Section 2 can be rephrased as follows: given an element, p, from DP or DF set, its associated metrics set M_p and a subset of design entities from E, we have to identify (using a fuzzy clustering approach) those design entities that capture deviations from a specified principle/heuristic or are affected by a specified design flaw. In this way, for each entity implied in the evaluation, we obtain a set of metrics values.

We may apply now the FDHC algorithm referred in Section 3. The design entities implied in the evaluation correspond to objects from the fuzzy clustering algorithm and the metrics values to the attributes of these objects.

After applying this algorithm each assessed entity is placed into a cluster having a membership degree. This approach offers a better interpretation of measurements results than the thresholds values-based interpretation.

5. Case study

In order to validate our approach we have used the following case study. The object oriented system proposed for evaluation is *log4net* [3], an open source application. It consists of 214 classes. The elements of the meta-model defined in Section 2.1 (design entities, their properties and the relations between them) ware identified using our own dveloped tool.

The objective of this case-study is to identify those entities affected by "God Class" [17] design flaws. So, the objects considered for fuzzy clustering algorithm are classes from the analyzed system.

The first step in this evaluation is to construct (from the graph principles-design flaws defined in Section 5) the subgraph that contains the node "God Class" and its related "heuristics/rules". As it is known, an instance of a god-class performs most of the work, delegating only minor details to a set of trivial classes and using the data from other classes. This has a negative impact on the reusability and the understandability of that part of the system. This design problem may be partially assimilated with Fowlers Large Class bad-smell. In this case we will start from a set of two heuristics found in Riels book [17]:

- Distribute system intelligence horizontally as uniformly as possible;
- Beware of classes with much non-communicative behavior.

The second step is to select proper metrics that best quantify each of the identified heuristics/rules. This means identifying the subgraph obtained by keeping the nodes corresponding with these heuristics and their corresponding metrics that we want to take into account.

In our case the first rule refers to a uniform distribution of intelligence among classes, and thus it refers to *high class complexity.* The second rule speaks about the level of intraclass communication; thus it refers to the *low cohesion* of classes. Therefore, we chose the following metrics:

- Weighted Method per Class (WMC) is the sum of the statical complexity of all methods in a class [6]. We considered the McCabes cyclomatic complexity as a complexity measure [13].
- Tight Class Cohesion (TCC) is the relative number of directly connected methods [5].
- Access to Foreign Data (ATFD) represents the number of external classes from which a given class accesses attributes, directly or via accessor-methods [11]. The higher the ATFD value for a class, the

higher the probability that the class is or is about to become a god-class.

As a remark, a possible suspect of “God Class” will have high values for the WMC and ATFD metrics and low values for the TCC metric.

Taking into account the metrics mentioned above each class from our system, c_i, will be identified by a vector of three elements, $c_i = (m_1, m_2, m_3)$, corresponding to the metrics values applied for class c_i.

The next step is to apply the FDHC algorithm described in Section 3. The objects from the algorithm are classes from our system and the features are the computed values of the metrics corresponding to these classes. The classification tree and the final binary partition produced by FDHC algorithm are represented in Figure 1. By interpreting the results obtained we may conclude that the algorithm has identified a list of suspects, those from class 1 and a list of the objects that do not need further investigation, class 2 of objects. The list of suspects from class 1 are further partitioned according to the values of the three metrics. For example, in class 1.1.1.1.1.1. the list of suspects have the value 0 of the TCC and ATFD metrics and low value for the WMC metric.

Due to space restrictions, we include in this paper only a subset of objects, containing a list of suspects. These objects are described in Figure 2. All other numerical data are available from the authors by request.

6. Related work

During the past years, various approaches have been developed to address the problem of detecting and correcting design flaws in an object-oriented software system using metrics. Marinescu [11] defined a list of metric-based detection strategies for capturing around ten flaws of object-oriented design at method, class and subsystem levels as well as patterns. However, how to choose proper threshold values for metrics and propose design alternatives to correct the detected flaws are not addressed in his research.

Mihancea et al. [15] presented an approach to establish proper threshold values for metrics-based design flaw detection mechanism. This approach, called tuning machine, is based on inferring the threshold values based on a set of reference examples, manually classified in flawed, respectively good design entities.

Trifu [18] introduced correction strategies based on the existing flaw detection and transformation techniques. This approach serves as reference descriptions that enable a human-assisted tool to plan and perform all necessary steps for the removal of detected flaws. Consequently, it is a methodology that can be fully supported.

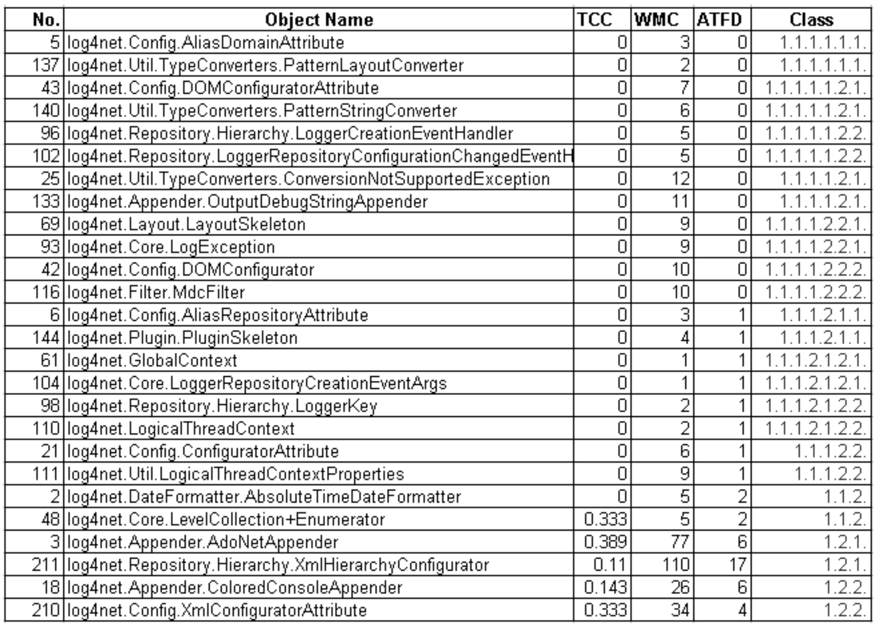
Class	Members
1.1.1.1.1.1.	1 5 15 24 35 46 137 155 183 198
1.1.1.1.1.2.1.	43 140 151
1.1.1.1.1.2.2.	41 96 102 103 105 106 207
1.1.1.1.2.1.	25 97 133 173
1.1.1.1.2.2.1.	69 86 91 93 152
1.1.1.1.2.2.2.	42 52 116 124
1.1.1.2.1.1.	6 26 27 57 71 83 84 115 129 144 166 170 199
1.1.1.2.1.2.1.	61 95 104 108
1.1.1.2.1.2.2.	98 110 191
1.1.1.2.2.	21 44 58 89 111 122 128 164 171
1.1.2.	2 14 28 45 47 48 49 66 88 113 120 121 123 136 146 160 161 162 178 187 192 193 194 197 201 206 208 213
1.2.1.	3 11 12 16 38 55 63 72 94 99 107 109 112 138 142 172 179 186 209 211
1.2.2.	18 19 37 77 87 114 1 26 132 134 135 139 163 167 168 169 180 190 210
2.	4 7 8 9 10 13 17 20 22 23 29 30 31 32 33 4 36 39 40 50 51 53 54 56 59 60 62 64 65 67 68 70 73 74 75 76 78 79 80 81 82 85 90 92 100 101 117 118 119 125 127 130 131 141 143 145 147 148 149 150 153 154 156 157 158 159 165 174 175 176 177 181 182 184 185 188 189 195 196 200 202 203 204 205 212 214

Figure 1. Classification tree and final partition for the set of 214 objects

M. Frenţiu and H.F.Pop [2] presented an approach based on fuzzy clustering to study dependencies between software attributes, using the projects written by second year students as a requirement in their curriculum. They have observed that there is a strong dependency between almost all considered attributes.

7. Conclusions and Future Work

We have presented in this paper a new approach that address the issue of setting up the software metrics threshold values, approach based on fuzzy clustering techniques. In order to validate our approach we have used a case study, presented in Section 5. Further work can be done in the following directions:



No.	Object Name	TCC	WMC	ATFD	Class
5	log4net.Config.AliasDomainAttribute	0	3	0	1.1.1.1.1.1.
137	log4net.Util.TypeConverters.PatternLayoutConverter	0	2	0	1.1.1.1.1.1.
43	log4net.Config.DOMConfiguratorAttribute	0	7	0	1.1.1.1.1.2.1.
140	log4net.Util.TypeConverters.PatternStringConverter	0	6	0	1.1.1.1.1.2.1.
96	log4net.Repository.Hierarchy.LoggerCreationEventHandler	0	5	0	1.1.1.1.1.2.2.
102	log4net.Repository.LoggerRepositoryConfigurationChangedEventH	0	5	0	1.1.1.1.1.2.2.
25	log4net.Util.TypeConverters.ConversionNotSupportedException	0	12	0	1.1.1.1.2.1.
133	log4net.Appender.OutputDebugStringAppender	0	11	0	1.1.1.1.2.1.
69	log4net.Layout.LayoutSkeleton	0	9	0	1.1.1.1.2.2.1.
93	log4net.Core.LogException	0	9	0	1.1.1.1.2.2.1.
42	log4net.Config.DOMConfigurator	0	10	0	1.1.1.1.2.2.2.
116	log4net.Filter.MdcFilter	0	10	0	1.1.1.1.2.2.2.
6	log4net.Config.AliasRepositoryAttribute	0	3	1	1.1.1.2.1.1.
144	log4net.Plugin.PluginSkeleton	0	4	1	1.1.1.2.1.1.
61	log4net.GlobalContext	0	1	1	1.1.1.2.1.2.1.
104	log4net.Core.LoggerRepositoryCreationEventArgs	0	1	1	1.1.1.2.1.2.1.
98	log4net.Repository.Hierarchy.LoggerKey	0	2	1	1.1.1.2.1.2.2.
110	log4net.LogicalThreadContext	0	2	1	1.1.1.2.1.2.2.
21	log4net.Config.ConfiguratorAttribute	0	6	1	1.1.1.2.2.
111	log4net.Util.LogicalThreadContextProperties	0	9	1	1.1.1.2.2.
2	log4net.DateFormatter.AbsoluteTimeDateFormatter	0	5	2	1.1.2.
48	log4net.Core.LevelCollection+Enumerator	0.333	5	2	1.1.2.
3	log4net.Appender.AdoNetAppender	0.389	77	6	1.2.1.
211	log4net.Repository.Hierarchy.XmlHierarchyConfigurator	0.11	110	17	1.2.1.
18	log4net.Appender.ColoredConsoleAppender	0.143	26	6	1.2.2.
210	log4net.Config.XmlConfiguratorAttribute	0.333	34	4	1.2.2.

Figure 2. A list of "God Class" design flaw suspects

(1) To apply this approach for more case studies;
(2) Comparison with others approaches regarding the issue of threshold values;
(3) To develop a tool that emphasizes the approach presented in this paper.

8. Acknowledgement

This research has been supported by CNCSIS - the Romanian National University Research Council, through the PNII-IDEI research grant ID_550/2007.

References

[1] W. Li and S. Henry. Maintenance Metrics for the Object Oriented Paradigm. IEEE Proc. First International Software Metrics Symp., pages 5260, may 1993.

[2] M. Frentiu and H.F. Pop. A study of dependence of software attributes using data analisys techniques. Studia Univ. Babes-Bolyai, Series Informatica, 2 (2002), 53–66.

[3] Project log4net.: http://logging.apache.org/log4net.

[4] Bezdek, J.: Pattern Recognition with Fuzzy Objective Function Algorithms. Plenum Press, New York, 1981.

[5] Bieman, J. and Kang, B.: Cohesion and reuse in an object-oriented system. Proc. ACM Symposium on Software Reusability, apr (1995).

[6] Chidamber, S. and Kemerer, C.: A metric suite for object- oriented design. IEEE Transactions on Software Engineering, 20(6):476–493, June (1994).

[7] Fowler, M. Beck, K. Brant, J. Opdyke, W. and Roberts, D.: Refactoring: Improving the Design of Existing Code. Addison-Wesley, (1999).

[8] Henderson-Sellers, B.: Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, (1996).

[9] Johnson, R. and Foote, B.: Designing reuseable classes. Journal of Object-Oriented Programming, 1(2):22–35, June (1988).

[10] Lorenz, M. and Kidd, J.: Object-Oriented Software Metrics. Prentice-Hall Object-Oriented Series, Englewood Cliffs, NY, (1994).

[11] R. Marinescu, Measurement and quality in object-oriented design. Ph.D. thesis in the Faculty of Automatics and Computer Science of the Politehnica University of Timisoara, 2003.

[12] R. Martin, Design Principles and Patterns. Object Mentor,http://www.objectmentor.com, 2000

[13] McCabe, T.: A complexity measure. IEEE Transactions on Software Engineering, 2(4):308–320, dec (1976).

[14] Meyer, B.: Object-Oriented Software Construction. International Series in Computer Science. Prentice Hall, Englewood Cliffs, (1988).

[15] P.F. Mihancea and R.Marinescu. Towards the optimization of automatic detection of design flaws in object-oriented software systems. In Proc. of the 9th European Conf. on Software Maintenance and Reengineering (CSMR), 92–101, (2005).

[16] Rajaraman, C. and Lyu, M.: Some coupling measures for c++ programs. Prentice-Hall Object-Oriented Series, In Proceedings of TOOLS USA92, Prentice-Hall, Englewood Cliffs, NJ, (1992).

[17] Riel, A.J.: Object-Oriented Design Heuristics. Addison-Wesley, (1996).

[18] Tahvildari, L. and Kontogiannis, K.: Improving design quality using meta-pattern transformations : A metric-based approach. Journal of Software Maintenance and Evolution : Research and Practice, 4-5(16):331–361, October (2004).

[19] D.Tegarden and S.Sheetz: Object-oriented system complexity: an integrated model of structure and perceptions. In OOPSLA92 Workshop on Metrics for Object-Oriented Software Development(Washington DC), (1992).

[20] Basili, V., Briand, L., and Melo, W.: A validation of object-oriented design metrics as quality indicators. IEEE Transactions on Software Engineering 22(10), 751-761, (1996).

[21] Zadeh L. A.: Fuzzy sets, Inf. Control, 8, 338–353, (1965).

[22] Dumitrescu, D.: Hierarchical pattern classification, Fuzzy Sets and Systems 28, 145–162, (1988).

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania

E-mail address: `camelia@cs.ubbcluj.ro, hfpop@cs.ubbcluj.ro`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

SECURING DISTRIBUTED .NET APPLICATIONS USING ADVANCED RUNTIME ACCESS CONTROL

KRISZTIÁN PÓCZA, MIHÁLY BICZÓ, ZOLTÁN PORKOLÁB

Abstract. The architecture and integration of distributed applications increased in complexity over the last decades. It was Service Oriented Architecture (SOA) that answered most of the emerging questions by its explicit and contract-based interface definitions for services and autonomous components. The exposed functionality can be used by anyone who has access to the public interface of SOA applications. Due to loose security handling, risks often emerge in SOA applications. Interfaces are usually published to an unnecessarily wide set of clients. Although there are attempts to implement fine-grained access control mechanisms in object-oriented programming languages like Eiffel, C# and Java, these solutions are in-process that means that they cannot cross service contract boundaries in distributed applications. For these, it is of utmost importance to validate the type and the identity of the caller, track the state of the business process and even validate the client itself using simple, declarative syntax. In this paper we present a framework that aims to introduce fine-grained access control mechanisms in the context of distributed .NET applications. We present a semi-formalized description of the framework and also a pilot implementation.

1. Introduction

The complexity of IT systems has been getting increasingly complex ever since the beginning of software development. IT systems and the business processes that they serve span over multiple networks, computers, and programming languages. What makes things even more complicated is that pieces of software serving specific business goals (the steps of business processes) are dynamically changing. As a consequence, a rchitects and developers face system integration issues in a dynamically changing technical and business environment. Until recently, integration of systems has been performed either

Received by the editors: August 5, 2008.

2000 *Mathematics Subject Classification.* 68M14.

1998 *CR Categories and Descriptors.* C.2.4 [**Computer Systems Organization**]: Computer-Communication Networks – *Distributed Systems*;

Key words and phrases. Distributed applications, Security, Runtime access control, .NET.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

manually or using hard-coded modules that were difficult to maintain and failed in a changing environment. Manual integration has been time consuming and prone to errors, while hard coded solutions require knowledge of all connected systems and have to be re-designed and implemented when any of the underlying systems or steps of the business process changes.

It is Service Oriented Architecture (SOA) [1] [10] that answers the most common difficulties of system integration. From the historical point of view, SOA is an evolution of modular programming, so it extends its basic principles. Reuse, granularity, modularity, composability, componentization, and interoperability are common requests for a SOA application as well as for modular object oriented applications.

However, while the elementary building block of an object oriented software is the class, the basic element of a SOA application is typically a much larger component. These larger chunks of functionality are called services, and this is where the name Service Oriented Architecture originates from. Services implement a relatively large set of functionality, and should be as independent of each other as possible. This means that services should have control over the logic they encapsulate and should not call each other directly. Rather, if a more complex behavior is required, they should be composed to more complex composite services. In other words, services should be autonomous and composable.

Services expose their functionality through service contracts. A contract describes the functions that can be invoked, the communication protocols as well as the authentication and authorization schemes. The exposed functionality is usually a public interface that can be called by anyone who is authenticated, aware of the existence of the service and uses the required communication protocol. The keyword is that the exposed functionality is basically *public*, and users have quite limited amount of control over the identity and the nature of a caller.

However, in a realistic scenario it can also happen that the identity of the caller or the set of allowed methods depends on the state of the underlying business process or other available information. This is usually hard to express, and due to the lack of technology support for fine-grained, or higher level access control, it is challenging to implement the above mentioned scenario using standard protocols, programming environments and tools.

In [4] [15] we have implemented a pilot approach to implement Eiffel-like selective feature export in C# 3.0. This solution makes it possible to control access to protected resources (methods of 'public' interfaces) in a declarative way using simple declarative syntax using the concepts of Aspect Oriented Programming [12]. Although the approach works well in everyday application, it cannot be applied in the case of distributed systems.

What makes things even more complicated is that SOA usually integrates systems running on multiple computers and environments, in other words these systems are distributed ones. To successfully implement our solution we have to sacrifice the interoperability property of SOA, which means that our connected applications have to be created using homogeneous communication platforms. The exposed services are required to be aware of some information about clients. Although this requirement is not common for SOA applications, however, other important properties of SOA can remain unchanged (contract based interface specification, autonomous services). Moreover, the security validation attributes can be regarded as part of the contract.

In this paper our aim is to establish a framework that enables users to control access to the members of public interfaces in a SOA-enabled distributed object system [17].

There are several authors who deal with the security of distributed applications and show the importance of the topic [2] [21]. There are techniques which can be used to generate formal proof of an access request satisfying an access-control policy [3].

[6] provides a method for specifying authorization constraints for workflow based systems that include separation of duty constraints (two different tasks must be executed by different users), binding of duty constraints (same user is required to perform multiple tasks) and cardinality constraints (specify the number that certain tasks have to be executed). A custom reference monitor has been also specified that checks the previously mentioned properties of workflows and workflow tasks.

The concepts in [8] are based on the workflow RBAC authorization rules that are represented as a tuple (r, t, execute, p) that states that users in r role can execute task t when an optional predicate p holds true). They create an extension to the WARM methodology that enables to determine workflow access control information from the business process model. [21] presents an approach where the workflow access control model is decoupled from the workflow model that enables them to create a service oriented workflow access control model. Our solution follows a different approach that makes it more compact but harder to configure.

Another way would be to create a DSL that is dedicated to implementing services [5] and extend this language with security concepts.

There are approaches that store and control policy settings using some centralized database [7] or have multiple layers of configuration [18]. We decided to create an application specific solution and have unified configuration methodology.

In Section 2 we present a simple motivating example that draws attention to issues when not using fine-grained access control mechanisms.

In Section 3 we present a semi-formalized approach to solving problems presented through the motivating example.

In Section 4 a possible implementation of the theory will be shown. The chosen environment is the .NET platform, the Workflow Foundation engine (now part of the .NET framework), and the C# programming language.

In the closing section we summarize our results, and present further research areas.

2. Motivating Example

2.1. **Ping-Pong Game.** In order to highlight the problematic parts when accessing fully public SOA interfaces, in this subsection we are going to show a simple motivating example, is a ping-pong game.

The players of the ping-pong game run on different computers, so it is a distributed application. We suppose that the reader knows the rules of the game. The players register themselves at the game manager, who assigns a unique identifier to both players.

The methods of the game are published as an interface. The Game manager class implements this interface and exposes methods of the game to possible clients, primarily players.

A possible object diagram can be seen in Figure 1.

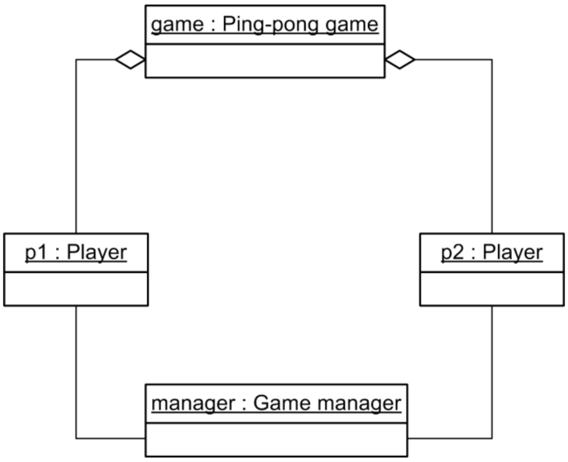


Figure 1. Possible object diagram of a distributed game

The game manager is a singleton, there is exactly one instance of the game manager class.

The 'rules' of the game can be described as a workflow. The workflow itself and its state transitions is a finite state automaton. The finite state automaton can be described as a UML activity diagram [19]. The activity diagram can be seen in Figure 2.

The objects may possibly be hosted on different computers. The difficulty is that we want to allow only objects of type Player to call methods of the Ping-pong game object in this distributed environment. What makes things even more complicated is that the ping-pong game has a well-defined sequence of allowed events with a well-defined set of allowed callers, and we have to keep the system consistent based on these rules.

2.2. **Security Shortcomings of Recent Business Applications.** In real world business applications the sequence and branches of business operations that instrument business processes are well defined and bounded. It is also well defined who can execute a business operation in the lifetime of a business process instance.

The business rules clearly define who is allowed to perform different tasks and also define the exact workflow of our ping-pong game in troduced before (even it is not a business application).

Unfortunately, in most real world applications these business rules are not enforced on the server side, they are rather hard coded in the client applications. Moreover, the restricted functions - based on the user role and the current state of the process - are simply hidden on the user interface. At the same time the server is open for any kind of requests, therefore an attacker can compromise the business process.

The reason of the previous can be one of the following:

1. Architects and developers do not care of business security
2. Architects and developers think that a simple firewall (that restricts the access of the server from specific subnets) or some built-in authentication is enough
3. Architects and developers think or decide that it is satisfying to implement business security on the client side
4. There is no time and money to implement adequate security mechanisms
5. It is hard to implement business security in a distributed environment

Of course it is hard or cannot be carried out to change the mind of architects and developers therefore we suggest a solution that makes server-side business security checks easier and faster to implement.

3. Solve Shortcomings

First we have to denote which client and business properties are suggested to be checked and tracked to raise the business process security level:

1. The runtime type of the caller class on the client side (pin g-pong player in the ping-pong game)
2. State of the business process (e.g. The rules of the ping-pong game in our example)

(3) The identity of the client (e.g. Is it the first or the second player in the ping-pong game?)
(4) Validate, verify the client itself (e.g. IP address, subnet or some kind of certification of the client)

All of the previous are static or internal properties from the view of the business process, therefore all of them can be checked using *declarative syntax* (statically burned in) or can be read from a configuration database.

When creating a SOA application we publish a contract (an interface) to clients. The previous properties can be validated contract-wide and can be validated only for particular business operations published by the contract.

It means that the above properties can be validated at method level at every single call. The granularity level of most of the above properties changes from application level to method level. Informally speaking, we introduce a business call level fine-grained " firewall".

In the next subsections we will examine these four properties from the validability point of view.

We identified the need to give a semi-formalized description of our solution. There are two approaches:

(1) Extend existing description languages (like BPEL [20] [11])
(2) Create a new language that only focuses on the problem presented in this article

Because BPEL focuses on the business process rather than security, and uses XML notation, we have chosen the second approach. BPEL and our semi-formal description can be used side-by-side.

A contract (C) can be defined as a triplet of set of methods, restrictions applied to the contract itself and the set of restrictions applied to individual methods published by the contract.

$C = (\{M_1, M_2, \ldots, M_n\}, R_C, \{R_{M_1}, R_{M_2}, \ldots, R_{M_n}\})$

The restrictions applied to the contract itself (R_c) can be formalized using the following triplet:

$R_c = (\{T_{c_1}, T_{c_2}, \ldots, T_{c_q}\}, \{I_{c_1}, I_{c_2}, \ldots, I_{c_w}\}, \{N_{c_1}, N_{c_2}, \ldots, N_{c_e}\})$

Here T_{c_i}s represents a contract-level type restriction for allowed callers (subsection 3.1), I_{c_i}s denotes a contract-level identity restriction for allowed callers (subsection 3.3), while N_{c_i}s defines a contract-level network restriction (subsection 3.4).

Security restrictions applied to a single method (M_i):

$$R_{M_i} =$$
$$(\{T_{M_i,1}, T_{M_i,2}, \ldots, T_{M_i,r_i}\}, \{(S_{M_i,1}, A_{M_i,1}), (S_{M_i,2}, A_{M_i,2}), \ldots, (S_{M_i,t_i}, A_{M_i,t_i})\},$$
$$\{I_{M_i,1}, I_{M_i,2}, \ldots, I_{M_i,y_i}\}, \{N_{M_i,1}, N_{M_i,2}, \ldots, N_{M_i,u_i}\})$$

Here $T_{M_i,i}$s, $I_{M_i,j}$s and $N_{M_i,j}$s are the same as their contract-level pairs, while$S_{M_i,j}, A_{M_i,j}$ pairs describe the allowed state and state transition constraints (subsection 3.2).

3.1. **Distributed Runtime Access Control.** We have stated in one of our previous work about in-process systems [4] that reducing the interface where software components can communicate with each other increases software quality, security and decreases development cost. Compile time or runtime visibility and access control checking that support encapsulation is the key part of modern languages and runtime environments [16]. They enforce responsibility separation, implementation and security policies. Most modern programming languages like C++, C# and Java do not have sophisticated access control mechanisms only introduce a subset or combination of the following access modifiers: public, private, protected, internal, and friend while Eiffel defines sophisticated selective access control called selective export.

The Eiffel programming language [13] allows features (methods) to be exposed to any named class. The default access level of a feature is the public level. However, an export clause can be defined for any feature which explicitly list classes that are allowed to access the underlying feature.

In this paper we suggest a runtime access control extension to distributed environments where only well identified classes are allowed to access particular methods. To achieve this goal, the server side should be extended with the ability to detect the runtime type of the caller (client) using a *declarative solution* that statically predefines the allowed callers at the contract or method level.

Another possibility would be to restrict access for clients based on group membership or roles (like DCOM [9]). In this case different callers in different roles are to be assigned to (domain level) groups and restrict access of published contracts for certain groups. Moreover, restrictions can be enforced at the operation (method) level to achieve more fine-grained security.

In our ping-ping example only players can participate in matches).

3.2. **Business Process Validation.** In [6] it is noted that it may be necessary to impose constraints on who can perform a task given that a prior task has been performed by a particular individual. In this section we feature another approach to solve the problems stated in [6].

As we mentioned before business applications are driven by rules that define the following properties:

(1) Who is allowed to perform specific actions in given states
(2) What is the resulting state of a state transition if a business operation succeeds
(3) What is the resulting state of a state transition if a business operation fails

In most cases, business processes defined by rules are hard-coded into applications, therefore they can be treated as static properties.

As suggested before operations exported on the interface are statically bounded to certain process states in which they can be executed, furthermore often initiate a state transition where the process gets into another well-defined state.

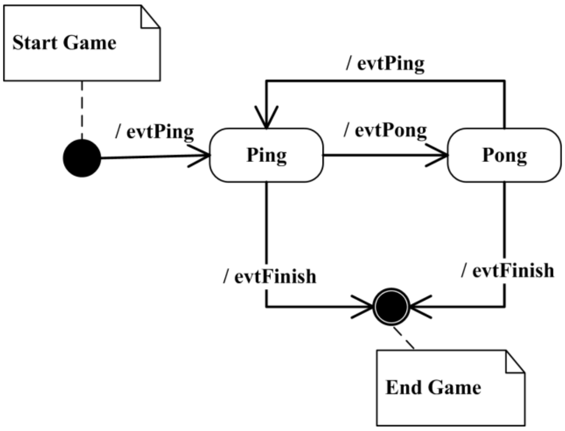


Figure 2. State Machine of the ping-pong game

Business processes can be represented by state machines which are a kind of directed graphs. Vertices of such a graph are the states of the state machine, while edges are the state transitions between states.

The state machine representing the 'business process' behind our p ing-pong game can be described by the following UML Activity diagram in Figure 2 For the sake of simplicity we have not incorporated the error states and events where for example one of the players loses the ball.

The first operation is where the first player gets the ball and hits it (*evtPing*) to the other player therefore the game will be in *Ping* state. After that the second player hits the ball (*evtPong*) to the first player and the game gets into *Pong* state. Now the first player comes again (*evtPing*). If any of the players get bored of the game the match can be finished (*evtFinish*).

It is easy to see that such state machines can be statically connected or bounded to one or more published contracts. Operations can be checked if the state machine is in a state that allows the particular operation and can trigger state transitions. When the user instantiates one of the published contracts a state machine instance is automatically attached to the contracts.

Static binding can be implemented declaratively and it is compulsory to have one state machine instance per session.

To describe it formally remind the definition of the finite state machine or simply state machine:

$$FSM = (\Sigma, S, s_0, \delta, F)$$

Where

(1) Σ represents the input alphabet, in our case the set of state transitions
(2) S is a finite not empty set of states
(3) s_0 is an initial state, that is member of S
(4) $\delta : S \times \Sigma \rightarrow S$ is the state transition function
(5) F is the set of finite states, non-empty set in our case

Using the above definition the following restrictions can be applied:

$$\forall i \varepsilon [1..n] : \forall j \varepsilon [1..t_i] \begin{cases} S_{M_i,j} \varepsilon S \\ A_{M_i,j} \varepsilon \Sigma \\ (S_{M_i,j}, A_{M_i,j}) \varepsilon D_\delta \end{cases}$$

It restricts the states, the state transitions and the state transitions available in certain states.

3.3. **Client Identity Validation.** In the previous two subsections we have shown that it is indispensable to restrict callers by runtime type or group membership and it is also indispensable to instrument the correct order of business operation execution, enforce business rules.

Notwithstanding the previously mentioned two assurances there is another problem that we show in the context of our ping-pong game. When Player 1 and Player 2 start playing a ping-pong match we have to ensure that the players remain the same until the end of the match. In other words, t hey do not change sides and they are not substituted with other players. In short we have to maintain and validate the identity of players until the end of the match.

It is possible to dedicate a referee or coordinator that assigns well-defined identities for participants that can be ensured at method calls. For example the player that gets elected as First Player always gets Identity no. *1* while the other player gets *2* .

The above may not give protection from tampering the player identity. But when we assign the *(Name of the Computer, Process Id, Object Reference Id)* triplet to the identity and track it on the server side, it cannot be tampered because the name of the computer must be unique on the network level. Similarly the process id must be unique on the computer level; while the object reference id (practically pair of the runtime type and some type-level unique object id) must be unique on the process level (e.g. hash code is unique for same-typed objects in .NET).

3.4. **Network and Certificate Validation of Clients.** Firewalls can restrict access from clients deployed on certain subnets or IP addresses to the server. More advanced firewalls can restrict access to the server by domain level user identity; however that capability is only a subset of distributed runtime access control described in this paper.

Our first aim is to declaratively restrict access to specific contracts and also methods for certain subnets even IP addresses.

The other thing that loosely relates to some sort of network-level validation of clients is client certificate validation. Using client certificates it can be verified if the server communicates with a certified, trusted, verified and possibly well-working client. The server can verify if it communicates with clients having the certificate issued by a trusted authority.

3.5. **Definition of Legal Calls.** Let H be the information-set provided and available at a method call:

$$H = (T_H, S_a, I_H, N_H)$$

Where

1. T_H is the type of the caller
2. S_a the actual state (business process state)
3. I_H is the identity of the caller
4. N_H is the network properties of the caller

We say that a call is legal with respect to a method (M_i), when H conforms to the following restrictions:

1. $T_H \varepsilon \{T_{M_i,1}, T_{M_i,2}, \ldots, T_{M_i,r_i}\} \bigcap \{T_{c_1}, T_{c_2}, \ldots, T_{c_q}\}$
2. $S_a \varepsilon \{S_{M_i,1}, S_{M_i,2}, \ldots, S_{M_i,t_i}\}$
3. $I_H \varepsilon \{I_{M_i,1}, I_{M_i,2}, \ldots, I_{M_i,y_i}\} \bigcap \{I_{c_1}, I_{c_2}, \ldots, I_{c_w}\}$
4. $N_H \varepsilon \{N_{M_i,1}, N_{M_i,2}, \ldots, N_{M_i,u_i}\} \bigcap \{N_{c_1}, N_{c_2}, \ldots, N_{c_e}\}$

The four restrictions apply to the four eligible properties of H. However, the second restriction applies only to the available states because the state transitions are restricted by the FSM itself.

4. Possible Implementation in the .NET 3.0 Environment

We have created a pilot implementation of the previously described security mechanism extension in .NET 3.0. .NET [14] is a programming platform from Microsoft that helps to easily and effectively create even large scale connected applications built on standard technologies like the Web Service platform [20].

Version 3.0 of .NET introduced two innovative technologies that are used by our solution:

1. WCF - Windows Communication Foundation and
2. WF - Windows Workflow Foundation

In the following two subsections we shortly describe the benefits of these technologies then show some implementation details.

4.1. **WCF - Windows Communication Foundation.** 'WCF is Microsoft's unified framework for building secure, reliable, transacted, and interoperable distributed applications.' [22]

In our situation it means that we get a unified interface for distributed communication while we have the possibility to configure the communication address and binding for our contracts. We can configure different transport and messaging formats (binary, HTTP request, SOAP (Web Service), WSE*, message queue, etc.), and the communication platform (data transfer protocol, encoding, formatting, etc.).

4.2. **WF - Windows Workflow Foundation.** 'WF is the programming model, engine and tools for quickly building workflow enabled applications. WF radically enhances a developer's ability to model and support business processes.' [23]

WF has the ability to model states and state transitions of state machines that resembles mathematical state machines.

4.3. **Ping-Pong Example.** Because of space limitations we can show only the server side of our implementation in detail. First we will show and explain the contract definition of our ping-pong game exposed by WCF.

The following listing shows the contract definition as an interface in C#:

```
[ServiceContract(SessionMode=SessionMode.Required)]
[StateMachineDriven]
[CallerIdentityDriven]
public interface IPingPongService : IMultiSession
{
    [OperationContract]
    [AllowedCaller("Client.Player")]
    [AllowedIdentity("1")]
    [AllowedState("stFirst,stPong")]
    [RaiseTransitionEvent("PingEvent")]
    int Ping();

    [OperationContract]
    [AllowedCaller("Client.Player")]
    [AllowedIdentity("2")]
    [AllowedState("stPing")]
    [RaiseTransitionEvent("PongEvent")]
    int Pong();

    [OperationContract]
    [AllowedCaller("Client.Player")]
    [AllowedIdentity("1,2")]
```

```
        [AllowedState("stPong")]
        [RaiseTransitionEvent("FinishEvent")]
        int Finish();
    }
```

The first line contains a built-in `ServiceContract` attribute attached to the `IPingPongService` interface that enables classes implementing the interface to be exported as a service.

The `StateMachineDriven` and the `CallerIdentityDriver` attributes are part of our framework that enables the contract to be validated against state machine states and events, and check for the caller.

The `IPingPongService` interface inherits from the `IMultiSession` interface which enables our solution to share the same session across multiple instances of the same contract and also multiple instances of multiple contracts. It is not used in this example; we only indicate the possibility with the remark that SOA applications and distributed object systems do not encourage the usage of sessions.

The `OperationContract` attribute is the method-level pair of the attribute `ServiceContract`. `AllowedCaller` and `AllowedIdentity` attributes define the allowed caller types and identities at particular methods. The `AllowedState` attribute relates to the state machine controlling the ping-pong game and dictate the states that certain operations can be executed at while the attribute `RaiseTransitionEvent` instructs our framework to do a state transition after successful method executions.

The previously explained interface is exposed to the client side also while the implementation of the interface stays on the server side and defines properties that are exclusively server specific:

```
    [StateMachineParameters(typeof(PingPongWF),
                           typeof(PingPongController))]
    class PingPongService : MultiSession,
                                 IPingPongService
    {
...
```

The `StateMachineParameters` attribute declares a state machine workflow type and a controller type that will be instantiated when the first call occurs. This state machine and controller instance will drive the process (the game in our example).

4.4. **Custom Behaviors.** Every call to the exposed operations has to be intercepted on the server side and the security checks described in this paper have to be performed. WCF has the ability to extend our service endpoints with custom behaviors that can be used to do security checks.

We mention that WCF calls do not submit the client side caller type and identity information automatically therefore at the client side we have to add headers to every call that contain this information using custom client-side behaviors.

The following XML fragment shows the server side configuration that defines the extension that is responsible for doing security checks before the execution of the exposed operation:

```
<extensions>
  <behaviorExtensions>
    <add name="distrRac"
         type="ServerLib.RACServerBehaviorExtension, ServerLib,
Version=1.0.0.0, Culture=neutral,
PublicKeyToken=d18ff0ec0229ce90" />
  </behaviorExtensions>
</extensions>
```

At the client side, there is a similar configuration setting that refers to the `ClientLib.RACClientBehaviorExtension` type in the `ClientLib` assembly.

Connecting these extensions to the appropriate services some more lines of XML configuration has to be added.

We show the client code fragment that adds the type of the caller to the request headers that will be verified on the server side:

```
StackTrace stackTrace = new StackTrace(false);
StackFrame callerFrame = ClientHelper.GetCallerMethod(stackTrace);
request.Headers.Add(MessageHeader.CreateHeader(
    DISTRRAC_HEADERID, DISTRRAC_NS,
    callerFrame.GetMethod().DeclaringType.FullName));
```

On the server side the following code fragment is executed that checks the type and identity of the caller:

```
string absUri = request.Headers.To.AbsoluteUri;
Type contract = ServerHelper.GetContract(absUri);
object []drivenAttrs=ServerHelper.GetDrivenByAttributes(contract);
MethodInfo targetMethod = ServerHelper.GetTargetMethod(absUri);

bool callerIdentityDriven =
             ServerHelper.IsDrivenByCallerIdentity(drivenAttrs);
bool stateMachineDriven =
             ServerHelper.IsDrivenByStateMachine(drivenAttrs);

if (callerIdentityDriven)
{
    object[] callerAttrs =
           ServerHelper.GetCallerAttributes(targetMethod);
    string callerType =
           request.Headers.GetHeader<string>(DISTRRAC_HEADERID,
                                              DISTRRAC_NS);
    if (!ServerHelper.IsCallerAllowed(callerAttrs, callerType))
    {
        throw new InvalidCallerException();
    }
}
```

The state machine based verification is performed similarly, however, in that case after the execution of the exposed operation the state machine is driven to the next state.

The other components of the H information set can be checked similarly therefore we omit the discussion of their implementation.

5. Summary and Future Work

In this paper we have studied access control mechanisms that can be applied in case of distributed software systems.

Applications serving business processes are usually implemented as a distributed system: they span over different servers on different networks. Typical properties of such applications include dynamism: the business goals they serve change just as often as the programming or hardware environments. In order to successfully fight challenges imposed by the nature of these applications, the basic principles of Service Oriented Architecture (SOA) have been formed. SOA is a natural extension and descendant of modular programming: the functions of modules are published through interfaces.

In our work we have focused on the public interfaces of SOA applications with the following restrictions:

(1) The application should use homogeneous communication platform and
(2) The service should have some information about the clients.

We have described motivating examples showing why it is often not enough to rely ourselves on standard security mechanisms of existing standards. Starting from the motivating examples we have shown why lower level access control mechanisms are necessary to protect the interfaces exposing functionality to the outside world.

We have elaborated our research and extended the security context of distributed applications based on the following properties: distributed runtime access control, business process and client identity validation, and the network identity validation of clients. The above properties can be validated at method level at every single call. The granularity level of most of the above properties changes from application level to method level. Informally speaking, we introduce a business call level fine-grained "firewall".

We have been following a semi-formal approach of the topic, and have given a definition of a legal method call. Other solutions described in the related work section solve only a part of the security problems specific to distributed enterprise applications while we aimed to create a framework that answers most of emerging questions.

The formal approach described important runtime restrictions for distributed object systems. However, the formal approach itself cannot guarantee that it can be implemented in practice. In order to prove the practical applicability of the proposal, we have implemented a pilot framework in the .NET 3.0 programming environment. The implementation uses the innovative technologies of the .NET framework: Windows Communication Foundation and Workflow Foundation. We exploited declarative programming to the maximal possible extent.

One of our further research directions can be the extension of the pilot implementation with different environments, such as the Java platform. The capabilities of widely used industrial standards should be analyzed, and, if necessary, the presented formal framework should be refined in order to adapt to different security mechanisms.

We designed our framework to be extensible with other custom security mechanisms that may be orthogonal to the formalized and implemented ones.

This paper also shows the need for runtime access control in order to secure distributed applications. Therefore we hope that similar frameworks will gain popularity and help the quality improvement of complex, distributed systems.

References

[1] A. Barros, G. Decker, M. Dumas, F. Weber: *Correlation Patterns in Service-Oriented Architectures*, In Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007), Braga (Portugal), 2007. Springer Verlag, pages 245-259.

[2] M. Blaze, J. Feigenbaum, J. Ioannidis, A. D. Keromytis. *The role of trust management in distributed systems security*, Secure Internet Programming. Springer Verlag, 1999, pages 185-210

[3] L. Bauer, S. Garriss, M. K. Reiter. *Efficient Proving for Practical Distributed Access-Control Systems*. Computer Security - ESORICS 2007, 2007, Springer Verlag, pages 19-37

[4] M. Biczó, K. Pócza, Z. Porkoláb. *Runtime Access Control in C# 3.0 Using Extension Methods*, Proceedings of the 10th Symposium on Programming Languages and Software Tools (SPLST 2007), Dobogókő (Hungary), 2007, pages 45-60.

[5] D. Cooney, M. Dumas, P. Roe: *GPSL: A Programming Language for Service Implementation*, In Proceedings of the 8th International Conference on Fundamental Approaches to Software Engineering (FASE), Vienna, Austria, March 2006. Springer Verlag, pages 3-17.

[6] J. Crampton: *A reference monitor for workflow systems with constrained task execution*, In Proceedings of the 10th ACM Symposium on Access Control Models and Technologies, pages 38-47, 2005.

[7] N. Damianou, N. Dulay, E. Lupu, M. Sloman and T. Tonouchi. *Policy Tools for Domain Based Distributed Systems Management* . IFIP/IEEE Symposium on Network Operations and Management. Florence, Italy, 2002.

[8] D. Domingos, A. R. Silva, P. Veiga. *Workflow Access Control from a Business Perspective.* International Conference on Enterprise Information Systems, 2004

[9] Frank E. Developing Distributed Enterprise Applications With the MS Common Object Model. Hungry Minds, 1997, ISBN 0-764580-44-2

[10] R. Gronmo, M. C. Jaeger, A. Wombacher: *A Service Composition Construct to Support Iterative Development*, In Proceedings of the 10th International Conference on Fundamental Approaches to Software Engineering (FASE 2007), Braga (Portugal), 2007. Springer Verlag, pages 230-244.

[11] M. B. Juric, B. Mathew, P. Sarang. *Business Process Execution Language for Web Services: BPEL and BPEL4WS*, Packt Publishing, 2004, ISBN 1-904811-18-3

[12] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, J. Irwin. *Aspect-Oriented Programming*, Proceedings of the European Conference on Object-Oriented Programming, 1997, Springer Verlag, pages 220-242.

[13] B. Meyer. Eiffel - The Language, Prentice Hall, 1992. ISBN 0-13-247925-7

[14] .NET Framework: http://msdn2.microsoft.com/netframework/

[15] K. Pócza, M. Biczó, Z. Porkoláb. *Runtime Access Control in C#*, Proceedings of the 7th International Conference on Applied Informatics (ICAI), Eger, Hungary, 2007, jan. 28-31.

[16] A. Snyder. *Encapsulation and inheritance in object-oriented programming languages.* In Proceedings of OOPSLA '86, pages 38-45. ACM Press, 1986.

[17] Z. Tari, O. Bukhres. *Fundamentals of Distributed Object Systems: The CORBA Perspective*, Wiley, 2001, ISBN 978-0-471-35198-6

[18] D. Thomsen, D. O'Brien, and J. Bogle. *Role Based Access Control Framework for Network Enterprises.* In Proceedings of 14th Annual Computer Security Applications Conference. December 1998

[19] UML: http://www.uml.org/

[20] S. Weerawarana, F. Curbera, F. Leymann, T. Storey, D. F. Ferguson. *Web Services Platform Architecture : SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging, and More* . Prentice Hall PTR, 2005.

[21] X. Wei, W. Jun, L. Yu, L. Jing. *SOWAC: a service-oriented workflow access control model.* Proceedings of the 28th Annual International Computer Security and Applications Conferences, 2004, pages 128-134.

[22] Windows Communication Foundation: http://wcf.netfx3.com/

[23] Windows Workflow Foundation: http://wf.netfx3.com/

Eötvös Loránd University, Fac. of Informatics, Dept. of Programming Lang. and Compilers, Pázmány Péter sétány 1/c. H-1117, Budapest, Hungary

E-mail address: kpocza@kpocza.net, mihaly.biczo@t-online.hu, gsd@elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

META<FUN> – TOWARDS A FUNCTIONAL-STYLE INTERFACE FOR C++ TEMPLATE METAPROGRAMS

ÁDÁM SIPOS, ZOLTÁN PORKOLÁB, AND VIKTÓRIA ZSÓK

Abstract. Template metaprogramming is an emerging new direction in C++ programming for executing algorithms at compilation time. Despite that template metaprogramming has a strong relationship with functional programming, existing template metaprogram libraries do not follow the requirements of the functional paradigm. In this paper we discuss the possibility to enhance the syntactical expressivity of template metaprograms using an embedded functional language. For this purpose we define *EClean*, a subset of Clean, a purely functional lazy programming language. A parser, and a graph-rewriting engine for EClean have been implemented. The engine itself is a compile-time template metaprogram library using standard C++ language features. To demonstrate the feasibility of the approach lazy evaluation of infinite data structures is implemented.

1. Introduction

Template metaprogramming is an emerging new direction in C++ programming for executing algorithms at compilation time. The relationship between C++ template metaprograms and functional programming is well-known: most properties of template metaprograms are closely related to the principles of the functional programming paradigm. On the other hand, C++ has a strong heritage of imperative programming (namely from C and Algol68) influenced by object-orientation (Simula67). Furthermore the syntax of the C++ templates is especially ugly. As a result, C++ template metaprograms are often hard to read, and hopeless to maintain.

Ideally, the programming language interface has to match the paradigm the program is written in. The subject of the *Meta<Fun>* project is writing template metaprograms in a functional language and embedding them into

Received by the editors: September 15, 2008.

1998 *CR Categories and Descriptors.* D.3 [**Programming Languages**]: D.3.2 Applicative (functional) languages; D.3 [**Programming Languages**]: D.3.2 Language Classification, C++.

Key words and phrases. C++ Template Metaprogramming, Clean.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

C++ programs. This code is translated into classical template metaprograms by a translator. The result is a native C++ program that complies with the ANSI standard [3].

Clean is a general purpose, purely functional, lazy language [8]. In our approach we explore Clean's main features including uniqueness types, higher order functions, and the powerful constructor-based syntax for generating data structures. Clean also supports infinite data structures via delayed evaluation. We defined *EClean* as a subset of the Clean language. EClean is used as an embedded language for expressing template metaprogramming.

In this article we overview the most important properties of the functional paradigm, and evaluate their possible translation techniques into C++ metaprograms. The graph-rewriting system of Clean has been implemented as a C++ template metaprogram library. With the help of the engine, EClean programs can be translated into C++ template metaprograms as clients of this library and can be evaluated in a semantically equivalent way. Delayed evaluation of infinite data structures are also implemented and presented by examples.

The rest of the paper is organized as follows: In section 2 we give a technical overview of C++ template metaprograms (TMP), and discuss the relationship between TMP and functional programming. Lazy data structures, evaluation, and the template metaprogram implementation of the graph rewriting system of Clean is described in section 3. Section 4 discusses future work, and related work is presented in section 5. The paper is concluded in section 6.

2. Metaprogramming and Functional Programming

Templates are key elements of C++ programming language [13]. They enable data structures and algorithms be parameterized by types thus capturing commonalities of abstractions at compilation time without performance penalties at runtime [17]. *Generic programming* [12, 11, 9], is a recently emerged programming paradigm, which enables the developer to implement reusable codes easily. Reusable components – in most cases data structures and algorithms – are implemented in C++ with the heavy use of templates.

In C++, in order to use a template with some specific type, an *instantiation* is required. This process can be initiated either implicitly by the compiler when a template with a new type argument is referred, or explicitly by the programmer. During instantiation the template parameters are substituted with the concrete arguments, and the generated new code is compiled. The instantiation mechanism enables us to write smart template codes that execute algorithms at compilation time [16, 18]. This paradigm, *Template Metaprogramming* (TMP) is used for numerous purposes. These include transferring calculations to compile-time, thus speeding up the execution of the program; implementing *concept checking* [22, 14, 21] (testing for certain properties of types at compilation); implementing *active libraries* [5], and others.

Conditional statements, like the stopping of recursions, are implemented with the help of specializations. Subprograms in ordinary C++ programs can be used as data via function pointers or functor classes. Metaprograms are first class citizens in template metaprograms, as they can be passed as parameters for other metaprograms [6]. Data and computation results are expressed at runtime programs as constant values or literals. In metaprograms we use `static const` and enumeration values to store quantitative information.

Complex data structures are also available for metaprograms. Recursive templates are able to store information in various forms, most frequently as tree structures, or sequences. Tree structures are the favorite implementation forms of expression templates [19]. The canonical examples for sequential data structures are `typelist` [2] and the elements of the `boost::mpl` library [22, 7, 1].

By enabling the compile-time code adaptation, C++ template metaprograms (TMP) is a style within the *generative programming* paradigm [6]. Template metaprogramming is *Turing-complete* [20], in theory its expressive power is equivalent to that of a Turing machine (and thus most programming languages).

Despite all of its advantages TMP is not yet widely used in the software industry due to the lack of coding standards, and software tools. A common problem with TMP is the tedious syntax, and long code. Libraries like `boost::mpl` help the programmers by hiding implementation details of certain algorithms and containers, but still a big part of coding is left to the user. Due to the lack of a standardized interface for TMP, naming and coding conventions vary from programmer to programmer.

Template metaprogramming is many times regarded as a pure functional programming style. The common properties of metaprogramming and functional languages include referential transparency and the lack of variables, loops, and assignments. One of the most important functional properties of TMP is that if a certain entity (the aforementioned constants, enumeration values, types) has been defined, it will be immutable. A metaprogram does not contain assignments. That is the reason why we use recursion and specialization to implement loops: we are not able to change the value of any loop variable. Immutability – as in functional languages – has a positive effect too: unwanted side effects do not occur.

In our opinion, the similarities require a more thorough examination, as the metaprogramming realm could benefit from the introduction and library implementation of more functional techniques.

Two methods are possible for integrating a functional interface into C++: modifying the compiler to extend the language itself, or creating a library-level solution and using a preprocessor or macros. The first approach is probably quicker, easier, and more flexible, but at the same time a language extension is undesirable in the case of a standardized, widely used language like C++.

Our approach is to re-implement the graph-rewriting engine of the Clean language as a compile-time metaprogram library using only ANSI standard compliant C++ language elements. Thus our solution has numerous advantages. As the user written embedded code is separated from the graph-rewriting engine, the embedded Clean code fragments can be translated into C++ template metaprograms independently. Since the engine follows the graph-rewriting rules of the Clean language as it is defined in [4], the semantics of the translated code is as close to the intentions of the programmer as possible. As our solution uses only standard C++ elements, the library is highly portable.

3. Lazy Evaluation and Implementation of the Graph-rewriting Engine

As lazy evaluation is one of the most characteristic features of the Clean language [10], our research centers around lazy evaluation and its application in C++ template metaprograms. A *lazy* evaluation strategy means that *”a redex is only evaluated when it is needed to compute the final result”*. This enables us to specify lists that contain an infinite number of elements, e.g. the list of natural numbers: `[1..]`. Our running example for the usage of lazy lists is the *Eratosthenes sieve* algorithm producing the first arbitrarily many primes. (The symbols R1..R6 are line numberings)

```
(R1)   take 0 xs = []
(R2)   take n [x,xs] = [x, take n-1 xs]
(R3)   sieve [prime:rest] = [prime : sieve (filter prime rest)]
(R4)   filter p [h:tl] | h rem p == 0  = filter p tl
                                     = [h : filter p tl]
(R5)   filter p [] = []
(R6)   Start = take 10 (sieve ([2..]))
```

The Clean graph rewriting engine carries out the following evaluation.

```
(F1)   take 10 (sieve [2..])
(F2)   take 10 (sieve [2, [3..]])
(F3)   take 10 ([2, sieve (filter 2 [3..])])
(F4)   [2, take 9 (sieve (filter 2 [3..]))]
(F5)   [2, take 9 (sieve [3, filter 2 [4..])]
(F6)   [2, take 9 [3, sieve (filter 3 (filter 2 [4..]))]]
(F7)   [2, 3, take 8 (sieve (filter 3 (filter 2 [4..])))]
```

. . .

In the following we present via examples the transformation method of an EClean program into C++ templates. Our EClean system consists of two main parts: the *parser* – responsible for transforming EClean code into metaprograms–, and the *engine* – doing the actual execution of the functional code.

The compilation of a C++ program using EClean code parts is done in the following steps:

- The C++ preprocessor is invoked in the execution of the necessary header file inclusions and macro substitutions. The EClean library containing the engine and supporting metaprograms is also imported at this point.
- The source code is divided into C++ parts and EClean parts.
- The EClean parts are transformed by the parser of EClean into C++ metaprogram code snippets.
- This transformed source code is handed to the C++ compiler.
- The C++ compiler invokes the instantiation chain at code parts where the `Start` expression is used, thus activating the EClean engine.
- The engine emulates Clean's graph rewriting, and thus executes the EClean program.
- When no further rewriting can be done, the finished expression's value is calculated, if necessary.

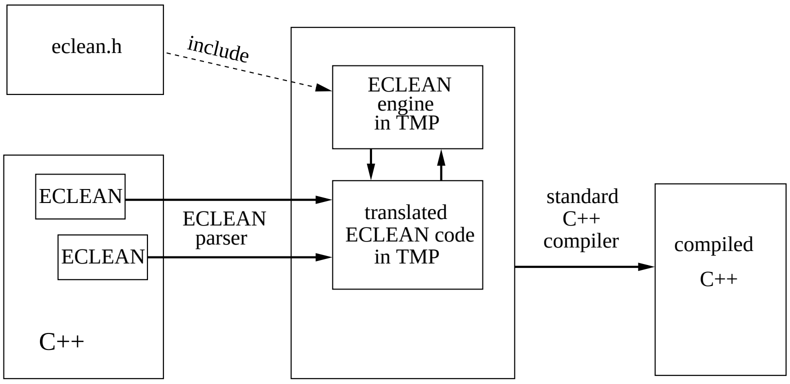


FIGURE 1. EClean transformation and compilation process

3.1. **The sieve program.** In Section 2 we have described the various language constructs available in metaprogramming. We now use `typedefs`, and types created from templates to represent the EClean expressions. In this approach our example Start expression has the form `take<mpl::int_<10>, sieve<EnumFrom<mpl::int_<2> > > >`. Here `take`, `sieve`, and `EnumFrom` are all `struct templates` having the corresponding signatures.

The graph rewriting process can be emulated with the C++ compiler's instantiation process. When a template with certain arguments has to be instantiated, the C++ compiler chooses the narrowest matching template of that name from the specializations. Therefore the rules can be implemented with template partial specializations. Each partial specialization has an inner `typedef` called `right` which represents the right side of a pattern matching rule. At the same time the template's name and parameter list represent the left side of a pattern matching rule, and the compiler will choose the most suitable of the specializations of the same name. Let us consider the following example, which describes the `sieve` rule (`sieve [prime:rest] = [prime : sieve (filter prime rest)]`).

```
template <class prime, class ys>
struct sieve<Cons<prime,ys> > {
    typedef Cons<prime,sieve<filter<prime,ys> > > right;
};
```

The `sieve` template has two parameters, `prime` and `ys`. This template describes the workings of (R3) in our Clean example. In case a subexpression has the form `sieve<Cons<N,T> >` where `N` and `T` are arbitrary types, the previously defined `sieve` specialization will be chosen by the compiler as a substitute for the subexpression. Note that even though `N` and `T` are general types, the `sieve` template expects `N` to be a `mpl::int_`, and `T` a list of `mpl::int_` types.

However, in order to be able to apply this rewriting rule, an exact match is needed during the rewriting process. For example in (F1) during the evaluation process the previous `sieve` rule will be considered as applicable when rewriting the subexpression `sieve [2..]`. The problem is that the argument `[2..]` (`EnumFrom` 2) does not match the `sieve` partial specialization parameter list which is expecting an expression in the form `Cons<N,T>` with types `N` and `T`. During the compilation the C++ compiler will instantiate the type `sieve<EnumFrom<mpl::int_<2> > >`. However this is a pattern matching failure which has to be detected. Therefore each function must implement a partial specialization for the general case, when none of the rules with the same name can be applied. The symbol `NoMatch` is introduced, which signs that even though this template has been instantiated with some parameter `xs`, there is no applicable rule for this argument. `NoMatch` is a simple empty `class`.

```
template <class xs>
struct sieve {
    typedef NoMatch right;
};
```

The previously introduced `filter` function's case distinction is used to determine at compilation time whether `x` is divisible by `p`, and depending on that decision either of the two alternatives can be chosen as the substitution. The C++ transformation of `filter` utilizes `mpl::if_` for making a compile-time decision:

```
template <int p, class x, class xs >
struct filter<boost::mpl::int_<p>, Cons<x,xs> > {
    typedef typename boost::mpl::if_
    <
        typename equal_to
        <
            typename modulus<x,p>::type,
            boost::mpl::int_<0>
        >::type,
        filter<p,xs>,
        Cons<x,filter<p,xs> >
    >::type right;
};
```

The `mpl::if_` construct makes a decision at compilation time. The first type parameter is the `if` condition, which in our case is an `equal_to` template, whose inner `type` typedef is a `mpl::bool_`. Depending on this `bool_`'s `value`, either the first, or the second parameter is chosen.

The working of the transformed `EnumFrom` is similar to the one in Clean: if a rewriting is needed with `EnumFrom`, a new list is created consisting of the list's head number, and an `EnumFrom` and the next number.

```
template <class r>
struct EnumFrom {
    typedef
        Cons<r,EnumFrom<boost::mpl::int_<r::value+1> > > right;
};
```

All other functions can also be translated into templates using analogies with the previous examples.

In the following we present the parser recognizing EClean expressions, and transforming them to the previous form.

3.2. **The parser.** The parser was written in Java, using the ANTLR LL(k) parser generator. The parser recognizes a subset of the Clean language, as our aim was to create an embedded language aiding programmers in writing metaprograms, and not the implementation of a fully capable Clean compiler.

The parser's workings are as follows. The first stage in transforming an embedded clean code into a template metaprogram is parsing the EClean code. The notation for distinguishing between regular C++ code and EClean code is the two apostrophes: ' '

3.2.1. *Function transformation.* Each function's signature is recorded when the function's declaration is parsed. At the same time, the declaration is transformed into a general template definition with the `NoMatch` tag, supporting the non-matching cases of the graph rewriting.
Let us consider the following example:

```
take :: Int [Int] -> [Int]
```

This function declaration is transformed into the following template:

```
template <class,class>
struct take {
    typedef NoMatch right;
};
```

The two function alternatives of `take` are transformed as follows:

```
template <int n, class x, class xs>
struct take<mpl::int_<n>, Cons<x,xs> > {
    typedef Cons<x,take<mpl::int_<n - 1>,xs> > right;
};

template <class xs>
struct take<mpl::int_<0>, xs> {
    typedef NullType right;
};
```

The first alternative accepts three parameters, an `int n` representing the first `Int` parameter (how many elements we want to take), and two arbitrary types `x` and `xs` representing the head and tail of a list. On the other hand it is guaranteed that when this function is invoked, `x` will always be a `mpl::int_`, and `xs` will either be a list of `mpl::int_` types, or the `NullType` (`Nil`). The working mechanism of the parser's code transformation is the guarantee for this.

3.3. **The graph-rewriting engine.** Until now we have translated the Clean rewriting rules into C++ templates, by defining their names, parameter lists (the rule's partial specialization), and their right sides. These templates will be used to create types representing expressions thus storing information at compilation time. This is the first abstraction layer. In the following we present the next abstraction level, that uses this stored information. This is done by the library's core, the partial specializations of the `Eval struct template`, which evaluate a given EClean expression.

Since the specialization's parameter is a template itself (representing an expression), its own parameter list has to be defined too. Because of this constraint separate implementations are needed for the evaluation of expressions with different arities. In the following we present one version of `Eval` that evaluates expressions with exactly one parameter:

```
 1 template <class T1, template <class> class Expr>
 2 struct Eval<Expr<T1> >
 3 {
 4     typedef typename
 5         if_c<is_same<typename Expr<T1>::right,
 6                      NoMatch>::value,
 7         typename
 8             if_c<!Eval<T1>::second,
 9                 Expr<T1>,
10                 Expr<typename Eval<T1>::result>
11             >::type,
12             typename Expr<T1>::right
13         >::type result;
14
15     static const bool second =
16         !(is_same<typename Expr<T1>::right,NoMatch>::value &&
17          !Eval<T1>::second);
18 };
```

The working mechanism of `Eval` is as follows. `Eval` takes one argument, an expression `Expr` with one parameter `T1`. The type variable `T1` can be any type, e.g. `int`, a list of `ints`, or a further subexpression. This way `Eval` handles other templates. The return type `result` defined in line 13 contains the newly rewritten subexpression, or the same input expression if no rule can be applied to the expression and its parameters.

When the template `Expr` has no partial specialization for the parameter `T1`, the compiler chooses the general template as described in Section 3.1. The compile-time `if_c` in line 5 is used to determine if this is the case, and the `Expr<T1>::right` is equal to `NoMatch`.

- If this is the case, another `if_c` is invoked. In line 8 `T1`, the first (and only) argument is evaluated, with a recursive call to `Eval`. The boolean `second` determines whether `T1` or any of its parameters could be rewritten. If no rewriting has been done among these children, `Eval`'s return type will be the original input expression. Otherwise the return type is the input expression with its `T1` argument substituted with `Eval<T1>::result`, which means that either `T1` itself, or one of its parameters has been rewritten. This mechanism is similar to type inference.

- On the other hand, if a match has been found (the `if_c` conditional statement returned with a false value), the whole expression is rewritten, and `Eval` returns with the transformed expression (line 12).

The aforementioned boolean value `second` is defined by each `Eval` specialization (line 15). It is the logical value signaling whether the expression itself, or one of its subexpressions has been rewritten.

The implementation of `Eval` for more parameters is very similar to the previous example, the difference being that these parameters also have to be recursively checked for rewriting.

As our expressions are stored as types, during the transformation process the expression's changes are represented by the introduction of new types. The course of the transformation is the very same as with the Clean example. The following types are created as `right` typedefs:

```
take<10,sieve<EnumFrom<2> > >
take<10,sieve<Cons<2,EnumFrom<3> > > >
take<10,Cons<2,sieve<filter<2,EnumFrom<3> > > > >
Cons<2,take<9,sieve<filter<2>,EnumFrom<3> > > >
Cons<2,take<9,sieve<3,filter<2,EnumFrom<4> > > > >
Cons<2,take<9,Cons<3,sieve<filter<3,EnumFrom<4> > > > > >
Cons<2,3,take<8,filter<3,filter<2,EnumFrom<4> > > > >
...
```

(Note that in the example all `mpl::int_` prefixes are omitted from the `int` values for readibility's sake.)

We have demonstrated the evaluation engine's implementation, and its working mechanism.

4. Future work

One of the most interesting questions in our hybrid approach is to distinguish between problems that can be dealt with by EClean alone, and those that do require template metaprogramming and compiler support. The EClean parser could choose function calls that can be run separately and their result computed without the transformation procedure and the invocation of the C++ compiler. On the other hand, references to C++ constants and types could be placed within the EClean code, and used by the EClean function in a callback-style. This would result in much greater flexibility and interactivity between EClean and C++.

In the future we will include support for more scalar types (`bool`, `long`, etc) besides the implemented `Int`, and the list construct. Another interesting direction is the introduction of special EClean types like `Type` representing a C++ type, `Func` representing a C++ function or even a function pointer.

5. Related Work

Functional language-like behavior in C++ has already been studied. *Functional C++* (FC++) [15] is a library introducing functional programming tools to C++, including currying, higher-order functions, and lazy data types. FC++, however, is a runtime library, and our aim was to utilize functional programming techniques at compilation time.

The `boost::mpl` library is a mature library for C++ template metaprogramming. `Boost::mpl` contains a number of compile-time data structures, algorithms, and functional-style features, like *Partial Metafunction Application* and *Higher-order metafunctions*. However, `boost::mpl` were designed mainly to follow the interface of the C++ Standard Template Library. There is no explicit support for lazy infinite data structures either.

6. Conclusion

In this paper we discussed the Meta<Fun> project which enhances the syntactical expressivity of C++ template metaprograms. EClean, a subset of the general-purpose functional programming language Clean is introduced as an embedded language to write metaprogram code in a C++ host environment. The graph-rewriting system of the Clean language has been implemented as a template metaprogram library. Functional code fragments are translated into classical C++ template metaprograms with the help of a parser. The rewritten metaprogram fragments are passed to the rewriting library. Lazy evaluation of infinite data structures is implemented to demonstrate the feasibility of the approach. Since the graph-rewriting library uses only standard C++ language features, our solution requires no language extension and is highly portable.

References

[1] D. Abrahams, A. Gurtovoy, *C++ template metaprogramming, Concepts, Tools, and Techniques from Boost and Beyond*, Addison-Wesley, Boston, 2004.

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley, 2001.

[3] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[4] T. H. Brus, C. J. D. van Eekelen, M. O. van Leer, M. J. Plasmeijer, *CLEAN: A language for functional graph rewriting*, Proc. of a conference on Functional programming languages and computer architecture, Springer-Verlag, 1987, pp.364-384.

[5] K. Czarnecki, U. W. Eisenecker, R. Glck, D. Vandevoorde, T. L. Veldhuizen, *Generative Programming and Active Libraries*, Springer-Verlag, 2000.

[6] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[7] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.

[8] P. Koopman, R. Plasmeijer, M. van Eeekelen, S. Smetsers, *Functional programming in Clean*, 2002

[9] D. R. Musser, A. A. Stepanov, *Algorithm-oriented Generic Libraries*, Software-practice and experience 27(7), 1994, pp.623-642.

[10] R. Plasmeijer, M. van Eeekelen, *Clean Language Report*, 2001.

[11] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.

[12] J. Siek, *A Language for Generic Programming*, PhD thesis, Indiana University, 2005.

[13] B. Stroustrup, *The C++ Programming Language Special Edition*, Addison-Wesley, 2000.

[14] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.

[15] B. McNamara, Y. Smaragdakis, *Functional programming in C++*, Proceedings of the fifth ACM SIGPLAN international conference on Functional programming, pp.118-129, 2000.

[16] E. Unruh, *Prime number computation*, ANSI X3J16-94-0075/ISO WG21-462.

[17] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[18] T. Veldhuizen, *Using C++ Template Metaprograms*, C++ Report vol. 7, no. 4, 1995, pp. 36-43.

[19] T. Veldhuizen, *Expression Templates*, C++ Report vol. 7, no. 5, 1995, pp. 26-31.

[20] T. Veldhuizen, *C++ Templates are Turing Complete*

[21] I. Zólyomi, Z. Porkoláb, *Towards a template introspection library*, LNCS Vol.3286 (2004), pp.266-282.

[22] Boost Libraries.
`http://www.boost.org/`

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary

E-mail address: `shp@inf.elte.hu, gsd@elte.hu, zsv@inf.elte.hu`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

A SOFTWARE TOOL FOR DATA ANALYSIS BASED ON FORMAL CONCEPT ANALYSIS

KATALIN TUNDE JANOSI RANCZ, VIORICA VARGA, AND JANOS PUSKAS

Abstract. Formal Concept Analysis is a useful tool to represent logical implications in datasets, to analyze the underground knowledge that lies behind large amounts of data. A database relation can be seen as a many-valued context [3]. J. Hereth in [4] introduces the formal context of functional dependencies. In this context, implications hold for functional dependencies. We develop a software application that analyzes an existing relational data table and detect functional dependencies in it. The user can choose to analyze a table from a MS SQL Server, Oracle or MySQL database and the software will build the formal context of functional dependencies. We use Conexp [6] to build the concept lattice and implications in this context. These implications will be the functional dependencies for the analyzed table. Having the functional dependencies, we can detect candidate keys and we can decide if the table is in 2NF or 3NF or BCNF. To our knowledge, this method was not implemented yet.

1. Introduction

Formal Concept Analysis (FCA) appeared in 1980s ([7]) as a mathematical theory which formalises the notion of *concept* and is nowadays considered as an AI theory. It is used as a technique for data analysis, information retrieval and knowledge representation with various successful applications ([3]).

Functional dependencies (FDs shortly) are the most common integrity constraints encountered in databases. FDs are very important in relational database design to avoid data redundancy. Extracting FDs from a relational

Received by the editors: August 9, 2008.

2000 *Mathematics Subject Classification.* 68P15 Database theory, 03G10 Lattices and related structures.

1998 *CR Categories and Descriptors.* H2 Database Management [**Topic**]: Subtopic – H2.1 Logical design *Normal forms*.

Key words and phrases. Formal concept analysis, Normal forms.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

table is a crucial task to understand data semantics useful in many database applications. Priss in [5] presents the visualization of normal forms using concept lattices, where the notion of functional dependencies is life-line.

The subject of detecting functional dependencies in relational tables was studied for a long time and recently addressed with a data mining viewpoint. Baixeries in [2] gives an interesting framework to mine functional dependencies using Formal Context Analysis. Detecting functional dependencies seems to be an actual theme, [5].

Hereth [4] presents how some basic concepts from database theory translate into the language of Formal Concept Analysis. The definition of the formal context of functional dependencies for a relational table can also be found in [4]. Regarding to this definition the context's attributes are the columns (named attributes) of the table, the tuple pairs of the table will be the objects of the context. [4] gives the proposition which asserts that in the formal context of functional dependencies for a relational table, implications are essentially functional dependencies between the columns of the relational database table.

Proposition 1. *Let $\mathcal{D}$ be a relational database and m a k-ary table in $\mathcal{D}$. For two sets $X, Y \subseteq \{\ 1, ..., k\}$ we have the following assertion: The columns Y are functionally dependent from the columns X if and only if $X \rightarrow Y$ is an implication in the formal context of functional dependencies for table m, which is notated $FD\left(m, \overrightarrow{K}(\mathcal{D})\right)$.*

Informally, normal forms are defined in traditional database theory as a means of reducing redundancy and avoiding update anomalies in relational databases. Functional dependency means that some attributes' values can be reconstructed unambiguously by the others [1].

In this paper we intend to extend a previous research presented in [8]. We implemented the method presented in [8] for database design and completed it with a software tool, which analyzes an existing relational database table. Our software named FCAFuncDepMine constructs the formal context of functional dependencies. It uses Conexp [10] to build the concept lattice and to determine the implications in this context. The implications obtained correspond to functional dependencies in the analyzed table. The software can be used in relational database design and for detecting functional dependencies in existing tables, respectively.

2. Software description

This section presents how our software constructs the context of functional dependencies of an existing relational database table. The method used in relational database design was described in [8].

The aim of our software tool is to connect to an existing database by giving the type and the name of the database, a login name and password, then the software offers a list of identified table names which can be selected for possible functional dependency examination.

The formal context of functional dependencies for the selected table has to be constructed. The attributes of the context will be the attributes of the studied table and the context's objects will be the tuple pairs of the table. A table may have a lot of tuples and much more tuple pairs. We optimize the construction of the context in both approaches.

The top of the concept lattice corresponds to tuple pairs in which there are no common values of the corresponding table attributes. Pairs of form (t, t), where t is a tuple of the table, have all attributes in common, these objects will arrive in the bottom of the lattice.

An existing table may have a very large number of tuples. In this version of our software we use Conexp, which is not able to handle very large context tables. An input set for Conexp that consists of 15 000 selected tuple pairs is processed in a reasonable time (some seconds), but if the size of the input set is larger than 20 000, Conexp will fail. In order to omit this failure, the user can set a limit for the number of the selected tuples.

Let T be this table having attributes $A_1, \dots, A_n$. The top of the concept lattice corresponds to tuple pairs in which there are no common values of the corresponding attributes. A lot of pairs of this kind may be present. Pairs which have all attributes in common, will arrive in the bottom of the lattice.

We tested concept lattices omitting tuple pairs in the top and the bottom of the lattice. During this test we did not find the same lattice as that obtained with these special tuple pairs. In order not to alter the implications, we generate only a few (but not all) of these pairs. On the other hand, we need pairs of tuples of table T, where at least one (but not all) of the attributes have the same value.

The connection being established and table T selected to analyze the existing functional dependencies, the program has to execute the next SELECT - SQL statement:

```
SELECT T1.A1,...,T1.An,T2.A1,...,T2.An
FROM T T1, T T2
WHERE (T1.A1=T2.A1 OR ... OR T1.An=T2.An)
 AND NOT (T1.A1=T2.A1 AND ... AND T1.An=T2.An)
```

This statement leads to a Cartesian-product of table T with itself, which is a very time consuming operation. The statement is transformed by eliminated NOT from it.

```
SELECT T1.A1,...,T1.An,T2.A1,...,T2.An
FROM T T1, T T2
WHERE (T1.A1=T2.A1 OR ... OR T1.An=T2.An)
  AND (T1.A1<>T2.A1 OR ... OR T1.An<>T2.An)
```

Both (s, u) and (u, s) pairs of tuples will appear in the result, but we need only one of these. Let $P1, P2, ..., Pk(k \geq 1)$ be the primary key of table T. The definition of a relational table's primary key can be found in [6]. In order to include only one of these pairs, we complete the statement's WHERE condition in case of $k = 1$ with:

```
AND (T1.P1 < T2.P1)
```

or if $k > 1$ with

```
AND (T1.P1k < T2.P1k)
```

where $P1k$ denotes the string concatenation of the primary key's component attributes, respectively.

Constructing a clustered index on one of the attributes can speed up the execution of the SELECT statement. The advantage of using this SELECT statement is that every Database Management System will generate an optimized execution plan.

With `Selected Columns` button the user can choose a list of attributes of the selected table, otherwise all attributes will be selected. In order to create the cex file for Conexp the `Select Tuple Pairs` button have to be pressed, which selects tuple pairs which have at least one of its attribute value in common. Tuple pairs in the top and in the bottom of the concept lattice can be generated optionally, checking the `Add Extra Tuple Pairs` option in the File menu. Tuple pairs being generated we have to save the cex file, then it can be used as input for Conexp, which will build the concept lattice and implications. In the following we will examine the concept lattice of the context of functional dependencies which was constructed for a relational table.

3. The structure of the application

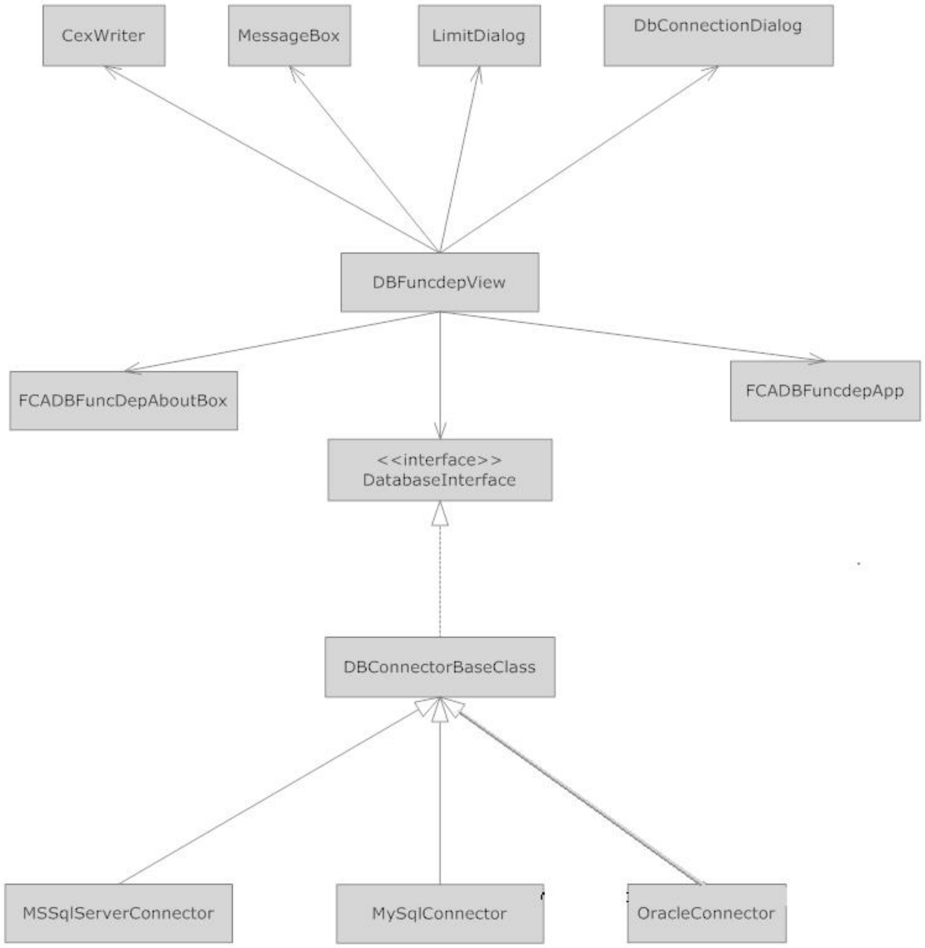


Figure 1. The class diagram of the application

The application was developed in Java programming language conforming to object-oriented programming paradigms. The class diagram of the application can be seen in Figure 1.

The class `DBFuncDepView` is the main class of the application. This class executes the graphical interface and also has a reference to the other classes, this means that every functionality can be reached trough this class.

In the interface `DatabaseInterface` are defined those functionalities with which we can execute the SELECT statement described in previous section against different Database Management Systems. It contains functionalities for selecting the list of the existing tables in the studied database and methods for selecting the rows and columns of the tables.

The class `DBConnectorBaseClass` implements the functions described in the earlier presented interface which can be implemented for each of the three DBMS.

The classes `MySQLConnector, MSSqlServerConnector, OracleConnector` contain the specific functions and drivers needed for the connection to the different databases.

The role of the `CexWriter` class is to export the tuple pairs of the context table in .cex format which is in fact in XML format. An example is in Figure 2.

4. Data Analysis

FD lattices can be used to visualise the normalforms [5]. The lattice visualizations can help to convey an understanding of what the different normalforms mean. All attributes of a database table must depend functionally on the table's key, by definition. Therefore for each set of formal concepts of a formal context there exists always a unique greatest subconcept (meet) which must be equal the bottom node of the FD lattice.

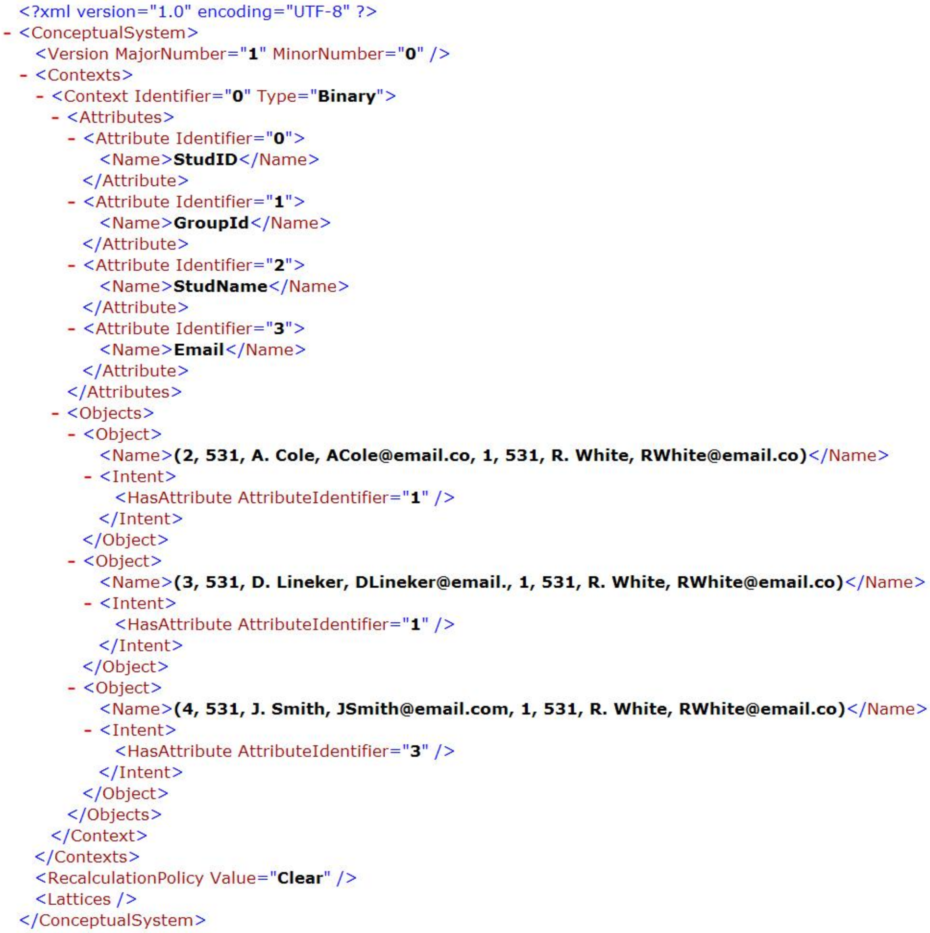
Let us begin with a simple example.

Example 1. Let be the next relational database table scheme:

`Students [StudID,StudName,GroupID,Email]`

We have analyzed this table with our software. The FD lattice and functional dependencies obtained for this table are shown in the Figure 3. The FD lattice interpretation is: the concept `StudID, StudName, Email` is a subconcept of concept `GroupID.` This means there is an implication from the concept `StudID, StudName, Email` to the concept `GroupID.` Accordingly, in every tuple pair where the `StudID` field has the same value, the value of the `GroupID` will remain the same.

Because all keys meet in the bottom node, for a determinant to be a candidate key means that the unique greatest subconcept of its attributes equals the bottom node. Consequently every attribute depends on `StudID,` therefore it is a candidate key. The same case is for `StudName` and `Email.`



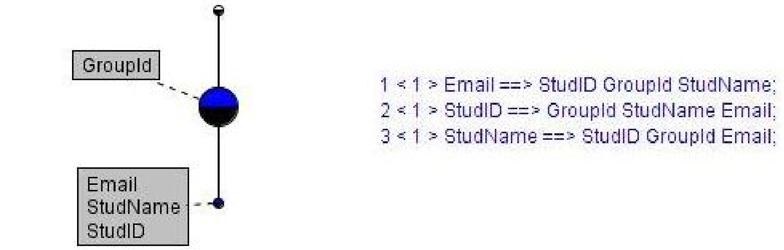
```
  <?xml version="1.0" encoding="UTF-8" ?>
- <ConceptualSystem>
    <Version MajorNumber="1" MinorNumber="0" />
  - <Contexts>
    - <Context Identifier="0" Type="Binary">
      - <Attributes>
        - <Attribute Identifier="0">
            <Name>StudID</Name>
          </Attribute>
        - <Attribute Identifier="1">
            <Name>GroupId</Name>
          </Attribute>
        - <Attribute Identifier="2">
            <Name>StudName</Name>
          </Attribute>
        - <Attribute Identifier="3">
            <Name>Email</Name>
          </Attribute>
        </Attributes>
      - <Objects>
        - <Object>
            <Name>(2, 531, A. Cole, ACole@email.co, 1, 531, R. White, RWhite@email.co)</Name>
          - <Intent>
              <HasAttribute AttributeIdentifier="1" />
            </Intent>
          </Object>
        - <Object>
            <Name>(3, 531, D. Lineker, DLineker@email., 1, 531, R. White, RWhite@email.co)</Name>
          - <Intent>
              <HasAttribute AttributeIdentifier="1" />
            </Intent>
          </Object>
        - <Object>
            <Name>(4, 531, J. Smith, JSmith@email.com, 1, 531, R. White, RWhite@email.co)</Name>
          - <Intent>
              <HasAttribute AttributeIdentifier="3" />
            </Intent>
          </Object>
        </Objects>
      </Context>
    </Contexts>
    <RecalculationPolicy Value="Clear" />
    <Lattices />
  </ConceptualSystem>
```

FIGURE 2. Cex file example for Students table

These attributes appear in the bottom of the FD lattice. `StudName` appears as candidate key, because all student name values were different in the analyzed table. We know that students may have the same name, but not the same ID, not the same Email address.

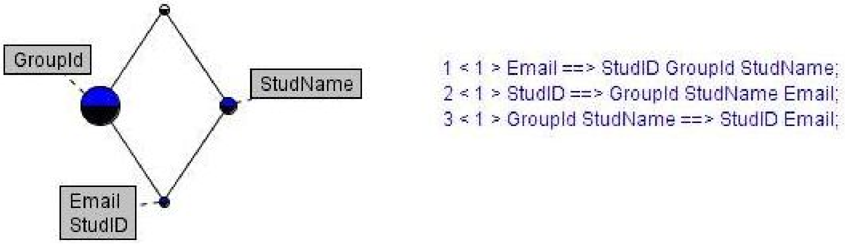
In Figure 4 we can see the results when in every group all students has different names than the `GroupID` and the `StudName` together form a composite key.

Figure 5 shows the FD lattice and functional dependencies obtained in case when we have same value for some student names in the same group and



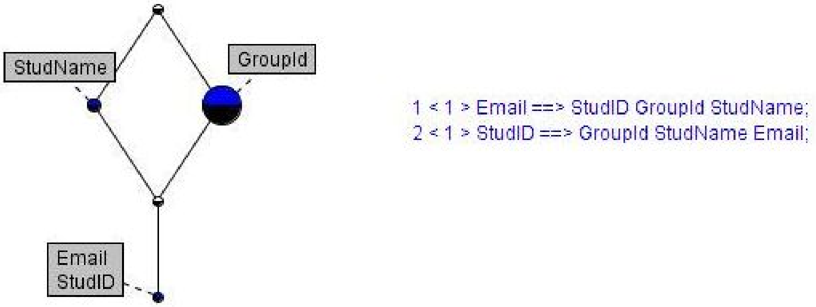
```
1 < 1 > Email ==> StudID GroupId StudName;
2 < 1 > StudID ==> GroupId StudName Email;
3 < 1 > StudName ==> StudID GroupId Email;
```

FIGURE 3. FD lattice and implications for the Students table with different values for student names



```
1 < 1 > Email ==> StudID GroupId StudName;
2 < 1 > StudID ==> GroupId StudName Email;
3 < 1 > GroupId StudName ==> StudID Email;
```

FIGURE 4. FD lattice and implications for the Students table when in every group all students has different names, but there are some students with same value for name



```
1 < 1 > Email ==> StudID GroupId StudName;
2 < 1 > StudID ==> GroupId StudName Email;
```

FIGURE 5. FD lattice and implications for the Students table with same value for some student names in the same group and in different groups too

in different groups too. This is the real case. We can see, that `StudName` doesn't appear in the bottom of the lattice, it isn't in the left hand side of any functional dependency, therefore it can't be a candidate key.

For determining whether an FD lattice is in BCNF, all non-trivial implications other than the ones whose left-hand side meets in the bottom node need to be checked. We can see, that every nontrivial functional dependency in the `Students` table has in its left hand side a superkey, therefore the table is in BCNF.

Example 2. Let `StudAdvisor` be a wrongly designed database table of a university database.

```
StudAdvisor [StudID,StudName,GroupID,StudEmail,SpecID,SpecName,
    Language,AdvisorId,TeacherName,TeacherEmail,TeacherPhone]
```

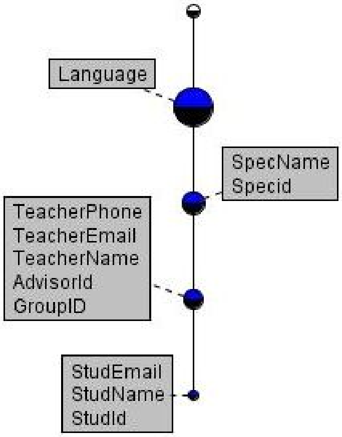
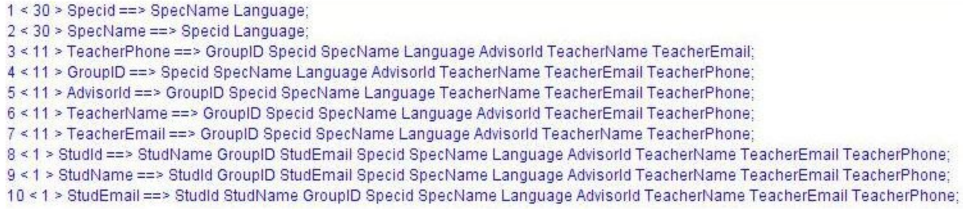


FIGURE 6. FD lattice for the StudAdvisor table

The FD lattice obtained for this table with software **FCAFuncDepMine** is in the Figure 6 and functional dependencies are in Figure 7. The concept `StudID, StudName, Email` is a subconcept of the concept `GroupID, AdvisorID, TeacherID, TeacherName, TeacherEmail,TeacherPhone,` which is the subconcept of the concept `SpecID, SpecName` and so on. The analysed data is not enough diversified, because every advisor has different name, every student has different name. The candidate keys of the table StudAdvisor are in the bottom of the FD lattice. But there are other functional dependencies, that has in its left hand side attributes, that are not in the bottom of the lattice: `SpecID, SpecName,TeacherPhone,`etc., therefore the table is not in BCNF.



```
1 < 30 > Specid ==> SpecName Language;
2 < 30 > SpecName ==> Specid Language;
3 < 11 > TeacherPhone ==> GroupID Specid SpecName Language Advisorld TeacherName TeacherEmail;
4 < 11 > GroupID ==> Specid SpecName Language Advisorld TeacherName TeacherEmail TeacherPhone;
5 < 11 > Advisorld ==> GroupID Specid SpecName Language TeacherName TeacherEmail TeacherPhone;
6 < 11 > TeacherName ==> GroupID Specid SpecName Language Advisorld TeacherEmail TeacherPhone;
7 < 11 > TeacherEmail ==> GroupID Specid SpecName Language Advisorld TeacherName TeacherPhone;
8 < 1 > StudId ==> StudName GroupID StudEmail Specid SpecName Language Advisorld TeacherName TeacherEmail TeacherPhone;
9 < 1 > StudName ==> StudId GroupID StudEmail Specid SpecName Language Advisorld TeacherName TeacherEmail TeacherPhone;
10 < 1 > StudEmail ==> StudId StudName GroupID Specid SpecName Language Advisorld TeacherName TeacherEmail TeacherPhone;
```

FIGURE 7. Functional dependencies in the StudAdvisor table

Introducing more varied data we get the FD lattice from the Figure 8 and functional dependencies from Figure 9.

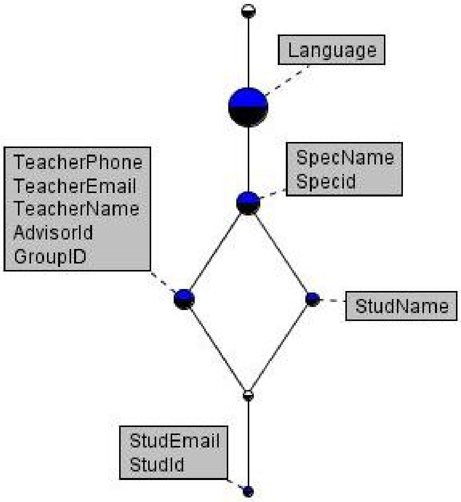
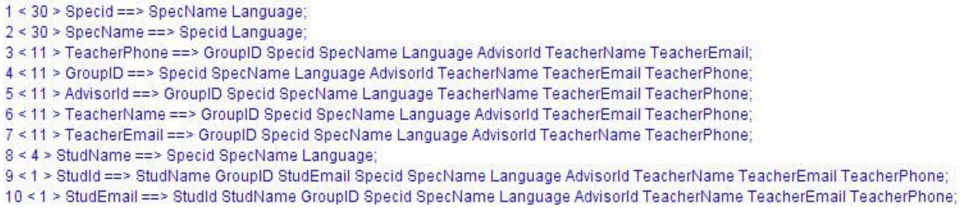


FIGURE 8. FD lattice for the StudAdvisor table with varied data

Having the functional dependencies, the candidate keys of the table can be seen and we can propose a decomposition of the table. From the last two FD's results, that every attribute is functionally dependent on `StudID,` as well as on `StudEmail,` therefore these two attributes are candidate keys. From the first two functional dependencies we can propose the next table:



```
1 < 30 > Specid ==> SpecName Language;
2 < 30 > SpecName ==> Specid Language;
3 < 11 > TeacherPhone ==> GroupID Specid SpecName Language AdvisorId TeacherName TeacherEmail;
4 < 11 > GroupID ==> Specid SpecName Language AdvisorId TeacherName TeacherEmail TeacherPhone;
5 < 11 > AdvisorId ==> GroupID Specid SpecName Language TeacherName TeacherEmail TeacherPhone;
6 < 11 > TeacherName ==> GroupID Specid SpecName Language AdvisorId TeacherEmail TeacherPhone;
7 < 11 > TeacherEmail ==> GroupID Specid SpecName Language AdvisorId TeacherName TeacherPhone;
8 < 4 > StudName ==> Specid SpecName Language;
9 < 1 > StudId ==> StudName GroupID StudEmail Specid SpecName Language AdvisorId TeacherName TeacherEmail TeacherPhone;
10 < 1 > StudEmail ==> StudId StudName GroupID Specid SpecName Language AdvisorId TeacherName TeacherEmail TeacherPhone;
```

Figure 9. Functional dependencies in the StudAdvisor table with varied data

```
Specializations [SpecId,SpecName,Language]
```

The FD's with number between 3 and 7 suggest the next table:

```
Advisors [GroupID,SpecId,AdvisorId,TeacherName,TeacherEmail,
                                       TeacherPhone]
```

The remaining attributes form the studied relation forms the next table:

```
Students [StudID,StudName,GroupID,StudEmail]
```

5. Conclusions and further research

We have proposed a software tool to detect functional dependencies in relational database tables. Our software constructs the power context family of the functional dependencies for a table, then Conexp gives the conceptual lattice and implications. Further we tend to analyze the functional dependencies obtained, to construct the closure of these implications and to give a correct database scheme by using an upgraded version of the proposed software, respectively.

References

[1] Abiteboul, S., Hull, R., Vianu, V.: Foundations of databases. Addison-Wesley, Reading - Menlo - New York (1995)

[2] Baixeries, J.: A formal concept analysis framework to mine functional dependencies, Proceedings of Mathematical Methods for Learning, (2004).

[3] Ganter, B., Wille, R.: Formal Concept Analysis. Mathematical Foundations. Springer, Berlin-Heidelberg-New York. (1999)

[4] Hereth, J.: Relational Scaling and Databases. Proceedings of the 10th International Conference on Conceptual Structures: Integration and Interfaces LNCS 2393, Springer Verlag (2002) 62–76

[5] Priss, U.: Establishing connections between Formal Concept Analysis and Relational Databases. Dau; Mugnier; Stumme (eds.), Common Semantics for Sharing Knowledge: Contributions to ICCS, (2005) 132–145

[6] Silberschatz, A., Korth, H. F.,Sudarshan, S.: Database System Concepts, McGraw-Hill, Fifth Edition, (2005)

[7] Wille, R. : Restructuring lattice theory: an approach based on hierarchies of concepts. In: I.Rival (ed.): Ordered sets. Reidel, Dordrecht-Boston, (1982) 445–470

[8] Janosi Rancz, K. T., Varga, V.: A method for mining functional dependencies in relational database design using FCA. Studia Universitatis "Babes-Bolyai" Cluj-Napoca, Informatica, vol. LIII, No. 1, (2008) 17–28.

[9] Yao, H., Hamilton, H. J.: Mining functional dependencies from data, Data Mining and Knowledge Discovery, Springer Netherlands, (2007)

[10] Serhiy A. Yevtushenko: System of data analysis "Concept Explorer". (In Russian). Proceedings of the 7th National Conference on Artificial Intelligence KII-2000, p. 127-134, Russia, 2000.

Sapientia University, Tg-Mures, Romania
E-mail address: tsuto@ms.sapientia.ro

Babes-Bolyai University, Cluj, Romania
E-mail address: ivarga@cs.ubbcluj.ro

Tg-Mures, Romania
E-mail address: puskasj@gmail.com

ON SUPERVISED AND SEMI-SUPERVISED k-NEAREST NEIGHBOR ALGORITHMS

ZALÁN BODÓ AND ZSOLT MINIER

Abstract. The k-nearest neighbor (kNN) is one of the simplest classification methods used in machine learning. Since the main component of kNN is a distance metric, kernelization of kNN is possible. In this paper kNN and semi-supervised kNN algorithms are empirically compared on two data sets (the USPS data set and a subset of the Reuters-21578 text categorization corpus). We use a soft version of the kNN algorithm to handle multi-label classification settings. Semi-supervision is performed by using data-dependent kernels.

1. Introduction

Suppose the training data is given in the form $D = \{(\mathbf{x}_i, y_i) \,|\, i = 1, 2, \ldots, \ell\} \cup \{\mathbf{x}_i \,|\, i = 1, 2, \ldots, u\}$ where the first set is called the labeled data, while the second is the unlabeled data set, which contains data drawn from the same distribution as the labeled points but there is no label information for them. Usually $\ell \ll u$. We will denote the size of the whole data set by $N = \ell + u$. The $\mathbf{x}_i \in X$ are called the *independent variables*, while the $y_i \in Y$ are the *dependent variables*, $X \subseteq \mathbb{R}^d$, $Y = \{1, 2, \ldots, K\}$, where K denotes the number of classes. In supervised classification we use only the first data set to "build" a classifier, while in semi-supervised classification we additionaly use the second data set that sometimes can improve predictions [11].

Semi-supervised learning (SSL) is a special case of classification; it is halfway between classification and clustering. The unlabeled data can be used to reveal important information. For example, suppose that in a text categorization problem the word "professor" turns out to be a good predictor for positive examples based on the labeled data. Then, if the unlabeled data shows

Received by the editors: September 15, 2008.

2000 *Mathematics Subject Classification.* 68T10, 45H05.

1998 *CR Categories and Descriptors.* I.2.6. [**Computing Methodologies**]: ARTIFICIAL INTELLIGENCE – *Learning.*

Key words and phrases. Supervised learning, Semi-supervised learning, k-nearest neighbors, Data-dependent kernels.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

that the words "professor" and "university" are correlated, then using both words the accuracy of the classifier is expected to improve. To understand how can one use the unlabeled data to improve prediction, consider the simplest semi-supervised learning method, called self-training or bootstrapping: train the classifier on the labeled examples, make predictions on the unlabeled data, add the points from the unlabeled set with the highest prediction confidence to the labeled set along with their predicted labels, and retrain the classifier. This procedure is usually repeated until convergence.

In order to be able to effectively use the unlabeled data to improve the system's performance some assumptions have to be conceived about the data. These are the smoothness assumption (SA), the cluster assumption (CA) and the manifold assumption (MA): SA says that points in a high density region should have similar labels, that is labels should change in low density regions, CA states that two points from the same cluster should have similar labels, while MA presumes that the data lies roughly on a low-dimensional manifold [7].

Most of the semi-supervised methods can be classified in the following four categories: generative models, low-density separation methods, graph-based methods and SSL methods based on change of representation. Methods belonging to the last category attempt to find some structure in the data which is better emphasized or better observable in the presence of the large unlabeled data set. These algorithms consist of the following following steps:

(1) Build the new representation – new distance, dot-product or kernel – of the learning examples.
(2) Use a supervised learning method to obtain the decision function based on the new representation obtained in the previous step.

Kernels referred in the first step are tools for non-linear extensions of linear algorithms like perceptron, linear support vector machines, kNN, etc. Kernel functions, or simply kernels were proposed for learning non-linear decision boundaries in 1964 in [1], but they became popular after the introduction of non-linear support vector machines (SVMs) in 1992 [6]. Kernel functions are symmetric functions of two variables, which return the "similarity" of two points in a high-dimensional space, without actually mapping the points to that space. More precisely, kernel functions return the dot product of two vectors in a so-called "feature" space:

$$k(\mathbf{x}, \mathbf{z}) = \phi(\mathbf{x})'\phi(\mathbf{z})$$

Any machine learning algorithm in which the input data appears only in the form of dot products can be extended to learn non-linear decision functions by simply using a positive semi-definite kernel function instead of the inner product of the vectors. This is called the "kernel trick". We call the matrix containing the dot products of the data points – i.e. the Gram matrix – the kernel matrix or simply the kernel. Whether we are referring to the kernel function or the kernel matrix by the expression “kernel” will be clear from the context.

Data-dependent kernels are similar to semi-supervised learning machines: the kernel function does not depend only on the two points in question, but in some form it makes use of the information contained in the whole learning data available. That is the value of $k(\mathbf{x}, \mathbf{z})$ with data set D_1 is not necessarily equal to the value of $k(\mathbf{x}, \mathbf{z})$ with data set D_2, however the kernel function – or more generally the kernel construction method – is the same. This can be formalized as

$$k(\mathbf{x}, \mathbf{z}; D_1) \Bumpeq k(\mathbf{x}, \mathbf{z}; D_2)$$

provided that the additional data sets are different, i.e. $D_1 \neq D_2$, where “$\Bumpeq$” means “not necessarily equal” and “;” stands for conditioning. In SSL methods with change of representation data-dependent kernels are used.

We will use data-dependent kernels to construct a semi-supervised version of the kNN classifier. These methods then will be empirically compared to another semi-supervised kNN method, the label propagation (LP) algorithm.

The paper is structured as follows. Section 2 introduces the kNN and the “soft” kNN classifier. In Section 3 we present label propagation for binary and multi-class cases. Label propagation can be viewed as a semi-supervised kNN technique. Section 4 describes the kernelization of the kNN classifier and shortly presents three data-dependent kernels, namely the ISOMAP, the multi-type and hierarchical cluster kernels, used in the experiments. The experiments and the obtained results are presented in Section 5. The paper ends with Section 6 discussing the results obtained in the experiments.

2. K-nearest neighbor algorithms

The k-nearest neighbor classification was introduced by Cover and Hart in [10]. The kNN classifier determines the label of an unseen point $\mathbf{x}$ by simple voting: it finds the k-nearest neighbors of $\mathbf{x}$ and assigns to it the winning label among these.

$$\widetilde{f}(\mathbf{x}) = \operatorname*{argmax}_{c=1,2,\ldots,K} \sum_{\mathbf{z} \in N_k(x)} \mathrm{sim}(\mathbf{z}, \mathbf{x}) \cdot \delta(c, f(\mathbf{z}))$$

where the function f assigns a label to a point, $N_k(\mathbf{x})$ denotes the set of k-nearest neighbors of $\mathbf{x}$, K is the number of classes, the function $\mathrm{sim}(\cdot, \cdot)$ returns the similarity of two examples, and $\delta(a, b) = 1$ if $a = b$, 0 otherwise. The function $\mathrm{sim}(\cdot, \cdot)$ is used to give different weights for different points. One choice could be to use some distance metric $d(\cdot, \cdot)$ with the property of

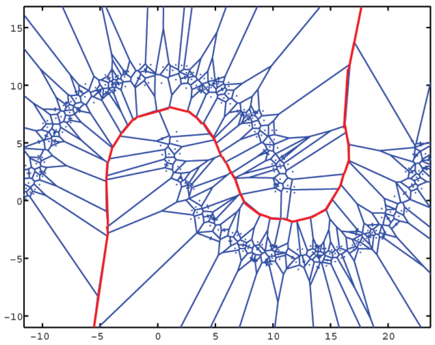


FIGURE 1. Figure showing the 1NN decision boundaries for the two-moons data set.

assigning a lower value to nearby points and a higher value to farther points to $\mathbf{x}$. Then one can choose for example

$$\mathrm{sim}(\mathbf{x},\mathbf{z}) = \frac{1}{g(d(\mathbf{x},\mathbf{z}))}$$

where $g(\cdot)$ is an adequate function. If the constant function $\mathrm{sim}(\mathbf{x},\mathbf{z}) = 1$ is chosen, we arrive to simple kNN, where all the neighbors have the same influence on the predicted label.

In order to work efficiently implement the kNN method, no explicit form of the inductive classifier is built, since representing and storing the decision boundaries can become very complex. On Figure 1 the decision boundaries of a 1NN classifier are shown; we used the popular "two-moons" data set for this illustration. Here we have two classes: the positive class is represented by the upper crescent, while the points of the negative class lie in the lower crescent. The polygons represent the area in which an unseen point gets the label of the point which "owns" the respective cell. The red curve shows the decision boundary between the classes.

2.1. **Soft kNN.** In the soft version of the kNN we average the labels of the surrounding points. That is the prediction function becomes

$$f(\mathbf{x}) = \frac{1}{\sum_{\mathbf{z}\in N_k(\mathbf{x})} W_{\mathbf{zx}}} \sum_{\mathbf{z}\in N_k(\mathbf{x})} W_{\mathbf{zx}} f(\mathbf{z})$$

where $W_{\mathbf{zx}}$ denotes the similarity between $\mathbf{z}$ and $\mathbf{x}$. In case of binary classification, that is $Y = \{-1, 1\}$ or $Y = \{0, 1\}$ we use thresholding after computing the prediction by the above formula, e.g. using the value 0 or 0.5 for the threshold. Thus we arrive to the same decision function.

3. Label propagation

Label propagation was introduced in [23] for semi-supervised learning. It is a transductive graph-based semi-supervised learning technique, i.e. the labels are determined only in the desired points. We can call LP a semi-supervised kNN algorithm, because the label of a point is determined considering only the labels of its neighbors. The only and considerable difference between kNN and LP is that while in LP the labels propagate through the neighbors, and the label of an unseen point depends on the labels of the other unseen/unlabeled points too, the labels are static in kNN and only the labeled points count.

For binary class learning consider the vector $\mathbf{f} \in \{-1, 1\}^N$ of class labels, where $N = \ell + u$. Then the energy/cost function to be minimized is the following

$$E_1(\mathbf{f}) = \frac{1}{2} \sum_{i,j=1}^{N} W_{ij}(f_i - f_j)^2 \tag{1}$$

where f_i, $i = 1, \ldots, \ell$ is fixed according to the labeled training points. If $\mathbf{f}$ is divided as $\begin{bmatrix} \mathbf{f}_L \\ \mathbf{f}_U \end{bmatrix}$ where $\mathbf{f}_L$ and $\mathbf{f}_U$ denote the parts corresponding to the labeled and unlabeled examples, then the optimization problem can be written as

$$\min_{\mathbf{f}_U} E_1(\mathbf{f})$$

It is easy to check that

$$\begin{aligned} E_1(\mathbf{f}) &= \sum_{ij} W_{ij} f_i^2 - \sum_{ij} W_{ij} f_i f_j \\ &= \mathbf{f}'\mathbf{D}\mathbf{f} - \mathbf{f}'\mathbf{W}\mathbf{f} = \mathbf{f}'\mathbf{L}\mathbf{f} \end{aligned}$$

where $\mathbf{L} = \mathbf{D} - \mathbf{W}$ is the graph Laplacian [9] of the similarity matrix of the points. For the sake of simplicity we divide the matrices into the following blocks:

$$\mathbf{W} = \begin{bmatrix} \mathbf{W}_{LL} & \mathbf{W}_{LU} \\ \mathbf{W}_{UL} & \mathbf{W}_{UU} \end{bmatrix}; \quad \mathbf{D} = \begin{bmatrix} \mathbf{D}_L & \mathbf{0} \\ \mathbf{0} & \mathbf{D}_U \end{bmatrix}$$

$$\mathbf{L} = \begin{bmatrix} \mathbf{L}_{LL} & \mathbf{L}_{LU} \\ \mathbf{L}_{UL} & \mathbf{L}_{UU} \end{bmatrix}; \quad \mathbf{P} = \begin{bmatrix} \mathbf{P}_{LL} & \mathbf{P}_{LU} \\ \mathbf{P}_{UL} & \mathbf{P}_{UU} \end{bmatrix}$$

We want to minimize $E_1(\mathbf{f})$, therefore we calculate its derivative and set to zero. Thus we obtain

$$\mathbf{f}_U = -\mathbf{L}_{UU}^{-1} \cdot \mathbf{L}_{UL} \cdot \mathbf{f}_L \tag{2}$$

or equivalently $(\mathbf{I} - \mathbf{D}_U^{-1}\mathbf{W}_{UU})^{-1}\mathbf{D}_U^{-1} \cdot \mathbf{W}_{UL} \cdot \mathbf{f}_L = (\mathbf{I} - \mathbf{P}_{UU})^{-1}\mathbf{P}_{UL} \cdot \mathbf{f}_L$.

The above energy function can be simply modified for the multi-class, multi-label case:

$$E_2(\mathbf{f}) = \frac{1}{2}\sum_{i,j=1}^{N} W_{ij}\|\mathbf{f}_{i\cdot} - \mathbf{f}_{j\cdot}\|_2^2$$

where now $\mathbf{f} \in \{0,1\}^{N\times K}$. One can observe that $E_2(\mathbf{f}) = \operatorname{tr}(\mathbf{f}'\mathbf{L}\mathbf{f})$. If we decompose $\mathbf{f}$ into column vectors

$$\mathbf{f} = \begin{bmatrix} \mathbf{f}_{\cdot 1} & \mathbf{f}_{\cdot 2} & \cdots & \mathbf{f}_{\cdot K} \end{bmatrix}$$

then the problem can be rewritten as K independent constrained optimization problems involving vectors of size $N \times 1$.

$$\begin{aligned}
\mathbf{f}'\mathbf{L}\mathbf{f} &= \begin{bmatrix} \mathbf{f}_{\cdot 1}{}' \\ \mathbf{f}_{\cdot 2}{}' \\ \vdots \\ \mathbf{f}_{\cdot K}{}' \end{bmatrix} \cdot \mathbf{L} \cdot \begin{bmatrix} \mathbf{f}_{\cdot 1} & \mathbf{f}_{\cdot 2} & \cdots & \mathbf{f}^{\cdot K} \end{bmatrix} \\
&= \begin{bmatrix} \mathbf{f}_{\cdot 1}{}'\mathbf{L}\mathbf{f}_{\cdot 1} & \cdot & \cdots & \cdot \\ \cdot & \mathbf{f}_{\cdot 2}{}'\mathbf{L}\mathbf{f}_{\cdot 2} & \cdots & \cdot \\ \vdots & \vdots & \ddots & \vdots \\ \cdot & \cdot & \cdots & \mathbf{f}_{\cdot K}{}'\mathbf{L}\mathbf{f}_{\cdot K} \end{bmatrix}
\end{aligned}$$

from which it follows that

$$\operatorname{tr}(\mathbf{f}'\mathbf{L}\mathbf{f}) = \operatorname{tr}(\mathbf{f}_{\cdot 1}{}'\mathbf{L}\mathbf{f}_{\cdot 1}) + \ldots + \operatorname{tr}(\mathbf{f}_{\cdot K}{}'\mathbf{L}\mathbf{f}_{\cdot K})$$

that is we can minize now $\mathbf{f}_{\cdot i}{}'\mathbf{L}\mathbf{f}_{\cdot i}$ with respect to $(\mathbf{f}_U)_{\cdot i}$, $i = 1, 2, \ldots, K$ and from these solutions the solution of the original problem can be built. In our notation used above $\mathbf{f}_{\cdot i}$ denotes the ith column, while $\mathbf{f}_{j\cdot}$ denotes the jth row of $\mathbf{f}$.

By calculating the derivative of $E_2(\mathbf{f})$ with respect to $\mathbf{f}_U$, we arrive to the same formula as (2), but $\mathbf{f}$ is now a matrix, not a vector.

The iterative solution for LP is composed of the following steps:

(1) Compute $\mathbf{W}$ and $\mathbf{D}$.
(2) $i = 0$; Initialize $\mathbf{f}_U^{(i)}$.
(3) $\mathbf{f}^{(i+1)} = \mathbf{D}^{-1}\mathbf{W}\mathbf{f}^{(i)}$.
(4) Clamp the labeled data, $\mathbf{f}_L^{(i+1)} = \mathbf{f}_L$.
(5) $i = i + 1$; Unless convergence go to step 3.

Convergence means that the difference between the class assignment matrices $\mathbf{f}$ obtained in two consecutive steps drops below a predefined threshold. The value of the threshold greatly influences the running time of the algorithm. The difference between consecutive solutions can be measured by the Frobenius matrix norm. The convergence of the above algorithm is proven in [23]. On

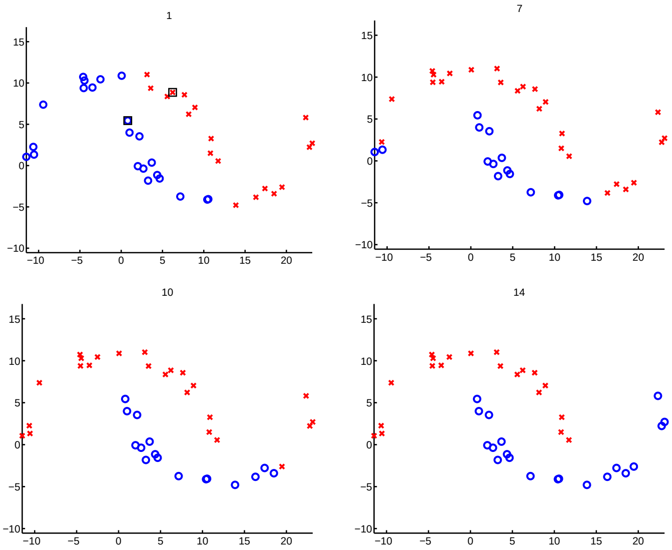


FIGURE 2. The propagation of labels (iteration 1, 7, 10 and 14). At the beginning – step 0, not shown separately here – only the 2 points put in the black squared frames are labeled.

Figure 2 the propagation of labels is illustrated on the two-moons data set, starting from only two labeled points.

If we decompose $\mathbf{f}$, we arrive to the simpler formula

$$\mathbf{f}_U^{(i+1)} = \mathbf{D}_U^{-1}\mathbf{W}_{UL}\mathbf{f}_L + \mathbf{D}_U^{-1}\mathbf{W}_{UU}\mathbf{f}_U^{(i)}$$

or by using the notation $\mathbf{A} = \mathbf{P}_{UL}\mathbf{f}_L = \mathbf{D}_U^{-1}\mathbf{W}_{UL}\mathbf{f}_L$ and $\mathbf{P}_{UU} = \mathbf{D}_U^{-1}\mathbf{W}_{UU}$, we obtain the update formula $\mathbf{f}_U^{(i+1)} = \mathbf{A} + \mathbf{P}_{UU}\mathbf{f}_U^{(i)}$, which can be included into the algorithm by replacing steps 3 and 4.

Consider now the the case of binary classification. Given the solution $\mathbf{f}$ according to the update formula we can write that

$$\begin{aligned} f_i &= (\mathbf{D}^{-1}\mathbf{W})_{i\cdot} \cdot \mathbf{f} \\ &= \frac{1}{\sum_{j=1}^{N} W_{ij}} \cdot \sum_{j=1}^{N} W_{ij} f_j \end{aligned}$$

that is the label of a point is equal to the weighted average of the other points' class labels. When the Gaussian kernel/similarity function is used (which is one of the most common choices in practice), which assigns an exponentially decreasing similarity to the more distant points, those weights can be considered to be equal to zero. Thus we get a kNN-like algorithm, where k changes dynamically, so this is rather an εNN algorithm, where ε denotes a threshold above which similarity is considered to be 0.

In the multi-class case we can write

$$\mathbf{f}_{\cdot j} = \mathbf{P}\mathbf{f}_{\cdot j}$$

that is

$$f_{ij} = \mathbf{P}_{i\cdot}\mathbf{f}_{\cdot j}$$

which is equivalent to

$$f_{ij} = \frac{1}{\sum_{k=1}^{N} W_{ik}} \sum_{k=1}^{N} W_{ik} f_{kj}$$

for all $i = 1, 2, \ldots, N$ and $j = 1, 2, \ldots, K$.

Label propagation can be considered as a constrained mincut problem, which is a very popular clustering technique [3]. Since graph mincut can be written as $(1/4) \cdot \mathbf{f}'\mathbf{L}\mathbf{f}$, where $\mathbf{f} \in \{-1, 1\}^N$, therefore label propagation is equivalent to searching for a mincut of the data graph, given that the labeled points are fixed.

4. Semi-supervised kNN

The k-nearest neighbor algorithm determines labels based on the labels of the nearest points. "Nearest" is defined using some metric, in the original formulation taking the Euclidean metric. The Euclidean distance can be rewritten in form of dot products as

$$\begin{aligned} \|\mathbf{x} - \mathbf{z}\|_2^2 &= \langle \mathbf{x}, \mathbf{x} \rangle + \langle \mathbf{z}, \mathbf{z} \rangle - 2 \cdot \langle \mathbf{x}, \mathbf{z} \rangle \\ &= k(\mathbf{x}, \mathbf{x}) + k(\mathbf{z}, \mathbf{z}) - 2 \cdot k(\mathbf{x}, \mathbf{z}) \end{aligned}$$

where $k(\cdot, \cdot)$ denotes in this case the linear kernel $k(\mathbf{x}, \mathbf{z}) = \langle \mathbf{x}, \mathbf{z} \rangle = \mathbf{x}'\mathbf{z}$. Using the kernel trick, this can be replaced by any other positive semi-definite kernel. Thus the points are implicitly mapped to a – possibly higher dimensional – space, where their dot product is given by the kernel function $k(\cdot, \cdot)$.

In multi-label learning let us denote the decision function as $f : X \rightarrow [0, 1]^K$. For hard classification we can set a threshold, for example 0.5, but for soft classification we use the values of the output vector as class membership probabilities. The decision function is expressed in the same way as in the case of binary classification,

$$f(\mathbf{x}) = \frac{1}{\sum_{\mathbf{z} \in N_k(\mathbf{x})} W_{\mathbf{z}\mathbf{x}}} \sum_{\mathbf{z} \in N_k(\mathbf{x})} W_{\mathbf{z}\mathbf{x}} f(\mathbf{z})$$

with the difference that now the output is a $K \times 1$ vector, instead of a scalar value.

4.1. **Data-dependent kernels.** There are other methods to determine the nearest neighbors of a point by using the following data-dependent kernels.

4.1.1. *The ISOMAP kernel.* ISOMAP (ISOmetric feature MAPping) was introduced in [20] for dimensionality reduction using the manifold assumption. The ISOMAP kernel is defined as

$$\mathbf{K}_{\text{isomap}} = -(1/2)\mathbf{J}\mathbf{G}^{(2)}\mathbf{J}$$

where $\mathbf{G}^{(2)}$ contains the squared graph distances (shortest paths in the graph whose vertices are the original data points and the edges are among the nearest neighbors of each point) and $\mathbf{J}$ is the centering matrix, $\mathbf{J} = \mathbf{I} - \frac{1}{N} \cdot \mathbf{1} \cdot \mathbf{1}'$, $\mathbf{I}$ is the identity matrix, and $\mathbf{1}$ is the $N \times 1$ vector of 1's. $\mathbf{G}^{(2)}$ is not necessarily positive semi-definite so neither is the ISOMAP kernel. But since only the largest eigenvalues and the corresponding eigenvectors are important, we proceed in the following way. The kernel matrix can be decomposed into $\mathbf{U}\mathbf{S}\mathbf{U}'$, where $\mathbf{U}$ contains the eigenvectors, while the diagonal matrix $\mathbf{S}$ holds the eigenvalues of the decomposed matrix [15, p. 393]. Then the ISOMAP kernel we will use is $\mathbf{K}_{\text{isomap}} = \mathbf{U}\widetilde{\mathbf{S}}\mathbf{U}'$, where $\widetilde{\mathbf{S}}$ is the diagonal matrix of the eigenvalues in which each negative eigenvalue was set to zero.

Informally, the ISOMAP kernel maps the points to the space, where their pointwise distances equal to the shortest path distances on the data graph in the input space. If the points are centered at each dimension, then $-(1/2) \cdot \mathbf{J}\mathbf{G}^{(2)}\mathbf{J}$ is equal to the dot products of the vectors mapped to the abovementioned space [5, p. 262].

4.1.2. *The multi-type cluser kernel.* In [8] the authors develop a cluster kernel which connects several techniques together like spectral clustering, kernel PCA and random walks. The proposed cluster kernel is built following the steps described below:

(1) Compute the Gaussian kernel and store in matrix $\mathbf{W}$.
(2) Symmetrically normalize $\mathbf{W}$, that is let $\mathbf{L} = \mathbf{D}^{-1/2}\mathbf{W}\mathbf{D}^{-1/2}$, where $\mathbf{D} = \operatorname{diag}(\mathbf{W} \cdot \mathbf{1})$, and compute its eigendecomposition, $\mathbf{L} = \mathbf{U}\mathbf{\Sigma}\mathbf{U}'$.
(3) Determine a transfer function $\varphi(\cdot)$ for transforming the eigenvalues, $\widetilde{\lambda}_i = \varphi(\lambda_i)$, and construct $\widetilde{\mathbf{L}} = \mathbf{U}\widetilde{\mathbf{\Sigma}}\mathbf{U}'$, where $\widetilde{\mathbf{\Sigma}}$ contains the transformed eigenvalues on the diagonal.
(4) Let $\widetilde{\mathbf{D}}$ be a diagonal matrix with diagonal elements $D_{ii} = 1/\widetilde{L}_{ii}$, and compute $\mathbf{K} = \widetilde{\mathbf{D}}^{1/2}\widetilde{\mathbf{L}}\widetilde{\mathbf{D}}^{1/2}$.

The kernel type depends on the chosen transfer function. We discuss here three types of transfer functions as in [8]. In the following let λ_i represent the eigenvalues of matrix $\mathbf{L}$ defined in step (2).

The step transfer function is defined as $\varphi(\lambda_i) = 1$ if $\lambda_i \geq \lambda_{\text{cut}}$ and 0 otherwise, where λ_{cut} is a predetermined cutting threshold for the eigenvalues. This results in the dot product matrix of the points in the spectral clustering representation [17].

The linear step transfer function simply cuts off the eigenvalues which are smaller that a predetermined threshold, $\varphi(\lambda_i) = \lambda_i$ if $\lambda_i \geq \lambda_{\text{cut}}$, otherwise equals 0. Without normalization, that is with $\mathbf{D} = \mathbf{I}$ and similarly $\widetilde{\mathbf{D}} = \mathbf{I}$, the method would be equal to the data representation in KPCA space [18], since in that case we simply cut off the least significant directions to obtain a low rank representation of $\mathbf{L}$.

The polynomial transfer function is defined as $\varphi(\lambda_i) = \lambda_i^t$, where $t \in \mathbb{N}$ or $t \in \mathbb{R}$ is a parameter. Thus the final kernel can be written as

$$\widetilde{\mathbf{K}} = \widetilde{\mathbf{D}}^{1/2}\mathbf{D}^{1/2}\left(\mathbf{D}^{-1}\mathbf{W}\right)^t\mathbf{D}^{-1/2}\widetilde{\mathbf{D}}^{1/2} \tag{3}$$

where $\mathbf{D}^{-1}\mathbf{W} = \mathbf{P}$ is the probability transition matrix, where P_{ij} is the probability of going from point i to point j. This is called the random walk kernel, since (3) can be considered as a symmetrized version of the transition probability matrix $\mathbf{P}$.

4.1.3. *The hierarchical cluster kernel.* Hierarchical cluster kernels for supervised and semi-supervised learning were introduced in [4]. We used hierarchical clustering techniques to build ultrametric trees [21]. Then we used the distances induced by the clustering method to build a kernel for supervised and semi-supervised methods. The hierarchical cluster kernels are generalizations of the connectivity kernel [14].

The algorithm has the following steps:

-2. Determine the k nearest neighbors or an ϵ-neighborhood of each point and take all the distances to other points equal to zero.

-1. Compute shortest paths for every pair of points – using for example Dijkstra's algorithm.

0. Use these distances in clustering for the pointwise distance $d(\cdot,\cdot)$ in single, complete and average linkages distances [16, Chapter 3], [13, Chapter 4].

1. Perform an agglomerative clustering on the labeled and unlabeled data using one of the above-mentioned linkage distances.

2. Define matrix $\mathbf{M}$ with entries M_{ij} = linkage distance in the resulting ultrametric tree at the lowest common subsumer of i and j; $M_{ii} = 0$, $\forall i$.

3. Define the kernel matrix as $\mathbf{K} = -\frac{1}{2}\mathbf{JMJ}$.

method	**accuracy**
kNN (linear, Gaussian)	94.00 $(k_{kNN} = 1)$
LP (linear, Gaussian)	80.29 $(1/(2 \cdot \sigma^2) = 0.05)$
kNN + ISOMAP	95.71 $(k_{isomap} = 5, k_{kNN} = 5)$
kNN + mt. cluster kernel	95.00 (linstep, $1/(2 \cdot \sigma^2) = 0.05$, $\lambda_{cut} = 0.1$, $k_{kNN} = 4$)
kNN + h. cluster kernel	**96.64** (average linkage, $k_{isomap} = 4$, $k_{kNN} = 3$)

Table 1. Accuracy results obtained for the modified USPS handwritten digits data set.

The first three steps of the method – numbered with -2, -1 and 0 – are optional; they can be applied if the semi-supervised manifold assumption is expected to hold.

5. Experiments

In the experiments we compared the methods of kNN, kNN with data-dependent kernels and label propagation. We used the data-dependent kernels presented in the previous section: the ISOMAP, the multi-type and the hierarchical clusters kernels. The methods were tested on two data sets: a modified version of the USPS *(United States Postal Service)* handwritten digits data set and a subset of Reuters-21578 [12]. The USPS data set is derived from the original USPS set of handwritten digits[1]. The set is imbalanced, since it was created by putting the digits 2 and 5 into one class, while the rest is in the second class. 150 images belong to each of the ten digits. Because the set was used as a benchmark data set for the algorithms presented in the book [7], it was obscured using a simple algorithm to prevent recognizing the origin of the data. The set contains 1500 examples and 2 × 12 splits of the data, where the first 12 splits contain 10 labeled, and 1490 unlabeled, while the second 12 splits contain 100 labeled and 1400 unlabeled examples. We used only the first split with 100 labeled points[2].

[1] `http://archive.ics.uci.edu/ml/datasets/`

[2] The modified USPS data set can de downloaded from `http://www.kyb.tuebingen.mpg.de/ssl-book/benchmarks.html`

method	**microBEP/macroBEP**
kNN (linear)	89.60 / 89.22 ($k_{kNN} = 14$)
kNN (Gaussian)	90.05 / 90.02 ($k_{kNN} = 17$)
LP (linear)	88.44 / 88.39
LP (Gaussian)	89.81 / 90.33 ($1/(2 \cdot \sigma^2) = 1$)
kNN + ISOMAP	88.37 / 87.99 ($k_{isomap} = 16, k_{kNN} = 16$)
kNN + mt. cluster kernel	**91.12 / 91.04** (linstep, $1/(2 \cdot \sigma^2) = 1$, $\lambda_{cut} = 0.01$, $k_{kNN} = 16$)
kNN + h. cluster kernel	87.46 / 87.42 (average linkage, $k_{isomap} = 3$, $k_{kNN} = 7$)

Table 2. Micro- and macro-averaged precision–recall breakeven point results for the modified Reuters-21578 text categorization corpus.

We also modified the Reuters-21578 text categorization corpus in order to make it smaller and to balance the categories. The original corpus[3] contains 12 902 documents – 9603 for training and 3299 for testing – categorized into 90 classes. We kept the following 10 categories: alum, barley, bop, carcass, cocoa, coffee, copper, cotton, cpi, dlr. Thus we were left with 626 training and 229 test documents. For representing documents we used the bag-of-words document representation [2, Chapter 2] with tfidf weighting [2, p. 29]. We stemmed the words of the documents using the Porter stemmer, and selected 500 terms with the χ^2 feature selection technique [22, 19].

For evaluation we used accuracy for the USPS data set and precision–recall breakeven point for the Reuters corpus.

The results are shown on Tables 1 and 2. For each method we searched for the parameters that result in the best performance on the test data (these parameters are shown in brackets). The best results for each data set were formatted with boldface.

[3] The 90 and 115-categories version of Reuters can be downloaded from the homepage of Alessandro Moschitti, `http://dit.unitn.it/~moschitt/corpora.htm`

6. Discussion

In this paper we compared kNN and some semi-supervised kNN methods on two data sets. We saw that semi-supervised kNN methods can outperform conventional kNN: for the USPS data set we obtained an improvement of 2.64% with the average linkage hierarchical kernel. However label propagation showed a very low performance on this data set. We considered the values between 1 and 5 for k, which means that the classes are separated quite well. For the Reuters corpus we found the multi-type cluster kernel with linear step function to provide the best performance, but the improvement was not so significant as for the USPS data set. This also shows that KPCA is able to remove irrelevant dimensions from the bag-of-words representation of the Reuters corpus. Label propagation with Gaussian kernel showed only little performance improvement to linear kNN. We see however that the best values of k for kNN were between 7 and 17, which could imply the intertanglement of the documents, that is one should search for a better initial representation than bag-of-words.

Acknowledgement

We acknowledge the support of the grants CNCSIS/TD-35 and CNCSIS/TD-77 by the Romanian Ministry of Education and Research.

References

[1] A. Aizerman, E. M. Braverman, and L. I. Rozonoer. Theoretical foundations of the potential function method in pattern recognition learning. *Automation and Remote Control*, 25:821–837, 1964.

[2] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval.* Addison-Wesley, 1999.

[3] Tijl De Bie. *Semi-Supervised Learning Based On Kernel Methods And Graph Cut Algorithms.* PhD thesis, Katholieke Universiteit Leuven, Kasteelpark Arenberg 10, 3001 Leuven (Heverlee), 2005.

[4] Zalán Bodó. Hierarchical cluster kernels for supervised and semi-supervised learning. In *Proceedings of the 4nd International Conference on Intelligent Computer Communication and Processing*, pages 9–16. IEEE, August 2008.

[5] Ingwer Borg and Patrick J. F. Groenen. *Modern multidimensional scaling, 2nd edition.* Springer-Verlag, New York, 2005.

[6] B. E. Boser, I. Guyon, and V. N. Vapnik. A training algorithm for optimal margin classifiers. *Computational Learning Theory*, 5:144–152, 1992.

[7] Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien. *Semi-Supervised Learning.* MIT Press, September 2006. Web page: http://www.kyb.tuebingen.mpg.de/ssl-book/.

[8] Olivier Chapelle, Jason Weston, and Bernhard Schölkopf. Cluster kernels for semi-supervised learning. In Suzanna Becker, Sebastian Thrun, and Klaus Obermayer, editors, *NIPS*, pages 585–592. MIT Press, 2002.

[9] Chung. Spectral graph theory (reprinted with corrections). In *CBMS: Conference Board of the Mathematical Sciences, Regional Conference Series*, 1997.

[10] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Transactions on Information Theory*, IT-13, 1967.

[11] Fabio G. Cozman and Ira Cohen. Risks of semi-supervised learning. In Olivier Chapelle, Bernhard Schölkopf, and Alexander Zien, editors, *Semi-Supervised Learning*, chapter 4, pages 55–70. MIT Press, 2006.

[12] Franca Debole and Fabrizio Sebastiani. An analysis of the relative hardness of reuters-21578 subsets. *Journal of the American Society for Information Science and Technology*, 56:971–974, 2004.

[13] Richard Duda, Peter Hart, and David Stork. *Pattern Classification.* John Wiley and Sons, 2001. 0-471-05669-3.

[14] Bernd Fischer, Volker Roth, and Joachim M. Buhmann. Clustering with the connectivity kernel. In Sebastian Thrun, Lawrence K. Saul, and Bernhard Schölkopf, editors, *NIPS*. MIT Press, 2003.

[15] Gene H. Golub and Charles F. Van Loan. *Matrix Computations, 3nd Edition.* The Johns Hopkins University Press, Baltimore, MD, 1996.

[16] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data.* Prentice-Hall, 1988.

[17] Andrew Y. Ng, Michael Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In T. G. Dietterich, S. Becker, and Zoubin Ghahramani, editors, *Advances in Neural Information Processing Systems 14*, Cambridge, MA, 2002. MIT Press.

[18] Bernhard Schölkopf, Alexander J. Smola, and Klaus-Robert Müller. Kernel principal component analysis. *Advances in kernel methods: support vector learning*, pages 327–352, 1999.

[19] Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Computing Surveys*, 34(1):1–47, 2002.

[20] J. B. Tenenbaum, V. de Silva, and J. C. Langford. A global geometric framework for nonlinear dimensionality reduction. *Science*, 290(5500):2319–2323, December 2000.

[21] Bang Ye Wu and Kun-Mao Chao. *Spanning Trees and Optimization Problems.* Chapman and Hall/CRC, Boca Raton, Florida, 2004.

[22] Yiming Yang and Jan O. Pedersen. A comparative study on feature selection in text categorization. In *International Conference on Machine Learning*, pages 412–420, 1997.

[23] Xiaojin Zhu and Zoubin Ghahramani. Learning from labeled and unlabeled data with label propagation. Technical Report CMU-CALD-02-107, Carnegie Mellon University, 2002.

Department of Computer Science, Babeş–Bolyai University, Mihail Kogălniceanu nr. 1, RO-400084 Cluj-Napoca

E-mail address: {zbodo,minier}@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

RECURSIVE AND DYNAMIC STRUCTURES IN GENERIC PROGRAMMING

ÁBEL SINKOVICS AND ZOLTÁN PORKOLÁB

Abstract. Generic programming – an emerging new programming paradigm – best known from Standard Template Library as an essential part of C++ provides an opportunity to extend libraries in an efficient way. Both containers (abstract data structures) and algorithms working on them could be independently designed and implemented at O(1) cost. Unfortunately, smoothless cooperation of generic programming and object orientation is still an open problem. In this paper we will focus on reducing development and maintenance costs of systems using generative programming with recursive data structures to avoid multiple implementations of the components. For cases when separate implementation of algorithms can't be avoided we provide a system protecting us against changing existing code during extension. Providing such a design is not a trivial problem using currently available tools. We will show a possible solution using a graphic library to demonstrate the problem and our solution with.

1. Introduction

In software development, working with recursive data structures is an ubiquitous problem. Graphic editors, web browsers, office software, etc. have to work with complex systems with sets of different (including recursive) component types. These software have to deal with algorithms operating on the components. The longest and most time and money consuming part of a software system's life is maintenance, and with poor design it is hard to maintain and extension is always expensive. Extending the system with new components, all algorithms have to be implemented for them. Extending the system

Received by the editors: September 14, 2008.

2000 *Mathematics Subject Classification.* 68N15, 68N19.

1998 *CR Categories and Descriptors.* D.2 [**Software Engineering**]: D.2.3 Coding tools and techniques – *Object-oriented and generative programming* D.3 [**Programming Languages**]: D.3.2 Language Classification – *C++*;

Key words and phrases. Generic programming, Software engineering, Expression problem, C++.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

with new algorithms, they have to be implemented for every component. Independent extension could make development and maintenance faster, more flexible, and cheaper. In this paper we will examine commonly used design patterns and will introduce a new one supporting independent development and extension in several cases.

The rest of the paper is organized as follows: In this section we present a practical example of the problem after which we analyse currently existing solutions in section 2. We present our solution in section 3 and use it to solve the practical example in section 3.3. We analyse runtime performance of our solution in section 3.4 and finally we summarize our results in section 4.

As a motivating example we chose a graphic application since it has every feature required to demonstrate the problem. Our sample graphic application supports different shapes and transformations. It is not uncommon to define such a system with at least 20 different shapes and 50 transformations. One of the suggested design methods [3] [10] [12] is using the Interpreter design pattern [15], which indicates to create an abstract base class called `Shape`, inherit all shapes from it and implement the transformations using virtual functions. The other suggested method [3] [10] [12] is using the Visitor design pattern [15], which indicates to create an abstract base class called `Visitor` for transformations with a virtual function for every shape type. For example if the system has `Oval` shape, `Visitor` should have a virtual function called `visitOval` accepting `Oval` shapes. Every transformation should be implemented in a different class inherited from `Visitor` and implement the virtual functions.

The example above refers to a well-known scaling problem of object-oriented library design. Philip Wadler called it the expression problem on the Java-Genericity mailing list for the first time in 1998 [13]. Given a set of recursive data structures and a set of operations operating on the data structures and a design is required which supports extension of both data types and operations independently. Extension (or modification) of one set should not affect the other and should not imply changes in existing code.

Zenger and Odersky presented a list of requirements [3] for a solution which we have extended with an extra item. Our main goal is to find a design which **(1)** is extensible with algorithms (transformations in the example) and data types (shapes in the example). **(2)** is independently extensible. Extensions shouldn't imply changes to previously written code. None of the data types should be changed because of writing a new operation and none of the operations should be changed because of creating a new data type. **(3)** is type safe. Validity of the operations are check at compile time eliminating runtime errors which can remain untested in practice causing embarrassing and commonly expensive issues in production. **(4)** is effective. When types are available at compile-time, the efficiency of the compiled code should be similar to hand-written code allowing automatic optimalisation of the code. **(5)** supports separate compilation. Different components of the system (different data types and algorithms) can be compiled independently from each other. **(6)** supports the reduction of the number of implementations where possible by using generic algorithms to describe similar implementations. This is our extension to Zenger's and Odersky's list. Generic algorithms – first introduced in Ada – can support data types which are written independently of the algorithm, maybe later after the creation of the algorithm.

2. Existing solutions

Matthias Zenger and Martin Odersky collected a set of (partial) solutions for the problem in [3]. We will go through them and see their benefits and drawbacks. Structural and functional decomposition are the two most important ones since the rest of the approaches are extensions or improvements of them.

2.1. **Structural decomposition.** Structural decomposition uses the Interpreter design pattern [15]. It requires a base class from which every data type is inherited, and the base class has a pure virtual function for every algorithm. Every data type implements it's own version of the algorithm.

Extension with a new data type is easy, a new class implementing the data type need to be created. One of the disadvantages is that every algorithm has to be implemented for it, but the main problem with this solution is that every class has to be changed during extension with a new algorithm: a new virtual function has to be created in the base class, and it has to be implemented in every class inherited from the base class.

2.2. **Functional decomposition.** Functional decomposition uses the Visitor desing pattern [15]. There are no restrictions for data types. Each algorithm is implemented by a class with multiple member functions – one for every data type. The objects of these classes are called visitors, and the classes are inherited from a base class which has a pure virtual function for every data type. These pure virtual functions are overridden in the visitor classes to implement the algorithm for the data types. To run an algorithm for a data object a visitor object needs to be created and it's member function for the data object needs to be called.

Extension with a new algorithm is easy, a new class has to be created for the new algorithm. It has the same problem as structural decomposition: every algorithm has to be implemented for every data type. The main problem with this solution is that extension with a new data type is difficult: every visitor has to be extended with a new member function.

(1) Extensible visitors Krishnamurti, Felleisen and Friedman [2] extended the Visitor pattern [15] to be more flexible. They refined the way to introduce new data types: visitors don't need to be changed, the set of member functions can be extended by subclassing. The main problem with this solution is that it is still not type safe – it requires casting. Zenger and Odersky advanced the approach [17] by adding default cases to types and visitors to handle future extensions, but was still not satisfactory because of allowing application of visitors to data variants they were not designed for.
(2) External extension of classes Some programming languages [16] support external extensions of classes making possible extension of a class without changing it's code. Defining functions externally requires default implementations making separate compilation impossible.
(3) Reflection based approach Palsberg and Jay advanced the Visitor pattern [15] to Walkabouts [11] which use reflection to iterate over attributes of unsupported objects, but their solution has no static type safety because of using reflection.
(4) Self types Kim B. Bruce presented a way [10] of using a type construct called `ThisType` to advance the Interpreter design pattern [15]. `ThisType` changes in subclasses and the signature of inherited methods using `ThisType` ensure better static type safety, but the dynamic type of an object has to be available at compile time when calling a method expecting an object with `ThisType` as it's static type.
(5) Generics based approaches Mads Torgersen used generic classes [12]. He presented an approach based on structural decomposition (which he called data-centered) and one on functional decomposition (he called operation-centered). They were both difficult to use for programmers and did not support independent extensibility.

3. Our solution

We approached the problem with generic programming but in a different way Mads Torgersen did [12]: we rely on the term concept defined in [4]. A concept is a set of requirements for a class, and a class models the concept if it meets the requirements. These requirements can be syntactic, semantic, etc. Syntactic requirements will be supported in the upcoming standard of C++ called C++0x [19] [20] [21]. We assume the existence of a concept every data type (including recursive ones) models. A data type can be any class model the concept and an operation can be any function relying only on the concept.

An example for this in the C++ Standard Template Library [9]: a class models the forward iterator concept if it can read through a sequence of objects in one direction. A container is forward iterable if it provides a subclass modelling the forward iterator concept and reading through the elements of the container. Algorithms use these iterators to access the containers (which they know nothing more about), they rely only on the concept.

These systems can be extended in non-intrusive way: new data types can be introduced by creating a new class modelling the concept, new operations can be introduced by writing new generic functions relying only on the concept. This solution is also efficient, since in most cases the compiler knows every type at compile time and can heavily optimise the program.

3.1. **Recursive data types.** We examine creation of recursive data types for this design. Recursive data types contain one or more data objects (modelling the concept) and are data objects themselves, so recursive data types model the concept as well. When the type of the child objects are known at compile time the interface of the children can be used directly: all types are known at compile time. When the types of the child objects are unknown at compile time the only thing the recursive object can assume is that they model the concept. Not only their dynamic but also their static type is unknown at compile time (there is no common base class for data types).

Mat Marcus, Jaakko Järvi and Sean Parent use the Bridge design pattern in [4]. The goal of this pattern is separation of abstraction (in our case the concept) and implementation (classes modelling the concept). They connect static and dynamic polymorphism by creating an abstract base class for every class modelling the concept and a generic wrapper class which is a subclass of the base class and can be instantiated by any class modelling the concept. The abstract base class provides pure virtual functions for every operation required by the concept and wrappers implement these virtual functions by using the wrapped object's interface since every wrapper knows the static type of the wrapped class at compile time.

Inheritance between the base class and wrappers implement dynamic, instantiation of the generic wrapper for each data type implements static polymorphism. Using this idea recursive data types could be implemented when static type of children is unknown at compile time using smart reference objects which could be special objects containing a pointer to wrapper objects and model the concept themselves by calling virtual functions of the wrappers.

Concepts requiring the existence of subtypes modelling another concept (e.g. STL containers need to have an iterator type [9]) make creation of the abstract base class more difficult: since the static type of the wrapped object is not known at compile time, neither does the compiler know the static type of the subtype. The open question is what type should the common base class provide as the subtype. Our answer to this question is repetition of the idea of Marcus, Järvi and Parent [4] for the subtype: the base class could provide the smart reference class to the real subtype as the subtype. For example the base class for STL containers could provide the smart reference class for iterators as it's iterator type and instances of STL algorithms not knowing the static type of the container at compile time could access the iterators through smart reference objects. For example a container could be created accepting any random access STL container [9] using this solution. First iterators of these containers need to be wrapped, so a base class is required for random access iterators. The codes here are not complete classes, just examples demonstrating the logic of the solution, and to keep examples simple we assume that the container's elements are `ints`.

```
class RandomAccessIteratorBase {
public:
  virtual int operator*() const = 0;
  virtual int& operator*() = 0;
};
```

The wrapper template needs to be implemented for iterators:

```
template <typename T>
class RandomAccessIteratorWrapper :
  public RandomAccessIteratorBase {
public:
  RandomAccessIteratorWrapper(const T& t) : _wrappedObject(t) {}
  virtual int operator*() const { return *_wrappedObject; }
  virtual int& operator*() { return *_wrappedObject; }
private:
  T _wrappedObject;
};
```

Finally a smart reference class needs to be created simulating a random access iterator and calling a wrapper in the background. (We use `shared_ptr` from Boost [18] as an underlying smart pointer implementation). We focus on the core idea here and skip other parts (e.g. copy constructor for `RandomAccessIterator`) which a real implementation has to deal with.

```
class RandomAccessIterator {
public:
  template <typename T> RandomAccessIterator(const T& t) :
    _wrapped(new RandomAccessIteratorWrapper<T>(t)) {}
  int operator*() { return _wrapped->operator*(); }
  int& operator*() const { return _wrapped->operator*(); }
private:
  boost::shared_ptr<RandomAccessIteratorBase> _wrapped;
};
```

Now since iterators have been wrapped the wrapper for containers can be created using the iterator wrapper:

```
class RandomAccessContainerBase {
public:
  virtual RandomAccessIterator begin() = 0;
  virtual RandomAccessIterator end() = 0;
};
template <typename T> class RandomAccessContainerWrapper :
  public RandomAccessContainerBase {
public:
  RandomAccessContainerWrapper(const T& t) : _wrapped(t) {}
  virtual RandomAccessIterator begin()
  { return _wrapped.begin(); }
  virtual RandomAccessIterator end() { return _wrapped.end(); }
private:
  T _wrapped;
};
class RandomAccessContainer {
public:
  template <typename T> RandomAccessContainer(const T& t) :
    _wrapped(new RandomAccessContainerWrapper<T>(t)) {}
  RandomAccessIterator begin() { return _wrapped->begin(); }
  RandomAccessIterator end() { return _wrapped->end(); }
private:
  boost::shared_ptr<RandomAccessContainerBase> _wrapped;
};
```

Every STL algorithm [9] for random access containers could work with these wrappers and accept any random access container – without recompiling the algorithm itself. It has a runtime cost but it still acceptable (we have implemented the motivating example using this and measured the runtime cost – see table 1 and table 2).

3.2. **Evaluation of our solution against the requirements.** We have a set of requirements (Zenger's and Odersky's list with an extension in section 3.3): **(1) Extensibility with algorithms and data types.** Algorithms can be added by implementing new generic functions, data types can be added by creating new classes modelling the concept. **(2) Independent extension.** Extension with a new generic function or a new class has no effect on data types or other functions. **(3) Static type safety.** Validity of calling a generic function on a data type is checked when the code calling the function is compiled. Data types have to model the concept and algorithms have to rely only on the concept. In case algorithms rely on a refinement of the original concept data types they are called with have to model that as well. When using unrestricted containers the type of the objects is checked when the objects are placed in the container, therefore algorithms operating on the elements of the container can assume that every element models the concept. **(4) Efficiency.** When runtime type of data objects is known at compile time the compiler can optimise the code. **(5) Separate compilation.** When using unrestricted containers (or references to objects modelling the concept) algorithms can be compiled separately from data types they use. **(6) Reduction of the number of implementations** Generic algorithms do this.

3.3. **Using this idea for the motivating example.** We are going to use this solution for the motivating example and check how effectively can this approach solve the problem compared to the commonly used design patterns. [3] [10] [12] In the motivating example a set of shapes and a set of transformations operating on the shapes are given. Groups of shapes can be created which groups are shapes themselves, operations need to support them either. Shapes are the data types of the expression problem, groups of shapes make them recursive. Transformations are operations operating on the data types.

Since our solution requires a generic concept for data types, the first step of applying the approach to a practical problem is finding one. This concept needs to be generic enough to avoid restriction of the data types since changes in the concept are likely to indicate changes in every data type and operation. In our example data types are shapes, a concept has to be generic enough to describe all kinds of shapes. There are multiple approaches to find one, we use one we found generic enough here for demonstration.

First we define a concept for vectors: we expect a vector type to have a scalar type associated with it, the scalar values form a field and the vectors form a vector space over this field. For example the vector space of two (or three) dimensional vectors over the field of real numbers satisfy these requirements. A generic concept for shapes can be defined based on the generic concept for vectors. Commonly used shapes can be described by a set of vectors. Here are the shapes of the well-known vector graphics standard the Scalable Vector Graphics (SVG) format [27]: **(1) line** can be represented by it's two endpoints. **(2) triangle** can be represented by it's three vertices. **(3) Rectangle**s can be represented by two or three vectors (depending on if their edges are always parallel to the axis of the coordinate system or they can be rotated by any angle). **(4) Ellipse**s can be represented by their bounding rectangle, which indicates that they can be represented by two or three vectors. **(5) Circle**s can be represented by the origin and one point on the edge. **(6)** Paths and shapes described by them (**polyline**s, **polygon**s, **bezier curve**s, **spline**s) can be represented by their control points. As we can see there are shapes which support extension and reduction of the set of their points (polylines, curves, etc.) and there are shapes where the number of points are fixed (rectangles, lines, etc.).

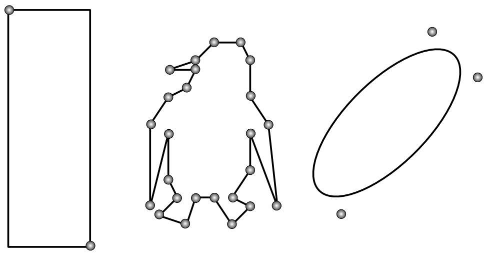


FIGURE 1. Example shapes and vectors describing them

A generic concept for shapes could be defined based on this idea: a shape is represented by a set of vectors completely describing it's shape, location, orientation and size. Each type of shape has a forward iterator type and has `begin()` and `end()` methods similarly to STL containers which can be used to iterate over the vectors representing the shape. Transformations could be implemented similarly to algorithms of the Standard Template Library: they are generic functions using iterators to access the shapes. Here is an example implementation of translation:

```
template <typename Shape>
void translate(Shape& shape, typename Shape::Vector d) {
  for (typename Shape::iterator i = shape.begin();
    i != shape.end(); ++i)
       *i += d;
}
```

Basic shapes like lines, curves, etc. could be implemented by containers of vectors. For example a rectangle or a line could be implemented by an array of vectors to provide O(1) random access to the vectors, a polyline could be implemented by a list of vectors to provide O(1) vector insertion and deletion. Groups of shapes could be implemented by containers of shapes, but they have to be shapes themselves. The union of sets representing the contained shapes could be the set of shapes representing the group itself as a shape since this is the set of shapes which satisfies the expectations of the abstract concept for shapes (completely describes the whole group). This indicates the creation of a special iterator iterating over the elements of the contained shapes. Unrestricted containers could be implemented by creating generic wrappers for shapes. Since the type of iterators is not fixed, generic wrappers have to be created for iterators either.

3.4. **Runtime performance.** We implemented the motivating example to measure the runtime performance of the solution. The test environment was a Linux box with the GNU C++ compiler version 4.1.2, and the code was compiled with level 6 optimalisation. We measured the speed of the translation of two dimensional points and polylines in homogeneous and in unrestricted containers. Using homogeneous containers dynamic type of the shapes are known at compile time making optimalisation possible but restricting flexibility while unrestricted containers accept any type of shapes but have runtime costs because of using dynamic polymorphism.

Table 1. Measurements with point shapes

		Unrestricted container (s)	Homogeneous container (s)
1 000 1 000	shapes times	0.373	0.056
10 000 1 000	shapes times	3.497	0.365
1 000 10 000	shapes times	3.546	0.950

Table 2. Measurements with polylines

		Unrestricted container (s)	Homogeneous container (s)
100 100 100	shapes times control points	0.092	0.059
1 000 100 100	shapes times control points	0.675	0.367
100 1 000 100	shapes times control points	0.940	0.903
100 100 1 000	shapes times control points	0.901	0.884

The results are what we expected – runtime polymorphism has a strong impact on runtime speed (unrestricted containers were 3 - 6 times slower than homogeneous ones).

4. Summary

A large class of software is working on recursive data types. Web browsers, office software, graphic editors, etc. have different components containing other components, and perform operations on them. These software need to be designed carefully, since by applying commonly used design patterns, the possibility of independent development and extension of these components and operations could be lost. In this paper we analysed common patterns and found that they supported independent extension of one of data types or operations, but not both of them. We analysed other existing approaches as well to see their benefits and drawbacks. We proposed a new approach using generic programming in C++ and a solution when a concept is available for data types. In our approach data types are required to model the concept and algorithms required to rely only on the concept when accessing data objects.

A drawback of generic programming in C++ is the lack of support for runtime polymorphism which is required to create unrestricted containers for data objects supporting any data type. We used the technique described by Mat Marcus, Jaakko Järvi and Sean Parent in [4] to connect compile time and runtime polymorphism. We extended the idea with support to unrestricted containers. After measuring the runtime cost of unrestricted containers we found that although they were 3-6 times slower than homogeneous ones, but they are more advantegous in means of flexibility, type safety, and quality of source code.

References

[1] Thomas Becker, *Type Erasure in C++: The Glue between Object-Oriented and Generic Programming*, ECOOP MPOOL workshop, pp.4-8, Berlin, 2007.

[2] Shriram Krishnamurthi, Matthias Felleisen, Daniel P. Friedman, *Synthesizing Object-Oriented and Functional Design to Promote Re-Use*, LNCS Vol.1445, p..91-111, 1998.

[3] Matthias Zenger, Martin Odersky, *Independently Extensible Solutions to the Expression Problem*, Technical Report IC/2004/33 EPFL, Lausanne, 2004.

[4] Mat Marcus, Jaakko Järvi, Sean Parent, *Runtime Polymorphic Generic Programming - Mixing Objects and Concepts in ConceptC++*, ECOOP MPOOL workshop, Berlin, 2007.

[5] Ronald Gracia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock, *A Comparative Study of Language Support for Generic Programming*, ACM SIGPLAN Notices, Vol.38, Issue 11, OOPSLA conference paper, pp.115-134, Anaheim, 2003.

[6] Scott Meyers, *Effective C++*, Addison-Wesley, 2005, [220], ISBN: 0321334876

[7] Scott Meyers, *More Effective C++*, Addison-Wesley, 1996, [336], ISBN: 020163371X

[8] Scott Meyers, *Effective STL*, Addison-Wesley, 2001, [288], ISBN: 0201749629

[9] Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley, 1997, [1040], ISBN: 0201327554

[10] Kim B. Bruce, *Some challenging typing issues in object-oriented languages.* In Proceedings of Workshop on Object-Oriented Development (WOOD'03), volume 82 of Electronic Notes in Theoretical Computer Science, 2003.

[11] Jens Palsberg, C. Barry Jay, *The Essence of the Visitor Pattern*, Proceedings of the 22nd International Computer Software and Applications Conference, p.9-15, August 19-21, 1998

[12] Mads Torgersen, *"The Expression Problem Revisited. Four New Solutions Using Generics."* In: M. Odersky (ed.): ECOOP 2004 - Object-Oriented Programming (18th European Conference; Oslo, Norway, June 2004; Proceedings). Lecture Notes in Computer Science 3086, Springer-Verlag, Berlin, 2004, 123-143.

[13] Philip Wadler, *The expression problem*, Message to Java-genericity electronic mail list, November 12, 1998.

[14] Philip Wadler, *The expression problem: A retraction*, Message to Java-genericity electronic mail list, February 11, 1999.

[15] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Abstraction and reuse of object-oriented designs*, Addison-Wesley, 1994, [416], ISBN: 0201633612

[16] C. Clifton, G. T. Leavens, C. Chambers, T. Millstein, *MultiJava: Modular open classes and symmetric multiple dispatch for Java*, Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications, pp.130-145, ACM Press, 2000.

[17] Matthias Zenger, Martin Odersky, *Extensible algebraic datatypes with defaults*, Proceedings of the International Conference on Functional Programming, Firenze, 2001.

[18] B. Karlsson, *Beyond the C++ Standard Library, An Introduction to Boost*, Addison-Wesley, 2005.

[19] Bjarne Stroustrup, *The Design of C++0x*, C/C++ Users Journal, May, 2005

[20] Douglas Gregor, Bjarne Stroustrup, *Concept Checking*, Technical Report, N2081, ISO/IEC SC22/STC1/WG21, Sept, 2006

[21] G. Dos Reis, B. Stroustrup, *Specifying C++ concepts*, Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL), 2006, pp. 295-308.

[22] ANSI/ISO C++ Committee, *Programming Languages – C++*, ISO/IEC 14882:1998(E), American National Standards Institute, 1998.

[23] K. Czarnecki, U. W. Eisenecker, *Generative Programming: Methods, Tools and Applications*, Addison-Wesley, 2000.

[24] J. Siek, A. Lumsdaine, *Essential Language Support for Generic Programming*, Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation, New York, USA, pp 73-84.

[25] Bjarne Stroustrup, *The Design and Evolution of C++*, Addison-Wesley, 1994

[26] D. Vandevoorde, N. M. Josuttis, *C++ Templates: The Complete Guide*, Addison-Wesley, 2003.

[27] Scalable Vector Graphics (SVG) 1.1 Specification, W3C Recommendation 14 January 2003.
`http://www.w3.org/TR/2003/REC-SVG11-20030114`

Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary

E-mail address: `abel@sinkovics.hu, gsd@elte.hu`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

THE RECONSTRUCTION OF A CONTRACTED ABSTRACT SYNTAX TREE

RÓBERT KITLEI

ABSTRACT. Syntax trees are commonly used by compilers to represent the structure of the source code of a program, but they are not convenient enough for other tasks. One such task is refactoring, a technique to improve program code by changing its structure.

In this paper, we shortly describe a representation of the abstract syntax tree (AST), which is better suited for the needs of refactoring. This is achieved by contracting nodes and edges in the tree. The representation serves as the basis of the back-end of a prototype Erlang refactoring tool, however, it is adaptable to languages different from Erlang.

In turn, we introduce an algorithm to reconstruct the AST from the representation. This is required in turn to reproduce the source code, the ultimate step of refactoring.

1. INTRODUCTION

The ASTs constructed using context-free grammars is the representation most applications choose to describe the syntactic structure of source code of programming languages. Most applications use standard lexers and parsers that are designed with the goals of compilers in mind. Compilers – and therefore their standard tools – drop inessential infomation such as punctuation and whitespace after using them to determine token boundaries. Such information is important if one has to preserve the source code as a whole. Also, ASTs are not designed to support searching, as this feature is not required in compilers, the most common users of ASTs.

The above representation does not sufficiently support some applications. An alternative representation is proposed by the Erlang refactoring group at

Received by the editors: September 16, 2008.

2000 *Mathematics Subject Classification.* 68N20,68Q42.

1998 *CR Categories and Descriptors.* D.2.10. [**Software**]: Software engineering – *Representation*.

Key words and phrases. Abstract syntax tree, Syntactic reconstruction.

Supported by GVOP-3.2.2-2004-07-0005/3.0 ELTE IKKK and Ericsson Hungary.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

Eötvös Loránd University (Budapest, Hungary), although this representation has proved useful for purposes other than refactoring as well. The group proposed this representation after previous experience with refactoring [5, 7]. Details about the representation and the refactoring tool can be found in [4].

Although the new representation is more convenient for many purposes, e.g. refactoring, there was a trade-off between usability and functionality, described in detail in section 3.1. Namely, for standard compiler tools, pretty-printing the source from the constructed AST is straightforward using a depth-first algorithm. However, since the new representation does not have all child edges of a node in order, a more elaborate algorithm was needed, which is described in section 3.

The structure of the paper is as follows. In section 2, the representation of the graph is described to such depth as is necessary for understanding the rest of the paper. Section 3.1 poses the central problem of the paper. The rest of section 3 proposes an algorithm that solves this problem. Sections 3.2 and 3.4 in this section describe the contribution of the paper. Finally, section 4 lists related work.

2. Representation structure

2.1. **Node and edge contractions.** ASTs built on top of source codes are typically created by compilers in compilation time. Such syntax trees are discarded after they have been used, and their construction does not involve complex traversals: they follow the construction of the tree. There are, however, applications in which the role of ASTs are augmented. In refactoring, for example, tree traversals are extensively used, because a lot of information is required that can be acquired from different locations.

In order to facilitate these traversals, a new representation of the AST was introduced, which is described in detail in [4]. Here we give an overview of the relevant parts of the representation.

ASTs inherently involve parts that are unnecessary for information collection, or are structured so that they make it more tedious. One obvious case is that of chain rules: the information contained in them could be expressed as a single node, yet the traversing code has to be different for each node that occurs on the way.

Another case can be described by their functionality: the edges of the nodes can be grouped so that one traversal should follow exactly those that are in one group. To give a concrete example, clauses in Erlang have parameters, guard expressions and a body, and there are associated tokens: parentheses and an arrow. Yet the actual appearance of the clauses can be vastly different, see Figures 1 and 2. When collecting information, often either all parameters

```
if
    X == 1 -> Y = 2;
    true   -> Y = 3
end
```

Figure 1. If clauses.

```
to_list(Text) when is_atom(Text)    -> atom_to_list(Text);
to_list(Text) when is_integer(Text) -> integer_to_list(Text);
to_list(Text) when is_float(Text)   -> float_to_list(Text);
to_list(Text) when is_list(Text)    -> Text.
```

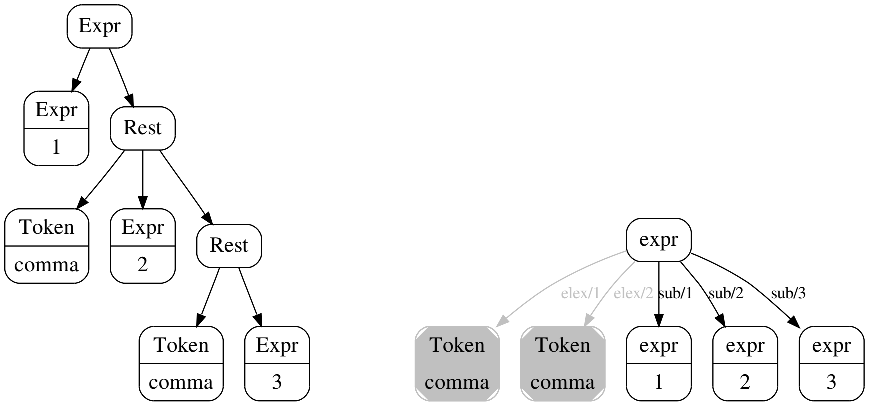
Figure 2. Function clauses with guards.

or all guard expressions are required at a time during a traversal pass, but seldom both at the same time of the traversal. Therefore, it is natural to partition the edges into groups along their uses. Since the partitions depend on the traversals used, the programmer has to decide by hand how groups should be made. This way, only as few groups have to be introduced as needed in a given application.

Another way to make the representation more compact is to contract repetitions. Repetitions are common constructs in programming languages: they are repeated uses of a rule with intercalated tokens as separators. Instead of having a slanted tree as constructed by an AST, it is more convenient for traversal purposes to represent them by a parent node with all of the repeated nodes and the intermediate tokens as its children. As a matter of fact, in the example in the above paragraph the parameters and guard expressions are already a result of such a contraction. These contractions are similar to the list formation annotations in Overbey and Johnson [2].

Performing the above contractions has two main advantages. One is that much fewer cases have to be considered. In the case of Erlang, the grammar contained 69 nonterminals, which was reduced to three contracted node groups: forms, clauses and expressions.

Since the contraction groups are different for each language (and may even differ in each application, depending on the needed level of detail), it is important that the approach should be adaptable to a wide range of grammars. This is one of the reasons why an XML representation was chosen. The grammar rules, the contraction groups and the edge labels are described in this file. The scanner and parser are automatically generated from this file. The contracted



(a) AST.

(b) Contracted AST.

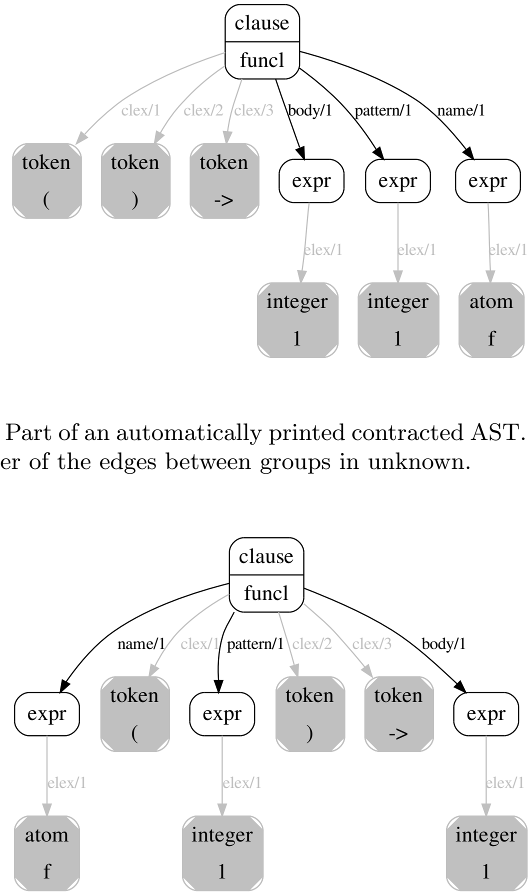
FIGURE 3. Repetition in the expression `1,2,3`.

structure is automatically constructed during parse time (not converted from an AST).

2.2. **Representation of the contracted AST.** The inner nodes of the contracted AST are the contracted nodes, which also contain the originating nonterminal as information. The leaf nodes of the contracted AST are the tokens, which contain the token text and the whitespace before and after the token. The nodes are connected by labelled edges; the labels determine the contraction classes they can connect.

Contractions do not fully preserve edge ordering: order is preserved only between the edges with the same label, not between different labels. This is why the original AST cannot be restored easily: in Figure 4b, it is not possible to determine whether the tokens of the clause come before, after or in between the expressions. To make it possible, more information about the structure of the contracted nodes is needed.

The lack of order between label groups is the result of using a database for storage, which is required for fast queries. However, it is expected to be a good trade-off, since the exact AST order of the nodes is seldom needed (most importantly, when reprinting the contents of the graph into a file), while it provides queries in linear time of their length. The order of the links with the same label, which is important during queries, is retained.



(a) Part of an automatically printed contracted AST. The order of the edges between groups in unknown.

(b) The nodes rearranged in the right order. The order within the groups is retained. The tokens read: `f(1) -> 1`.

Figure 4. A contracted AST node with a `body`, a `pattern`, a `name` and three `clex` edges.

3. Reconstruction of the AST

3.1. Problem when reproducing the original token order. In the previous versions of RefactorErl, the token nodes in a file were linked by edges with the special label `next`, with the first token linked from the file by `first_token`.

This solved the problem of getting the original tokens: they could be acquired by getting the first token, then iterating on the `next` edges until there were none left. Another related question, determining the token at a given position in the file, was also solved easily by iteration on the `next` edges, and calculating the remaining positions. However, these edges have proved to be too difficult to handle when manipulating the syntax tree: the `next` edges would have to be synchronised each time parts of the syntax tree were inserted, removed or moved. Also, when manipulating repeat constructs such as lists, some tokens (in the case of lists, the separating commas) would have to be dealt with.

The approach taken in this paper is different. Instead of repairing the `next` edge links, they are omitted altogether. This immediately solves the problem that occurs when manipulating the syntax tree, because the adjacent tokens are not linked anymore. At the same time, the two other questions are reopened: how to get the token by position and how to print the file. In the rest of the chapter, a method is presented to reproduce the AST. This also yields the original tokens as the front of the tree. Using the original tokens, both questions are trivially answered.

3.2. **Grammar rule constructs.** The chosen grammar description is close to a BNF description. The grammar rules are grouped by what contraction group their head belongs to. Rules, of course, may have more alternatives. The right hand sides of rules consist of a sequence of the following:

- **tokens**, that contain the token node label,
- **symbols**, that contain the child symbol's nonterminal and the edge label,
- **optional constructs**, sequences that either appear or not in a concrete instance and
- **repeat constructs** that contain a symbol and a token; its instances are several (at least one) symbols with tokens intercalated.

Since optionals and repeats may contain one another, we shall refer to the number of contained nestings as the depth of the construct.

As an example that contains both constructs described above, let us examine the structure of lists. The structure of lists is described as follows. Lists start with an opening bracket token and end with a closing bracket token. Between them is an optional construct. The optional part consists of a repeat construct. The repeat construct uses comma tokens to separate symbols that are linked using "sub" edges from the parent node. The portion of the actual Erlang code that shows the above structure is shown in figure 5 in order to have a more concise overview.

Lists can be empty lists, or lists containing expression symbols separated by comment tokens. In the first case, the optional part is not present. In

```
[{token,"op_bracket"},
 {optional,[{repeat,"comma","sub"}]},
 {token,"cl_bracket"}];
```

FIGURE 5. The structure of lists as an Erlang structure used in the actual implementation. Slightly abridged.

the second case, the optional is present. If there is one element in the repeat construct, there is exactly one symbol element present which denotes the expression.

$$\underbrace{[}_{\text{token}}\quad\underbrace{\qquad}_{\text{empty optional}}\quad\underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\quad\underbrace{1}_{\text{repeat in optional}}\quad\underbrace{]}_{\text{token}}$$

$$\underbrace{[}_{\text{token}}\underbrace{1,2,3,4,5,6,7,8+9,f()}_{\text{repeat in optional}}\underbrace{]}_{\text{token}}$$

The grammar description contains the following restrictions. First, no optionals may start with another optional. Second, two repeats in the same rule may not contain the same symbols, nor tokens. Third, no constructs (optionals and repeats) may have a depth of more than two.

The main reason for these restrictions is that they help prevent ambiguities, as seen in the `absence` and `multichoice` constructs in the description or reconstruction.

The third restriction is not necessary for theoretical, but for practical purposes: it is there to keep the processing algorithm described later at a manageable size and complexity while not deducing the expressive power of the constructs too much. Indeed, for a construct at any depth, a new nonterminal can be introduced to take its place, thereby reducing the depth of the parent construct. This way, the depth of the constructs could be limited to one; practice has shown that two is a reasonable limit.

The grammar is expressive enough, as even without the constructs it has Chomsky class L_2.

The first restriction can be enforced by the DTD of the XML. The third restriction could also be enforced if the inner optionals and repeats would have different names, at the expense of comfort.

3.3. **Derived constructs used in reconstruction.** The rule descriptions above are sufficient in most cases to reconstruct the original node order of a node in the contracted AST by looking at only the nonterminal of the node, the node's child links and the rule description. Yet there are two types of rules where these data are not enough. In these cases, another kinds of constructs have to be prepared before reconstruction. These structural constructs are automatically derived from the syntax description like the scanner and the parser. Both of these constructs require information about the children nodes, and conglomerate several grammar rules.

The first skeleton construct is called **absence multi-rule construct** because it selects the appropriate grammar rule based on the absence or presence of a token or a symbol. The following example shows a fun-expression that can either have explicit clauses (in the first case) or can be an implicit fun expression, just showing the function name and arity (the second case). Here, the only way to decide which rule to use is to check for the `end` token: if it is present, it is the first rule, if it is not, the second.

$$\underbrace{fun}_{\text{token}}\underbrace{(1) \to ok;\ (2) \to error}_{\text{repeat}}\underbrace{end}_{\text{token}}$$

$$\underbrace{fun}_{\text{token}}\underbrace{\text{another\_module/2}}_{\text{repeat}}$$

Named functions have the name of the function as a subexpression in the beginning of each clause. The clauses of unnamed functions start immediately with the parameter list in parentheses. The only way to decide between them is to search for the symbol at the beginning. (Note that symbols also contain the link label. Its omission, similar to calling the parameter list a "repeat in optional," is a simplification.)

$$\underbrace{search}_{\text{symbol}}\underbrace{\ (\ }_{\text{token}}\underbrace{Structure, Pattern}_{\text{repeat in optional}}\underbrace{\ )\ }_{\text{token}}\underbrace{\to}_{\text{token}}\underbrace{\ldots}_{\text{repeat}}$$

$$\underbrace{\ (\ }_{\text{token}}\underbrace{Structure, Pattern}_{\text{repeat in optional}}\underbrace{\ )\ }_{\text{token}}\underbrace{\to}_{\text{token}}\underbrace{\ldots}_{\text{repeat}}$$

The second skeleton construct the **multichoice multi-rule construct**. In it, there is a list of possible present symbols or tokens. The actual rule can be decided depending on which of the symbols (or tokens) occur. The symbols (or tokens) listed are mutually exclusive: one and only one occurs, provided that the source is valid.

Both if and case clauses are branch clauses and they may look identical. Similarity occurs when the case clause has no guard and the guard of the if expression is a single variable. They can be separated only if they make different links to their first symbol as “guard” and “pattern” respectively.

Infix expressions provide an example for a token-based multichoice construct. Logical operators `andalso` and `orelse` (and several other operators) can function on the same pair of arguments. Here, checking all the possible token types, exactly one will be present, and this of course determines the operation as well.

3.4. **AST reconstruction.** From the XML syntax description, a node structure skeleton is automatically generated. It assigns to each contracted node type either a one-rule structure, or an absence or multichoice multi-rule construct.

The syntax tree can be reconstructed using a recursive algorithm. Starting from the node in the tree that corresponds to the file, we do the following.

(1) We determine the structure of the actual rule which is used. If a one-rule structure is assigned to the parent node, it is the structure; if a multi-rule construct describes it, we have to check the children of the node as well.
(2) The sequence in the structure is processed.
    (a) For any token or symbol, take the next fitting one.
    (b) For repeats, take all symbols (altogether n) with the appropriate edge label, and take $n - 1$ fitting tokens.
    (c) For optionals beginning with a token or symbol, use the optional sequence if a fitting child is present.

Tokens’ edge labels are determined by the type of the parent node. We call a token node fitting the token in the description if it is linked to the parent node by such a link. A symbol is called fitting in a similar way, except that for symbols, the description explicitly contains their expected links.

Using the above algorithm, the original AST can be recovered. Strictly speaking, this is not the AST, as chain rules are still not expanded; this does not add significant information, and can easily be done, should the need arise.

The front of the AST contains the token nodes in their original order. Since all whitespace information is contained in the tokens, and punctuation tokens are not omitted, the whole original file can be reprinted. Determining the token at a given position of the file can be done by doing a linear search on the original tokens in order.

With an additional layer between the lexer and the parser, it is possible to handle preprocessor constructs such as include files and macros (even ones that cross-cut the syntax). Additional information relating to such preprocessor constructs can be stored in the graph as well. During reconstruction, finding a node that originates from such a construct does not pose a challenge, as these constructs mostly involve directly storing all of their relevant tokens.

```
{absence, "end", token,
    [{token, "fun"}, {repeat, ";", "exprcl"}, {token, "end"}],
    [{token, "fun"}, {symbol, "sub"}]
}
```

Figure 6. The skeleton description of a fun expression.

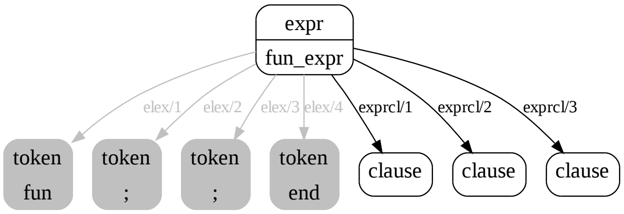


Figure 7. Graph representation of a function expression with three clauses. One instance of such a function is the following.

```
fun
(X) when X > 0 -> ok, X;
(X) when X < 0 -> ok, -X;
(0) -> error
end
```

3.5. **Example.** Figure 7 shows a fun expression with three subclauses in the graph representation. The nodes representing the clauses are not in order, as ordering exists only within the `elex` and the `exprcl` edge classes. Let us use the algorithm described in Section 3.4 in order to recover the order of the nodes.

The description of the fun expression in 6 contains a skeleton construct. In order to eliminate it, we have to check whether the actual structure contains an `end` token. It does, therefore the first of the two descriptions is chosen. This description starts with a `token`, therefore the first element in the order is the first token that is connected to the `expr` parent node. The label of the connecting edge is determined by class of the parent node, `expr`: `elex`. Thus, the first child node in order is the one connected by `elex/1`. Next in the description is a repeat construct with the `exprcl` symbol link and the semicolon. For this, we take all three nodes that are linked by `exprcl`, and one less token (linked, as before, by `elex/1`). The restored order is the symbols with the tokens intercalated between them. The last element of the description is another token, for which we take the last remaining token. Since all the description and the actual nodes are consumed, the representation is syntactically valid. The restored order can be seen in Figure 8; restoration is continued for all child nodes.

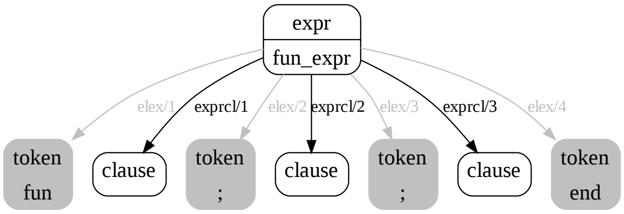


FIGURE 8. Graph representation of a function expression with three clauses.

4. Related work

The design of the representation was shaped through years of experimentation and experience with refactoring functional programs. The first refactoring tools produced at Eötvös Loránd University [5, 7] used standard ASTs for representing the syntax. It became evident that such a representation is not convenient enough for refactoring purposes, and a new design was needed. The resulting design [4] used the contracted graph described in section 2 as representation of the syntax tree, but it relied on superfluous `next` edges to maintain the order of tokens. Section 3.1 argues why having these was undesirable, and the whole of section 3 describes the new structures and algorithms that were necessary to avoid them.

The Java language tools srcML [8], JavaML [3] and JaML [1] use XML to model Java source code. Since XML naturally outlines a tree structure, these representations conserve node order, which enables them to easily reprint the source.

Since the representation outlined in this paper differs so much from the usual approach taken – using a contracted representation instead of the more conventional ASTs – the problem of reproducing the original nodes in order does not appear in other works, as this task is trivial when using an AST.

References

[1] G. Fischer, J. Lusiardi, J. Wolff v. Gudenberg, *Abstract syntax trees and their role in model driven software development,* in Proceedings of the International Conference on Software Engineering Advances, IEEE Computer Society (2007), page 38.

[2] J. Overbey, R. Johnson, *Generating Rewritable Abstract Syntax Trees,* in Proceedings of the 1st International Conference on Software Language Engineering (SLE 2008), Toulouse, France, 2008.

[3] G. J. Badros, *Javaml: a markup language for Java source code,* in Proceedings of the 9th international World Wide Web conference on Computer networks: the international journal of computer and telecommunications networking, North-Holland Publishing Co. Amsterdam, The Netherlands, 2000, pp. 159–177.

[4] R. Kitlei, L. Lövei, T. Nagy, Z. Horváth, T. Kozsik, *Preprocessor and whitespace-aware toolset for Erlang source code manipulation,* in Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, Hatfield, UK, 2008.

[5] R. Szabó-Nacsa, P. Diviánszky, Z. Horváth, *Prototype environment for refactoring Clean programs,* in Proceedings of the 4th Conference of PhD Students in Computer Science (CSCS 2004), Szeged, Hungary, 2004.

[6] H. Li, S. Thompson, L. Lövei, Z. Horváth, T. Kozsik, A. Víg, T. Nagy, *Refactoring Erlang Programs,* in Proceedings of the 12th International Erlang/OTP User Conference, 2006.

[7] Lövei, L., Horváth, Z., Kozsik, T., Király, R., Víg, A. Nagy T, *Refactoring in Erlang, a Dynamic Functional Language,* in Proceedings of the 1st Workshop on Refactoring Tools, Berlin, Germany, 2007, pp. 45-46.

[8] J. I. Maletic, M. L. Collard, A. Marcus, *Source code files as structured documents,* in Proceedings of 10th IEEE International Workshop on Program Comprehension (IWPC'02), IEEE Computer Society, Washington, DC, USA, 2002, pp. 289–292.

Faculty of Informatics, Eötvös Loránd University, Pázmány Péter sétány 1/c, H–1117 Budapest, Hungary

E-mail address: kitlei@elte.hu

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

ADVANTAGES AND DISADVANTAGES OF THE METHODS OF DESCRIBING CONCURRENT SYSTEMS

ANITA VERBOVÁ AND RÓBERT HUŽVÁR

Abstract. This paper provides a review of existing paradigms for modelling concurrent processes. First we describe in short some formal methods designed for the development of the theory of concurrency.

Because no unified theory or calculus for concurrency has showed up, we concentrate on interaction categories and their features relevant for our purposes. They are able to describe some essential features of communicating processes.

Finally we confront all these methods and point out their limitations and expressive power. We highlight some open problems with regard to reasoning about concurrent systems.

1. Description of concurrent systems by process calculi

There exists many methods for the formal description of concurrent systems. The most substantial of these paradigms is the process calculus [3]. Its pioneers were Milner a Hoare with their methods CCS [7] and CSP [5] respectively. There are also another paradigms, which describe concurrent processes and some of their properties. Here belongs for instance the π-calculus [10], the structure of events [17], Petri nets [4] and SCCS [8]. SCCS (synchronous calculus of communicating systems) is a process algebra in which processes contribute their visible activity synchronously, or in other words, in unison with a global clock. The algebra also contains operators for structuring process

Received by the editors: September 14, 2008.

2000 *Mathematics Subject Classification.* 18C10.

1998 *CR Categories and Descriptors.* F.3.2 [**Logics and Meanings of Programs**]: Semantics of Programming Languages – *Process models*; F.4.1 [**Logics and Meanings of Programs**]: Mathematical Logic – *Proof theory* .

Key words and phrases. Category theory, Concurrent systems, Process algebra, Interaction categories.

This paper has been presented at the 7th Joint Conference on Mathematics and Computer Science (7th MaCS), Cluj-Napoca, Romania, July 3-6, 2008.

definitions, renaming and inhibiting actions and permitting nondeterministic choices of behaviour [11].

Main result of this paradigm is to develop an algebraic theory of concurrency as a foundation for structural methods for describing concurrent systems.

2. Description of concurrent systems by the π-calculus

In [9] Milner describes the π-calculus as a step towards a canonical calculus for concurrent systems. It is a minimal calculus such that all programs that are computable by a concurrent system can be encoded in it. The π-calculus hopes to play a similar role for concurrent systems to that played by the λ-calculus for sequential computation.

The π-calculus is a process algebra, similar to CCS, but is designed to model systems with dynamically changing structure: that is, the links between the components of a system can vary dynamically during the evolution of the system. This property, which is called *mobility*, can at best be modelled indirectly in established process algebras.

The π-calculus allows channel names to be passed as values in communications. In fact the π-calculus combines the concepts of channel names, value and value variables into a single syntactic class: *names*. The π-calculus is not a higher order calculus: it is only accesses to agents that are being passed in communications, not the agents themselves. The passing of agents as parameters in communications is undesirable since agents would then become replicated, and the replication of agents with state is difficult. Limiting ourselves to the passing of accesses means that we can allow certain agents only limited access to other agents, and have several agents having different access abilities to some common agent.

The main features of the π-calculus are the dynamic creation of channel names and handshake communication on these names.

3. Mathematical theory of computational paradigm

The latest established of current paradigms for the semantics of computation is *denotational semantics*. In spite of its pretensions to universality, denotational semantics has a natural slant to computational paradigm: *functional computation*. By this we mean not only functional programming languages, but the whole range of computation, where the behaviour of the program is abstracted as the computation of a function. This view of programs as functions is built into the fundamental mathematical framework, which was denotational semantics found on: a category of sets for the interpretation of types, and specific functions between these sets for the interpretation of programs.

4. Category theory for modelling concurrent systems

The development of interaction categories [15] results from the limitations of the paradigms mentioned above. These paradigms have developed independently. Their separate development is considered to be the main open problem, i.e. how can we combine the functional and concurrent process paradigms with their associated mathematical support in a single unified theory. This unification is the consequence of the following investigations:

(1) in process algebras:
- There is no typing, hence there is a need of a good *type theory for concurrent processes.*
- Stress is laid mainly on which are these processes, rather than on what structure they have collectively.
- There is a real *confusion* of formalisms, combinators and equivalences.
- Their major objection is that did not appear any *generalized theory* or calculus for concurrency.

(2) in denotational semantics:
- Denotational semantics works well not only for the description but also for the language design and programming methods.

Unification of these two methods is necessary to obtain correct basis for languages connecting concurrent processes and communication with types on one hand, and higher order constructions with polymorphism on the other. It is also desirable for the foundations of suitable type systems for concurrency.

4.1. **Interaction categories.** In the categorical semantic approach, we define a category of processes [2], where we model types as objects, processes as morphisms, and interaction as morphism composition.

Once this structure of typed arrows closed under composition has formulated, then a great amount of further structure is determined up to isomorphism.

4.2. **Categorical structure of synchronous processes.** In [2] interaction categories are introduced by presentation of a canonical example, category of synchronous processes **SProc**. In general *objects* of interaction categories are concurrent system specifications, their *morphisms* are synchronisation trees, *composition* is given by synchronous product and restriction and *identities* are synchronous buffers. The category **SProc** has a very rich structure.

More formally the *objects* of **SProc** are pairs $A = (\Sigma_A, S_A)$, where Σ_A is an alphabet of actions (labels) and $S_A \subseteq^{nepref} \Sigma_A^*$ is a safety specification. Hence, a safety specification is a non-empty and prefix-closed set of traces over A, which represents a linear-time safety property.

A process p of type A, written $p : A$, is a synchronization tree modulo strong bisimulation, with labels from Σ_A, such that $\texttt{traces}(p) \subseteq S_A$. Following Aczel we use a representation of synchronization trees as non-well-founded sets, in which a process p with transitions $p \xrightarrow{a} q$, $p \xrightarrow{b} r$ becomes $\{(a,q)\,(b,r)\}$.

The most convenient way of defining the morphisms of **SProc** is first to define a *-autonomous structure on objects, then say that the morphisms from A to B are processes of the internal hom type $A \multimap B$. Given objects A and B, the object $A \otimes B$ has

$$\begin{aligned} \Sigma_{A\otimes B} &= \Sigma_A \times \Sigma_B \\ S_{A\otimes B} &= \left\{\sigma \in \Sigma_{A\otimes B}^* | \texttt{fst}^*(\sigma) \in S_A \wedge \texttt{snd}^*(\sigma) \in S_B\right\}. \end{aligned}$$

The duality is trivial on objects: $A^\perp = A$. This means that at the level of types, **SProc** makes no distinction between input and output. Because communication is based on synchronization, rather than on value-passing, processes do not distinguish between input and output either.

The definition of $\otimes$ makes clear how are processes in **Sproc** synchronous. An action performed by a process of type $A \otimes B$ consists of a pair of actions, one from the alphabet of A and one from that of B. Thinking of A and B as two ports of the process, synchrony means that at every time step a process must perform an action at every one of its ports.

A *-autonomous category in which $\otimes$ is self-dual, i.e. such that $(A \otimes B)^\perp \cong A^\perp \otimes B^\perp$, is a compact closed category. Hence in a compact closed category $A \parr B \cong A \otimes B$. In the special case when $A^\perp \cong A$ the linear implication, defined by $A \multimap B = A^\perp \parr B$, also corresponds to $A \otimes B$. In **SProc** $A^\perp = A$, and so $A \parr B = A \multimap B = A \otimes B$.

Not all interaction categories are compact closed, but those that are, support more process constructions than those, that are not.

A *morphism* of **SProc** $p : A \to B$ is a process p of type $A \multimap B$. Since $A \multimap B = A \otimes B$, this means for the process p that it is of type $A \otimes B$.

Given $p : A \to B$ and $q : B \to C$ then we can define their *composition* $p;q : A \to C$ in the category **SProc** as follows:

$$\frac{p \xrightarrow{(a,b)} p' \quad q \xrightarrow{(b,c)} q'}{p;q \xrightarrow{(a,c)} p';q'}$$

in which matching of actions takes place in the common type B (as in relational composition), at each time step. This ongoing communication is the *interaction* of interaction categories.

The identity morphisms are synchronous buffers: whatever is received by $\mathtt{id}_A : A \to A$ in the left copy of A is instantaneously transmitted to the right copy (and vice versa – there is no real directionality). If p is a process with sort Σ and $S \subseteq^{nepref} \Sigma^*$ then the process $p \restriction S$ is defined by:

$$\frac{p \xrightarrow{a} q \quad a \in S}{p \restriction S \xrightarrow{a} q \restriction (S/a)}$$

where $S/a \overset{\text{def}}{=} \{\varepsilon\} \cup \{\sigma | a\sigma \in S\}$.

The *identity morphism* $\mathtt{id}_A : A \to A$ is defined by $\mathtt{id}_A \overset{\text{def}}{=} \mathtt{id} \restriction S_{A \multimap A}$ where the process $\mathtt{id}$ with sort Σ_A is defined by:

$$\frac{a \in \Sigma_A}{\mathtt{id} \xrightarrow{(a,a)} \mathtt{id}}$$

Since $\oplus$ is a coproduct, its dual is a product; because all objects of **SProc** are self-dual, this means that $A \oplus B$ is itself also a product of A and B – so it is a biproduct. If $p; q : A \to B$ then their non-deterministic combinator is defined by:

$$\begin{aligned} p + q &= A \xrightarrow{\Delta_A} A \oplus A \xrightarrow{[p,q]} B \\ &= A \xrightarrow{\langle p,q \rangle} B \oplus B \xrightarrow{\nabla_B} B \end{aligned}$$

where $\Delta_A \overset{\text{def}}{=} \langle \mathtt{id}_A, \mathtt{id}_A \rangle$ is the *diagonal* and $\nabla_B \overset{\text{def}}{=} [\mathtt{id}_B, \mathtt{id}_B]$ is the *codiagonal*. To make clear the definition of $+$, consider the composition $\langle p, q \rangle ; \nabla_B$. Pairing creates a union of the behaviours of p and q, but with disjointly labelled copies of B. Composing with ∇_B removes the difference between the two copies. A choice can be made between p and q at the first step, but then the behaviour continues as behaviour of p or behaviour of q. Thus we obtain the natural representation of the *non-deterministic sum* in terms of synchronisation trees in CCS.

4.3. **Categorical structure of asynchronous processes.** Category of processes **Buf** with a similar structure as interaction categories is defined in [14]. Morphisms are given by labelled transition systems representing processes in a language like CCS. These processes are asynchronous in the sense that a sender does not wait for the message delivering as in handshake mechanism of CCS. In the category **Buf**

- Objects are sets (names of channels).
- Morphisms $A \to B$ are labelled transition systems with input actions from A and output actions from B, illustrated according to weak bisimulation.
- Composition of morphisms is interaction in the form of parallel composition and restriction.
- Identities are asynchronous buffers, *i.e.* processes, which simply forward the messages, which they deliver and they do not necessarily preserve order.

We can define products as parallel composition without interaction, and **Buf** is a traced monoidal category [6], thus it provides a feedback operation, and we are able to build cycles of processes.

The category **Buf** is obtained by restricting the sets of morphisms to those processes that are buffered. In [13], axioms are given to classify those processes that behave the same when composed with a buffer, for the case when the buffer does not preserve the order of messages (as in **Buf**), and for first-in-first-out buffers. These axioms are quite strong. They require, for example, that a process can at every state do an input transition on each input channel. For first-in-first-out buffers, they require that from each state there is at most one output transition.

5. Comparison of paradigms for the description of concurrent systems

In this section we summarize the point of view of the designed paradigms.

Still is widely appreciated that the functional computation is only one, relatively restricted part of computational universe, where distributed systems, real-time systems and reactive systems do not really fit. Success of **denotational semantics** out of the area of *functional computation* is very limited.

Partly because of the absence of a good type theory, in **process algebras** has been a considerably systematic *chaos between specifications and processes. Names* in process calculi are used as corresponding names, which distinguish these calculi syntactically and strongly from the others. For example, process algebra tend to be more abstract and specification-oriented than Petri nets, while the latter describe concurrency at a more intricate structural level.

The π**-calculus** is not higher order, unlike the λ-calculus where λ expressions (interpreted as agents) can be passed as arguments to functions and bound to variables. In the π-calculus we cannot pass processes themselves in communications or substitute them for names. We can construct implementations of functional and higher order programming languages on the basis of passing simple data items between registers and carrying out simple operations on them, where these data items function either as pointers to the code of functions or other complex data structures, or as values, instead of passing the functions and complex data structures themselves. Perhaps the most valuable aspect of the π-calculus is that it gives us an abstract, mathematical way to model this kind of computing, and so allows us to reason about such implementations in a formal way.

It is debatable whether the π-calculus can be extended in such a way as to make representations of complex constructions easier. Summation and τ-actions produce semantic difficulties, and so it might be worth investigating some other external choice operator. Even with summation and conditional guards we could not build the infinite functions and operators. The question of how best to extend the calculus in order to make it more useful therefore remains open. Similar open problem is the extension of π-calculus to include some notions of type.

Method of **formal calculus** [1] stems from the set of combinators forming a syntax. The weakness of these methods is already in the use of this set of *combinators* rather then another.

In the category **Sproc** a synchronous product is choosen to represent the interaction of processes for the following reasons:

- Buffers are taken as the identity morphisms. This is in accordance both with *synchronous* processes, where buffers are without delay – they behave like hardware wires, also in the case where are buffered processes, in which they are insensitive to delay. Also it satisfies *asynchronous* case, where identity morphisms are also synchronous buffers.
- Milner‘s synchronous calculus SCCS is very expressive. Asynchronous calculi such as CCS and CSP can be derived from SCCS. Therefore we can take synchronous interaction as a basic notion.

Instead of considering labels to be appropriate names, a typed framework [12] is used to take a more structural view of concurrent processes. Interpretations of type constructors in **interaction categories** require set-theoretic constructions on the set of labels (sorts) associated with each type. A cartesian *product* of sorts (pairing of labels) is used to express the concurrent execution of some distributed actions. *Coproduct* is used to tag actions to allow controlled choices. Multisets of actions are used to support *replication* of processes. Product, coproduct and multisets represent in the notions of linear types [16] multiplicatives, additives and exponentials respectively. In that way we can generate such a set of categorical combinators for process algebra, which is free of labels. Therefore we should use categorical combinators for the translation of functional programs in a variable-free fashion.

Interaction categories clearly distinguish processes (computational entities) and specifications (logical entities).

Hoare in CSP considers processes with one input and one output, designed to be connected in a pipeline – this is very close to the view indicated in interaction categories. The same *divergence* problem arises in the case of interaction categories as in CSP. Two conditions are defined to avoid this situation. For the process $p; q$ must hold that p have to be *left-guarded* and q *right-guarded*. In that case p cannot perform an infinite sequence of actions in its right port without doing some actions in its left port; process q is defined symmetrically. These conditions ensure that the process $p; q$ does not diverge. Therefore if we adjust this idea to interaction categories, then we require all morphisms to be left- and right-guarded, so that all composites are non-divergent.

Here we would like to compare categories **SProc** and **Buf**. In contrast to the category **SProc**, processes $A \to B$ in the category **Buf** are oriented – channels in A are input channels, these in B are output channels.

In **SProc** the identity process is a process that continually offers to do the same action on both sides of its interface — it can be seen as a buffer that immediately sends on any message it receives. Because it is *synchronous*, the receive and the send actions happen at the same time, and so it cannot be distinguished whether a message was sent through the buffer or not. In an *asynchronous* setting a buffer will not generally work as an identity for composition.

Acknowledgement

This work was supported by VEGA Grant No.1/0175/08: Behavioral categorical models for complex program systems.

References

[1] Abramsky, S. What are the fundamental structures of concurrency? we still don't know! In *Electronic Notes in Theoretical Computer Science, 162* (2006), pp. 37–41.

[2] Abramsky, S., Gay, S., and Nagarajan, R. Interaction categories and the foundations of typed concurrent programming. In *Proceedings of the NATO Advanced Study Institute on Deductive program design* (Secaucus, NJ, USA, 1996), Springer-Verlag New York, Inc., pp. 35–113.

[3] Baeten, J. C. M. A brief history of process algebra. *Theor. Comput. Sci. 335*, 2-3 (2005), 131–146.

[4] Brauer, W., Reisig, W., and Rozenberg, G., Eds. *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, 8.-19. September 1986* (1987), vol. 255 of *Lecture Notes in Computer Science*, Springer.

[5] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice Hall, 1985.

[6] Joyal, A., Street, R., and Verity, D. Traced monoidal categories. *Math. Proc. Cambridge Philos. Soc. 119*, 3 (1996), 447–468.

[7] Milner, R. *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science.* Springer-Verlag, Berlin, 1980.

[8] Milner, R. *Communication and Concurrency.* 1989.

[9] Milner, R. Functions as processes. In *ICALP* (1990), pp. 167–180.

[10] Parrow, J. *An Introduction to the π-Calculus, in The Handbook of Process Algebra.* Elsevier, Amsterdam, 2001, p. 479.

[11] Ross, B. J. Mwsccs: A stochastic concurrent music language. In *In: Proc. II Brazilian Symposium on Computer Music* (1995).

[12] Schweimeier, R. A categorical framework for typing ccs-style process communication. *Electr. Notes Theor. Comput. Sci. 68*, 1 (2002).

[13] Selinger, P. First-order axioms for asynchrony. In *International Conference on Concurrency Theory* (1997), pp. 376–390.

[14] Selinger, P. Categorical structure of asynchrony. *Electr. Notes Theor. Comput. Sci. 20* (1999).

[15] Verbová, A., Hužvár, R., and Slodičák, V. On describing asynchronous processes by traced monoidal categories. In *Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering* (2008), elfa, s.r.o. Košice, pp. 99–106.

[16] Verbová, A., Novitzká, V., and Slodičák, V. From linear sequent calculus to proof nets. In *Informatics 2007, Proceedings of the Ninth International Conference on Informatics* (2007), Slovak Society for Applied Cybernetics and Informatics Bratislava, pp. 100–107.

[17] Winskel, G., and Nielsen, M. Models for concurrency. In *Handbook of Logic in Computer Science*, S. Abramsky, D. Gabbay, and T. S. E. Maibaum, Eds. Oxford University Press, 1995.

Department of Computers and Informatics, Technical University of Košice, Slovakia
E-mail address: `anita.verbova@tuke.sk`

Department of Computers and Informatics, Technical University of Košice, Slovakia
E-mail address: `robert.huzvar@tuke.sk`

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

A PARTITIONAL CLUSTERING ALGORITHM FOR IMPROVING THE STRUCTURE OF OBJECT-ORIENTED SOFTWARE SYSTEMS

ISTVAN GERGELY CZIBULA AND GABRIELA CZIBULA

Abstract. In this paper we are focusing on the problem of program restructuring, an important process in software evolution. We aim at introducing a partitional clustering algorithm that can be used for improving software systems design. The proposed algorithm improve several clustering algorithms previously developed in order to recondition the class structure of a software system. We experimentally validate our approach and we provide a comparison with existing similar approaches.

1. Introduction

The software systems, during their life cycle, are faced with new requirements. These new requirements imply updates in the software systems structure, that have to be done quickly, due to tight schedules which appear in real life software development process. That is why continuous restructuring of the code is needed, otherwise the system becomes difficult to understand and change, and therefore it is often costly to maintain. Without continuous restructurings of the code, the structure of the system becomes deteriorated. Thus, *program restructuring* is an important process in software evolution.

A continuous improvement of the software systems structure can be made using *refactoring*, that assures a clean and easy to maintain software structure.

We have previously introduced in [6] a clustering approach for identifying refactorings in order to improve the structure of software systems. For this purpose, a clustering algorithm named *kRED* was introduced. To our knowledge, there is no approach in the literature that uses clustering in order to improve the class structure of a software system, excepting the approach introduced in

Received by the editors: November 10, 2008.

2000 *Mathematics Subject Classification.* 68N99, 62H30.

1998 *CR Categories and Descriptors.* D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement –*Restructuring, reverse engineering, and reengineering*; I.5.3 [**Computing Methodologies**]: Pattern Recognition – *Clustering*.

Key words and phrases. Software design, Refactoring, Clustering.

[6]. The existing clustering approaches handle methods decomposition [26] or system decomposition into subsystems [13].

We have improved the approach from [6] by developing several clustering algorithms that can be used to identify the refactorings needed in order to recondition the class structure of an object-oriented software system [3–5, 20, 21].

The aim of this paper is to introduce a partitional clustering algorithm which takes an existing software and reassembles it, in order to obtain a better design, suggesting the needed refactorings. The clustering algorithm proposed in this paper improves all the algorithms that we have already developed.

The rest of the paper is structured as follows. Section 2 presents the main aspects related to the clustering approach (*CARD*) for determining refactorings [6] that we intend to improve in this paper. A new partitional clustering algorithm for determining refactorings is introduced in Section 3. Section 4 presents experimental evaluations of the proposed approach: the open source case study JHotDraw [10] and a real software system. Some conclusions and further work are given in Section 6.

2. Background

We have previously introduced in [6] a clustering approach (*CARD*) in order to find adequate refactorings to improve the structure of software systems. *CARD* approach consists of the following steps:

(1) The existing software system is analyzed in order to extract from it the relevant entities: classes, methods, attributes and the existing relationships between them: inheritance relations, aggregation relations, dependencies between the entities from the software system.

(2) The set of entities extracted at the previous step are re-grouped in clusters (classes) using a clustering algorithm (*PARED* in our approach). The goal of this step is to obtain an improved structure of the existing software system.

(3) The newly obtained software structure is compared with the original software structure in order to provide a list of refactorings which transform the original structure into an improved one.

3. A Partitional Clustering Algorithm for Refactorings Determination (PARED)

In this section we introduce a new partitional clustering algorithm (*PARED*) (Partitional Clustering Algorithm for Refactorings Determination). *PARED* algorithm can be used in the **Grouping** step of *CARD* in order to identify a

partition of a software system S that corresponds to an improved structure of it.

In our clustering approach, the objects to be clustered are the entities from the software system S, i.e., $\mathcal{O} = \{s_1, s_2, \ldots, s_n\}$. Our focus is to group similar entities from S in order to obtain high cohesive groups (clusters).

We will adapt the generic cohesion measure introduced in [22] that is connected with the theory of similarity and dissimilarity. In our view, this cohesion measure is the most appropriate to our goal. We will consider the dissimilarity degree between any two entities from the software system S. Consequently, we will consider the distance $d(s_i, s_j)$ between two entities s_i and s_j as expressed in Equation (1).

$$(1) \qquad d(s_i, s_j) = \begin{cases} 1 - \frac{|p(s_i) \cap p(s_j)|}{|p(s_i) \cup p(s_j)|} & \textit{if} \ \ p(s_i) \cap p(s_j) \neq \emptyset \\ \infty & \textit{otherwise} \end{cases},$$

where, for a given entity $e \in S$, $p(e)$ defines a set of relevant properties of e, expressed as follows. If $e \in \mathit{Attr}(S)$ (e is an attribute) then $p(e)$ consists of: the attribute itself, the application class where the attribute is defined, and all the methods from $\mathit{Meth}(S)$ that access e. If $e \in \mathit{Meth}(S)$ (e is a method) then $p(e)$ consists of: the method itself, the application class where the method is defined, all the attributes from $\mathit{Attr}(S)$ accessed by the method, all the methods from S used by e, and all methods from S that overwrite method e. If $e \in \mathit{Class}(S)$ (e is an application class) then $p(e)$ consists of: the application class itself, all the attributes and the methods defined in the class, all interfaces implemented by class e and all classes extended by class e.

Our distance, as defined in Equation (1), highlights the concept of cohesion, i.e., entities with low distances are cohesive, whereas entities with higher distances are less cohesive.

Based on the definition of distance d (Equation (1)) it can be easily proved that d is a semi-metric function, so a *k-medoids* based approach can be applied.

In order to avoid the two main disadvantages of the traditional *k-medoids* algorithm, *PARED* algorithm uses a heuristic for choosing the number of medoids (clusters) and the initial medoids. This heuristic is particular to our problem and it will provide a good enough choice of the initial medoids.

After selecting the initial medoids, *PARED* behaves like the classical *k-medoids* algorithm.

The main idea of *PARED*'s heuristic for choosing the initial medoids and the number p of clusters (medoids) is the following:

(i) The initial number p of clusters is n (the number of entities from the software system) and the intial number nr of medoids is 0.

(ii) The entity chosen as the first medoid is the most "distant" entity from the set of all entities (the entity that maximizes the sum of distances from all other entities). The number *nr* of medoids becomes 1.

(iii) In order to choose the next medoid we reason as follows. For each remaining entity (that was not chosen as medoid), we compute the minimum distance (*dmin*) from the entity and the already chosen medoids. The next medoid is chosen as the entity e that maximizes *dmin* and this distance is greater than a positive given threshold (*distMin*), and *nr* is increased. If such an entity does not exist, it means that e is very close to all the medoids and should not be chosen as a new medoid (from the software system structure point of view this means that e should belong to the same application class with an existing medoid). In this case, the number p of medoids will be decreased.

(iv) The step (iii) will be repeatedly performed, until the number *nr* of chosen medoids is equal to p.

We have to notice that step (iii) described above assures, from the software system design point of view, that near entities (with respect to the given threshold *distMin*) will be merged in a single application class (cluster), instead of being distributed in different application classes.

We mention that at steps (ii) and (iii) the choice could be a non-deterministic one. In the current version of *PARED* algorithm, if such a non-deterministic case exists, the first selection is made. Future improvements of *PARED* algorithm will deal with these kind of situations.

The main idea of the *PARED* algorithm that we apply in order to group entities from a software system is the following:

(i) The initial number p of clusters and the initial medoids are determined by the heuristic described above.

(ii) The clusters are recalculated, i.e., each object is assigned to the closest medoid.

(iii) Recalculate the medoid i of each cluster k based on the following idea: if h is an object from k such that $\sum\limits_{j \in k}(d(j,h) - d(j,i))$ is negative, then h becomes the new medoid of cluster k.

(iv) Steps (ii)-(iii) are repeatedly performed until there is no change in the partition $\mathcal{K}$.

We mention that *PARED* algorithm provides a partition of a software system S, partition that represents a new structure of the software system. Regarding to *PARED* algorithm, we have to notice the following:

- If, at a given moment, a cluster becomes empty, this means that the number of clusters will be decreased.

- Because the initial medoids are selected based on the heuristic described above, the dependence of the algorithm on the initial medoids is eliminated.
- We have chosen the value 1 for the threshold *distMin*, because distances greater than 1 are obtained only for unrelated entities (Equation (1)).

The main refactorings identified by *PARED* algorithm are *Move Method*, *Move Attribute*, *Inline Class*, *Extract Class* [9]. We have currently implemented the above enumerated refactorings, but *PARED* algorithm can also identify other refactorings, like: *Pull Up Attribute*, *Pull Down Attribute*, *Pull Up Method*, *Pull Down Method*, *Collapse Class Hierarchy.* Future improvements will deal with these situations, also.

4. Experimental Evaluation

In order to experimentally validate our clustering approach, we will consider two evaluations, which are described below.

Our first evaluation is the open source software JHotDraw, version 5.1 [10]. It is a Java GUI framework for technical and structured graphics, developed by Erich Gamma and Thomas Eggenschwiler, as a design exercise for using design patterns. It consists of **173** classes, **1375** methods and **475** attributes. The reason for choosing JHotDraw as a case study is that it is well-known as a good example for the use of design patterns and as a good design.

Our focus is to test the accuracy of our approach on JHotDraw, i.e., how accurate are the results obtained after applying *PARED* algorithm in comparison with the current design of JHotDraw. As JHotDraw has a good class structure, *PARED* algorithm should generate a nearly identical class structure.

After applying *PARED* algorithm, we have obtained a partition in which there are no misplaced methods and attributes, meaning that the class structure discovered by *PARED* is identical to the actual structure of JHotDraw.

Our second evaluation is a DICOM (*Digital Imaging and Communications in Medicine*) [8] and HL7 (*Health Level 7*) [11] compliant PACS (*Picture Archiving and Communications System*) system, facilitating medical images management, offering access to radiological images, and making the diagnosis process easier. We have applied *PARED* algorithm on one of the subsystems from this application, subsystem containing **1015** classes, **8639** methods and **4457** attributes.

After applying *PARED* algorithm, a total of 84 refactorings have been suggested: 7 *Move Attribute* refactorings, 75 *Move Method* refactorings, and

2 *Inline Class* refactoring. From the refactorings obtained by *PARED* algorithm, 55% were accepted by the developers of the considered software system.

Analyzing the obtained results, we have concluded that a large number of miss-identified refactorings are due to technical issues: the use of Java anonymous inner classes, introspection, the use of dynamic proxies. These kind of technical aspects frequently appear in projects developed in JAVA. In order to correctly deal with these aspects, we have to improve only the data collection step from our approach, without modifying *PARED* algorithm. Another cause of miss-identified refactorings is due to the fact that the *distance* (Equation (1)) used for discriminating entities in the clustering process take into account only two aspects of a good design: *low coupling* and *high cohesion*. It would be also important to consider other principles related to an improved design, like: *Single Responsibility Principle*, *Open-Closed Principle*, *Interface Segregation Principle*, *Common Closure Principle* [7], etc. Future improvements of our approach will deal with these aspects, also.

5. Related Work

In this section we present some approaches existing in the literature in the fields of *software clustering* and *refactoring*. We provide, for similar approaches, a comparison with our approach.

There is a lot of work in the literature in the field of *software clustering*.

One of the most active researches in the area of software clustering were made by Schwanke. The author addressed the problem of automatic clustering by introducing the *shared neighbors* technique [17], technique that was added to the low-coupling and high-cohesion heuristics in order to capture patterns that appear commonly in software systems. In [18], a partition of a software system is refined by identifying components that belong to the wrong subsystem, and by placing them in the correct one. The paper describes a program that attempts to reverse engineer software in order to better provide software modularity. Schwanke assumes that procedures referencing the same name must share design information on the named item, and are thus "design coupled". He uses this concept as a clustering metric to identify procedures that should be placed in the same module. Even if the approaches from [17] and [18] were not tested on large software systems, they were promising.

Mancoridis et al. introduce in [14] a collection of algorithms that facilitate the automatic recovery of the modular structure of a software system from its source code. Clustering is treated as an optimization problem and genetic algorithms are used in order to avoid the local optima problem of *hill-climbing* algorithms. The authors accomplish the software modularization process by constructing a *module dependency graph* and by maximizing an objective function based on inter- and intra-connectivity between the software components. A clustering tool for the recovery and the maintenance of software system structures, named *Bunch*, is developed. In [15], some extensions of *Bunch* are presented, allowing user-directed clustering and incremental software structure maintenance.

A variety of software clustering approaches have been presented in the literature. Each of these approaches looks at the software clustering problem from a different angle, by either trying to compute a measure of similarity between software objects [17]; deducing clusters from file and procedure names [1]; utilizing the connectivity between software objects [2, 12, 16]; or looking at the problem at hand as an optimization problem [14]. Another approach for software clustering was presented in [1] by Anquetil and Lethbridge. The authors use common patterns in file names as a clustering criterion. The authors' experiments produced promising results, but their approach relies on the developers' consistency with the naming of their resources.

The paper [24] also approaches the problem of software clustering, by defining a metric that can be used in evaluating the similarity of two different decompositions of a software system. The proposed metric calculates a distance between two partitions of the same set of software resources. For calculating the distance, the minimum number of operations (such as moving a resource from one cluster to another, joining two clusters etc.) one needs to perform in order to transform one partition to the other is computed. Tzerpos and Holt introduce in [25] a software clustering algorithm in order to discover clusters that follow patterns that are commonly observed in decompositions of large software systems that were prepared manually by their architects.

All of these techniques seem to be successful on a number of examples. However, not only is there no approach that is widely recognized as superior, but it is also hard to compare the effectiveness of different approaches. As presented above, the approaches in the field of *software clustering* deal with the software decomposition problem. Even if similarities exist with refactorings extraction, a comparison is hard to make due to the different granularity of the decompositions (modules vs. classes, methods, fields).

There were various approaches in the literature in the field of *refactoring*, also. But, only very limited support exists in the literature for automatic refactorings detection.

For most existing approaches, the obtained results for relevant case studies are not available. There are given only short examples indicating the obtained refactorings. That is why we have selected for comparison only two techniques mentioned below.

The paper [23] describes a software vizualization tool which offers support to the developers in judging which refactoring to apply. We have applied *PARED* algorithm on the example given in [23] an the *Move Method* refactoring suggested by the authors was obtained.

A search based approach for refactoring software systems structure is proposed in [19]. The authors use an evolutionary algorithm for identifying refactorings that improve the system structure.

The advantages of our approach in comparison with the approach presented in [19] are illustrated bellow. Our technique is deterministic, in comparison with the approach from [19]. The evolutionary algorithm from [19] is executed **10** times, in order to judge how stable are the results, while *PARED* algorithm from our approach is executed just **once**. The technique from [19] reports **11** misplaced methods, while in our approach there are **no** misplaced methods. The overall running time for the technique from [19] is about **300** minutes (30 minutes for one run), while *PARED* algorithm in our approach provide the results in about **1.2** minutes. We mention that the execution was made on similar computers. Because the results are provided in a reasonable time, our approach can be used for assisting developers in their daily work for improving software systems.

6. Conclusions and Future Work

We have presented in this paper a new partitional clustering algorithm (*PARED*) that can be used for improving software systems design. We have demonstrated the potential of our algorithm by applying it to the open source case study JHotDraw and to a real software system, and we have also presented the advantages of our approach in comparison with existing approaches. Based on the feedback provided by the developers of a real software system we have identified some potential improvements of our approach.

Further work will be done in the following directions: to use other search based approaches in order to determine refactorings that improve the design of a software system; to improve the *distance* function used in the clustering process; to apply *PARED* algorithm on other large software systems; to apply our approach in order to transform non object-oriented software into object-oriented systems.

ACKNOWLEDGEMENT

This work was supported by the research project TD No. 411/2008, sponsored by the Romanian National University Research Council (CNCSIS).

References

[1] Nicolas Anquetil and Timothy Lethbridge, *Extracting concepts from file names; a new file clustering criterion*, 20th International Conf. Software Engineering, 1998, pp. 84–93.

[2] Song C. Choi and Walt Scacchi, *Extracting and restructuring the design of large systems*, IEEE Softw. **7** (1990), no. 1, 66–71.

[3] I.G. Czibula and G. Serban, *A hierarchical clustering algorithm for software systems design improvement*, KEPT 2007: Proceedings of the first International Conference on Knowledge Engineering: Principles and Techniques, August 2007June 6, pp. 316–323.

[4] I. G. Czibula and G. Serban, *Hierarchical clustering for software systems restructuring*, INFOCOMP Journal of Computer Science, Brasil **6** (2007), no. 4, 43–51.

[5] I.G. Czibula and G. Serban, *Software systems design improvement using hierarchical clustering*, SERP'07: Proceedings of SERP'07, 2007, pp. 229–235.

[6] Istvan G. Czibula and Gabriela Serban, *Improving Systems Design Using a Clustering Approach*, International Journal of Computer Science and Network Security (IJCSNS) **6** (2006), no. 12, 40–49.

[7] Tom DeMarco, *Structured analysis and system specification* (2002), 529–560.

[8] *Digital Imaging and Communications in Medicine.* http://medical.nema.org/.

[9] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts, *Refactoring: Improving the design of existing code*, Addison-Wesley, Reading, MA, USA, 1999.

[10] E. Gamma, *JHotDraw Project.* http://sourceforge.net/projects/jhotdraw.

[11] *Health Level 7.* www.hl7.org/.

[12] David H. Hutchens and Victor R. Basili, *System structure analysis: clustering with data bindings*, IEEE Trans. Softw. Eng. **11** (1985), no. 8, 749–757.

[13] Chung-Horng Lung, *Software architecture recovery and restructuring through clustering techniques*, Isaw '98: Proceedings of the third International Workshop on Software Architecture, 1998, pp. 101–104.

[14] S. Mancoridis, B. S. Mitchell, C. Rorres, Y. Chen, and E. R. Gansner, *Using automatic clustering to produce high-level system organizations of source code*, IEEE Proceedings of the 1998 int. Workshop on Program Understanding (IWPC'98), 1998, pp. 45–52.

[15] Spiros Mancoridis, Brian S. Mitchell, Yih-Farn Chen, and Emden R. Gansner, *Bunch: A clustering tool for the recovery and maintenance of software system structures*, ICSM, 1999, pp. 50–59.

[16] James M. Neighbors, *Finding reusable software components in large systems*, Working Conference on Reverse Engineering, 1996, pp. 2–10.

[17] R. W. Schwanke and M. A. Platoff, *Cross references are features*, Proceedings of the 2nd International Workshop on Software Configuration Management, 1989, pp. 86–95.

[18] Robert W. Schwanke, *An intelligent tool for re-engineering software modularity*, ICSE '91: Proceedings of the 13th International Conference on software engineering, 1991, pp. 83–92.

[19] Olaf Seng, Johannes Stammel, and David Burkhart, *Search-based determination of refactorings for improving the class structure of object-oriented systems*, GECCO '06: Proceedings of the 8th annual conference on genetic and evolutionary computation, 2006, pp. 1909–1916.

[20] G. Serban and I.G. Czibula, *A new clustering approach for systems designs improvement*, SETP-07: Proceedings of the International Conference on Software Engineering Theory and Practice, December 2007 July 9, pp. 47–54.

[21] G. Serban and I. G. Czibula, *Restructuring software systems using clustering*, ISCIS 2007: Proceedings of the 22nd International Symposium on Computer and Information Sciences, September 2007 November 7, pp. 33, IEEExplore.

[22] Frank Simon, Silvio Loffler, and Claus Lewerentz, *Distance based cohesion measuring*, Proceedings of the 2nd European Software Measurement Conference (FESMA), 1999, pp. 69–83.

[23] Frank Simon, Frank Steinbruckner, and Claus Lewerentz, *Metrics based refactoring*, CSMR '01: Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, 2001, pp. 30–38.

[24] Vassilios Tzerpos and Richard C. Holt, *Mojo: A distance metric for software clusterings*, Working conference on reverse engineering, 1999, pp. 187–193.

[25] Vassilios Tzerpos and Richard C. Holt, *ACDC: An algorithm for comprehension-driven clustering*, Working conference on reverse engineering, 2000, pp. 258–267.

[26] Xia Xu, Chung-Horng Lung, Marzia Zaman, and Anand Srinivasan, *Program restructuring through clustering techniques*, SSAM '04: Proceedings of the Workshop on source code analysis and manipulation, Fourth IEEE International (SCAM'04), 2004, pp. 75–84.

Department of Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu Street, Cluj-Napoca, Romania,
E-mail address: istvanc@cs.ubbcluj.ro

Department of Computer Science, Babeş-Bolyai University 1, M. Kogălniceanu Street, Cluj-Napoca, Romania,
E-mail address: gabis@cs.ubbcluj.ro

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

VIRTUAL ORGANIZATIONS IN EMERGING VIRTUAL 3D WORLDS

DUMITRU RĂDOIU

Abstract. Our paper explores virtual organizations supported by emerging virtual world platforms, analysing them in the perspective of the supporting technology. The shortcomings of the used paradigms are identified as well as new directions for research. The paper concludes that, in order for virtual organizations to take full advantage of virtual world platforms, a new architecture based on open standards is needed, a new in-world paradigm to secure intellectual property and an agent-web service gateway to allow the composition of services between virtual worlds and Web.

1. The Problem

Emerging virtual worlds (VW) push the experience from 2D to 3D, from flat to immersive, from one-on-one to social. As all human beings live in a 3D real world (RW) and our experiences in virtual VW closely parallel our real life experiences, we witness an accelerated acceptance speed of VW platforms and 3D GUIs. The mix of grid computing, physics engine and spatial data, that enable virtual worlds, is also becoming also more powerful and well-fit to disrupt the present social and economic landscape. The anticipated huge impact on IT, business, and society in the very near future makes this field worth researching.

In the last few years, based on these new VW platforms, new virtual organization (VO) models have been developed. Our paper explores these new models analysing them in the perspective of the supporting technology.

2. The Concepts

As there is no large agreement in the available literature on the terminology, we've considered useful introducing the following definitions.

Real World (RW): Physical World, Universe

Digital World (DW) 2D Web, Internet

Received by the editors: June 15, 2008.

2000 *Mathematics Subject Classification.* 68N30.

1998 *CR Categories and Descriptors.* D.2.9 [**Software Engineering**]: Management – *Software process models*.

Key words and phrases. Virtual organization, Virtual worlds, Metaverse.

Virtual World (VW): a fully immersive 3D virtual space. VWs use the metaphor of the real world, but without its physical limitations.

Avatar (AV) A 3d representation of an agent, operating in a VW, also called "digital persona". Avatars with facial expressions and body language provide a virtual experience almost as rich as real-life.

3D Web: VW interconnected

Metaverse: A Virtual World that has primarily social and economic role. Users (represented in-world by agents/avatars interact with each other (socially and economically) and with other software agents. Metaverse characteristics:

- Scalability
- Access levels: from low quality to very high
- Face to face (F2F) communication
- Code protocols as law: coding protocols define what can and cannot be done, what is legal, what is not
- Economics

Paraverse: A Virtual World linked to regions and/or bodies in the RW (e.g. Google Earth, virtual surgery or virtual shared meeting places)

Intraverse: A Virtual World built behind a firewall (concept similar to 2D intranet). A grid of a company, the region domain only allows agents from their agent domain to connect, and they can be sure that all those people in their agent domain are actually employees.

Open Ended VW (OEVW): A Virtual World in which **residents** (represented by avatars) use communication, available co-operation services and their skills to involve in social and/or economic activities. These virtual worlds exist simply as places to explore, experience, create and, based on IP Intellectual Property, to exchange goods and values (i.e. to conduct commerce).

Open Ended VW are currently developing relationships (economic, social, cultural and legal) with the RW. Here are some similarities between RW and OEVW socio-economic features: Innovation and Intellectual property (IP), market (goods and values exchange, from both RW and VW), currency, financial organizations, face to face (F2F) communication, identification and authorisation, mass-media, education organizations, political organizations.

The technological features available now in OEVW which support co-operation are: voice communication, audio and video feeds, instant messaging, file exchange, encryption.

Closed Ended VW (CEVW): A Virtual World (the stage) in which **players** (represented by avatars as alter egos) use communication, available co-operation services and their skills to involve activities related to a **scenario**. These virtual worlds have as goal or purpose a **game**.

In this paper, we will analyse only virtual organizations operating in open-ended virtual worlds.

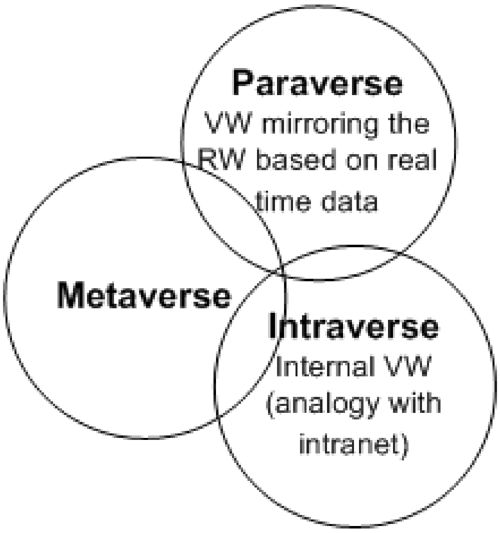

Paraverse
VW mirroring the RW based on real time data
Metaverse
Intraverse
Internal VW (analogy with intranet)

FIGURE 1. Open-ended virtual worlds

The paper addresses virtual organizations (VO) in open ended virtual worlds. The VO definition with which we operate is:

Virtual Organization (VO) is an organization with the following characteristics [1]:

- spatial: operates within geographically distributed work environments
- temporal: has a limited life span, until it performs its tasks or actions
- configurational: uses information and communication technology ICT to enable, maintain and sustain member relationships

VO in VW are those virtual organizations which allow its members to work in a 3D immersive environment by emulating face-to-face communication with colleagues.

3. The Platform

We start from a generic architecture [2] of a VO operating in 2D and 3D (Figure 2) the dotted line representing the focus area of this paper.

Most of the VW run on proprietary collaboration platform, not open yet (only the client); exception OpenSimulator and collaboration services run with disruption.

The OpenSimulator Project is an open source Virtual Worlds Server which can be used for creating and deploying 3D Virtual Environments, able to run both in a standalone mode or connected to other OpenSimulator instances through built in grid technology. It can be extended to produce more specialized 3D interactive applications via plug-in modules. Several OEVW were

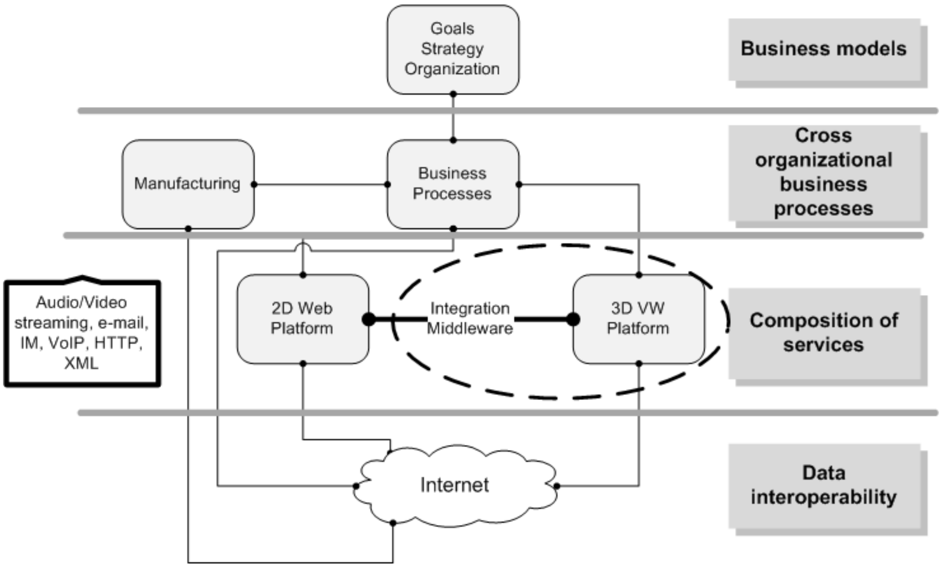


FIGURE 2. A generic architecture of a VO operating in 2D and 3D

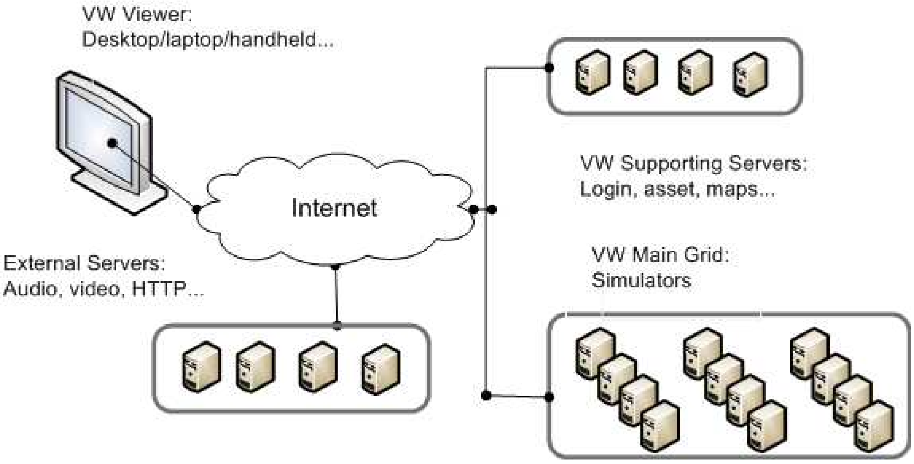


FIGURE 3. A generic, proprietary VW infrastructure

built with open source technology from the open simulator project (Openlife Grid, DeepGrid, OSGrid, 2008)

OpenSimulator uses *libsecondlife* to handle communication between the client and server, so it is possible to connect to an OpenSim server using the

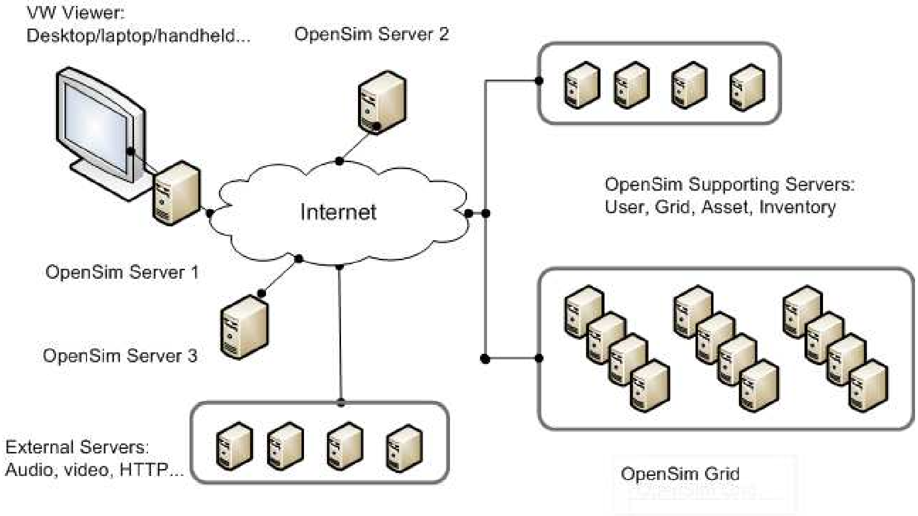


FIGURE 4. A generic, open VW infrastructure

Linden Lab Second Life viewer. Second Life (SL) is the largest proprietary metaverse, owned by Linden Research, Inc.

OpenSimulator operates in one of two modes: standalone or grid mode. In standalone mode, a single process handles the entire simulation. In grid mode, various aspects of the simulation are separated among multiple processes, which can exist on different machines. In grid mode, responsibilities are divided among five servers: the user server, the grid server, the asset server, the inventory server and the simulation server.

3.1. **Platform features supporting collaboration. Security:** All VW platforms include a number of security levels. One is based on the ability to secure the deployment and access of the collaboration infrastructure. The second level is based on the collaboration session itself and the ability to secure the access to a session and/or encrypt the data communication channel. You can encrypt IM/Chat and/or Video and/or Audio from your viewer to the grid. Second Life (SL) uses standard JAVA encryption libraries (JCE).

Communication: Many VW platforms are voice enabled adding more to the "realism" of F2F communication. Text Chat, Instant Messaging, and E-mail are common features in almost all VW platforms.

Movement: Features like teleport and location are also common.

Transfer: Notes, Files/objects/money transfer are available

Interface with 2D Web: Audio-video streaming, instant messaging, e-mail, VoIP, XML from 2D Web to virtual worlds are available

3.2. **Platform issues with regard to collaboration. Scalability:** Present VW architectures present a limited scalability which does not support the expected increased number in regions, users/residents and concurrency (number of users simultaneously connected to the VW). Second Life estimates for the next ten years a growth to 50 million regions, 2 billion users and 50 million concurrencies. As a first step, scalability was addressed by the subdivision of the metaverse into fix sized regions, of 256x256 m, each being emulated by a simulator running on one CPU core. At this moment, the simulator handles everything that happens in the region, avatar agents included. Because there's a limit in what a processor can handle, Linden Lab is considering as a next step be a separation between agents and regions into two separate domains: the agent domain and the region domain. The agent domain knows everything about an agent: name, profile, inventory etc. This halves the load of the CPU. The agent domain consists of some web services which allows to login, to retrieve inventory etc. The region domain consists of a number of simulators and knows everything about regions: their name, location, and what's on them. The viewer needs to connect to both domains to first login the agent and then connect to the region.

Standards: There's no standard yet for VW; you cannot host your own simulator connected to a different main region grid (than the one you belong to and which "recognises your avatar: its identity, inventory, and payment info.

The solution is obviously an open standard for VW, an open architecture allowing the development of 3D Web, grid architecture similar to the web where everybody can connect their own server.

Interoperability: The metaphor used in 2D Web is that of services [3], while the one used in VW is that of agents. The two domains use different directory services, different transport services and different languages (syntax and semantics). Web services aim is to enable dynamic service discovery, composition, invocation, execution and monitoring. Software agents – on the other hand – are designed as autonomous, proactive entities. Software agents have been envisioned as potential user of semantic Web services in order to interact with semantic descriptions of SWS to autonomously discover, select, compose, invoke and execute the services based on user requirements [4]. The communication gap between the two worlds resides in the fact that software agents are not compatible with widely accepted standards of Web services. Research is conducted [4] to make multi agent systems compatible with existing Web services standards without changing the existing specifications and implementations.

At the moment, with no interoperability between software agents and semantic Web services, most of VOs operate either in 2D Web basing their processes on Web services or completely in-world. Further research and standardisation is needed on the Agent-Web gateway, to enable interoperability between 2D and the future 3D Web.

4. Virtual organizations in virtual worlds

Open ended virtual worlds are platforms for three key functions: social interactions, business, and entertainment. Social interactions and entertainment are the most visible. In SL for instance, there are16 million users, almost 50000 concurrent users at any given moment, millions of dollars businesses, hundreds of universities, virtual embassies, thousands of companies. An entire economy exists, facilitated by intellectual property and virtual world banks.

Social events participation is limited only by the simulators concurrency limit. They are so successful because they are face to face (F2F), voice enabled events, with interactive sharing, allowing an almost real life interaction.

The most visible reasons for businesses for establishing a presence in VW are:

- to extend their brand into a virtual world (information centres, training, interactive demonstrations, virtual 3D stores, collecting data on shopping experience, customer feedback, free market research)
- to brand engagement (e.g. witness the construction of your own laptop or desktop computer while you interact and select components)
- to engage in virtual worlds specific new businesses (e.g. terra-forming, building, creation and scripting)

The real huge advantage for virtual organizations is face to face, voice enabled real time communication.

We can distinguish between virtual organizations which processes are based entirely on the interoperability provided by the virtual world platform and virtual organization whose processes span over both virtual world and digital world.

4.1. **Virtual data centre.** IBM had built a 3D data centre application in an effort to leverage VW capabilities to RL business processes, thus gaining a competitive edge. RW data centres (serviced by IBM) are connected to a VW data centre which mirrors the real environment. The virtual world platforms that render the 3-D environment is based on the OpenSim Application Platform for 3D Virtual Worlds. The VW data center comprises of models of RW equipment and facilities such as servers, racks, network equipment, and power and cooling equipment. The VW models receive data from live RW enterprise management systems (IBM Director, Enterprise Workload Manager, Tivoli Omegamon, and MQ Series). Live RW information is aggregated (using VW SDK) and presented in 3D. Functions like power control and virtual machine migration can be performed completely in world, managers being able to respond quickly to alerts and events on demand. The 3D data center allows an intuitive visual inspection of how the real data centre is performing. Specific VW effects (sounds, particle effects) are used to visualize if there are network or server issues allowing event location in a timely manner [5]. Multiple users collaborate in-world, explore the operations in 3D in near real time, take part in the analysis and the decision making process. The Intraverse solution (privately hosted VW) was adopted for security reasons. Interoperability between RW and VW is provided by a proprietary virtual world middleware, named Holographic Enterprise Interface (HEI).

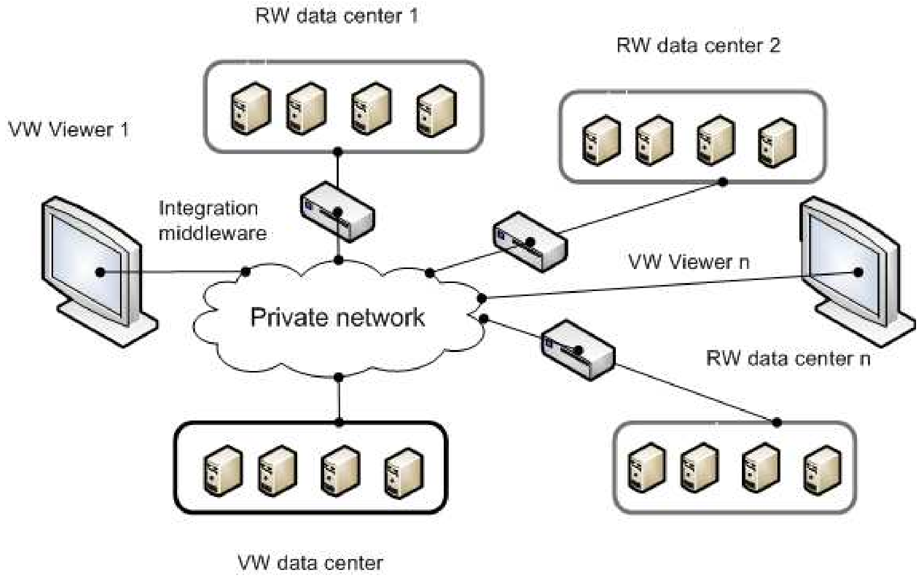


Figure 5. Intraverse 3D data centre architecture

IBM has more than 5,000 employees using VW for purposes such as sales training or collaborating across different geographic regions, showcase for different offerings, meetings with clients in current projects (virtual conferences), sales meetings, presenting concepts in a manner not attainable in RW or 2D Web (e.g. manipulation of 3D models).

5. Comments, preliminary conclusions and further research

To the above mentioned issues (3.2), we can add some more, like the limited access to VW (residents have a single access point, the PC, cross-platform online access from the large range of converged consumer electronics devices is still in the research phase. Yet, despite all these challenges, for many VW communities (e.g. SL) a certain kind of virtual economy has evolved.

Gartner [6] has opined that, by 2011, 80 percent Fortune 500 companies will have some kind of virtual world presence meaning that major transformation into how the organizations will interact in the near future will occur in the near future. It seems that at least for a good time from now on, the 3D platform will be completing the current web platform, rather than replacing it.

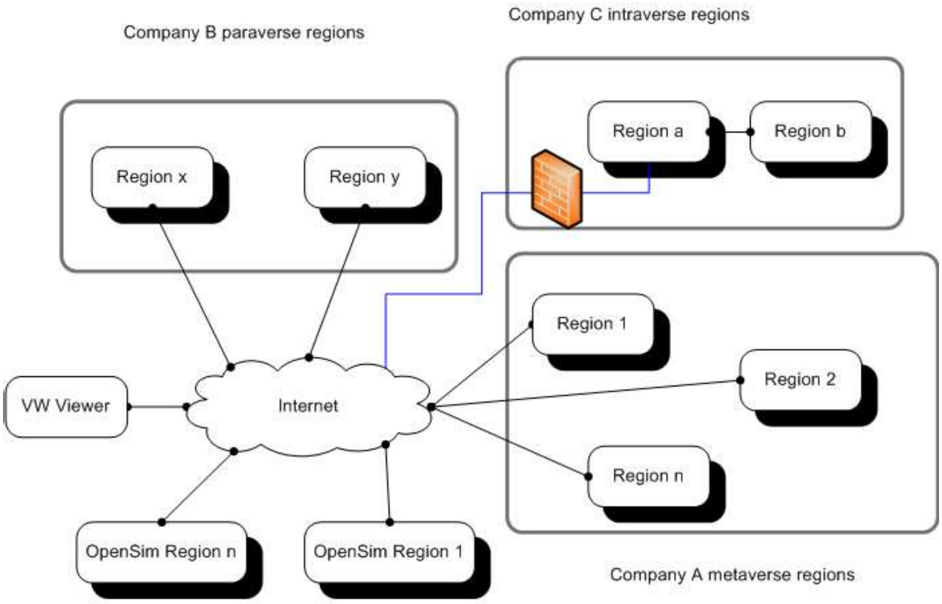


FIGURE 6. 3D Web topology

3D Web topology will look probably like the one depicted in Figure 6, with the enterprise-class virtual worlds running behind firewalls.

For such a topology to exist, a number of issues must be addressed:

- New trustful, open architecture enabling viewers to handle assets and inventory services
- Standard libraries used in communication with the viewer
- Portable identities (the same AV can travel in different virtual worlds); probably associated with an AV certification system (maybe through vendors of trusted agents)
- Standard interfaces between worlds

- O standardized software stack that will be portable outside and beyond VW
- Open standards for the representation of information
- Business level quality of in-world services (security, performance, reliability, stability, availability)
- API and SDK for developing custom business applications in-world
- A new way to address intellectual property, presently handled through permissions

Permissions represent a crude way to enforce licenses and can't anticipate all possible licensing scenarios. For instance, SL provides about everything needed to copy about anything in-world, excepting scripts. So, in an open 3D Web, if we attach permission to an object, that object permission could get ignored in some regions.

Virtual Worlds Web Integration is a growing research field which might lead to intertwining between the two despite the huge difference between the metaphors they are built on: agents vs. services. A detailed discussion of the research status in this area is behind the scope of this paper. But we can only imagine the impact it will have on the web as we know it: web pages empowered with immersive, presence-based features.

References

[1] Radoiu D, Contributions to Conceptual Modelling of Virtual Organizations, in submitted to 5th International Workshop on Grid Economics and Business Models, Las Palmas, Canary Island, Spain, 2008

[2] Karagiannis D., Kun H., Metamodelling Platforms, LNCS 2455, Spinger-Verlag 2002, p 182

[3] 3S Green Paper on Software and Service Architectures, Infrastructures and Engineering – a working document for the future EU Action paper on the area, version 1.2URL: www.eu-ecss.eu (Retrieved 30 March 2008)

[4] Omair Shafiq M., Ding Y., Fensel D., Bridging Multi Agent Systems and Web Services: towards interoperability between Software Agents and Semantic Web Services, Enterprise Distributed Object Computing Conference, 2006. EDOC apos;06. 10th IEEE International

[5] IBM Corporation: Made in IBM Labs: IBM 3-D Data Centers Show Virtual Worlds Fit for Business, Press Release, 21 February 2008, http://www-03.ibm.com/press/us/en/pressrelease/23565.wss (Retrieved 15 April 2008)

[6] The Gartner Scenario: Current Trends and Future Direction of the IT Industry, April 22-26, 2007, San Francisco, CA

Petru Maior University, 1, Nicolae Iorga, Târgu Mures, Romania
E-mail address: Dumitru.Radoiu@Sysgenic.com

STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LIII**, Number 2, 2008

ON SIMPLIFYING THE CONSTRUCTION OF EXECUTABLE UML STRUCTURED ACTIVITIES

C.-L. LAZĂR AND I. LAZĂR

Abstract. UML, with its Action Semantics package, allows the user to create object-oriented executable models. Creating such models, however, is a very difficult task, because the UML primitives are too fine-grained and because UML has many variation points. This article proposes a computationally complete subset of the Action Semantics and raises the level at which the user works, from actions to statements and expressions. New graphical notations are also proposed, so that the resulting structured activity diagram is more intuitive and clear.

1. Introduction

The Action Semantics package from UML [9] gives the user the possibility to create executable models [11]. Before, the behavior of an operation, for instance, had to be specified using an opaque expression, which means platform dependent code. The package supports many features, so that it may be used in different domains, not just in tasks similar to programming. The actions are also very flexible: the structured control nodes, for instance, are more general than the corresponding statements found in the most used programming languages [5].

The UML activities support both a structured action model and a flow action model, each one being more suited for a specific modeling task than another. They are equivalent only for small examples, and, in general, the functionality written in one action model can be converted in the other form, though not easy. The two action models are not independent of each other, as the structured action model mainly addresses control, and still needs flow to pass data between actions.

Received by the editors: October 20, 2008.

1991 *Mathematics Subject Classification.* 68N15, 68N30.

1998 *CR Categories and Descriptors.* D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques – *Computer-aided software engineering, Flow charts, Object-oriented design methods*.

Key words and phrases. UML, Action Semantics, Structured Activities, Action Language.

The structured model fits better with a textual notation style, which is usually designed for well nested control, using variables to pass data between actions, instead of data flow. The textual notation is what most programmers are used to, so the structured action model would be more suited for programmers using UML.

1.1. The Problem and Motivation.

It is very hard to use the Action Semantics package directly while trying to reproduce the functionality from a simple piece of code written in a programming language. This happens because the Action Semantics package supports too many features, which makes it hard to be learned, it has many variation points, which makes it hard to be used properly [13], and combining the actions inside an activity feels like working in an assembler language. Many implicit things from a programming language code have to be explicitly formulated in the UML model, which creates a need for a better tool support in this area.

Another problem with the Action Semantics is that no notations are given for many elements. In general, the graphical notation from Action is used, with different stereotypes. The graphical notations can be improved a lot, and this article proposes a new set of graphical notations. Also, textual notations may be used, but this is not covered here.

1.2. The Solution.

In this article we choose a well defined subset of the UML action semantics, in order to represent the structured activities, as this is still in the process of standardization [7]. The subset must be computationally complete, and have a precise behavior (as opposed to the semantic variation points from UML, which are many).

New graphical notations are introduced, which help create a clear and simplified view of the structured activity. The expressions, for instance, will be presented as an aggregate to the user, not as distinct UML objects, even though, behind the scenes, the expressions are represented using the UML model.

This article is meant to expand the Procedural Action Language model, the UML profile and the graphical notations proposed in [10].

2. Action Semantics (subset)

A Procedural Action Language (PAL) model was presented in previous articles [12, 6, 10]. A UML profile was also defined, so that the PAL model can be exchanged among UML compliant tools. This PAL model is used as a target of what are the desired capabilities of the chosen subset of Action Semantics, with certain deviations.

The new model moves away from the *procedural* aspect, to *object − oriented*. The structured Activity will no longer be done for a standalone operation or program, but as the behavior of an Operation that is an owned operation of an UML Class.

2.1. **SequenceNode and StructuredActivityNode.** We choose to represent the main block of an activity and all the other blocks with SequenceNodes, and we follow as much as possible the structured programming model. If the *push* model (as described below) is used to represent the statements, then each group of actions corresponding to one statement from a programming language will be grouped inside a StructuredActivityNode. This is done in order to maintain a manageable model, because the number of actions will grow very fast.

The chosen structure of the Activity is like this: the Activity has one InitialNode that marks the beginning of the execution, one SequenceNode as the main node of the activity (the body of the operation), and an ActivityFinalNode that marks the finalization of the execution [3, 5]. One ControlFlow edge will go from the InitialNode to the main SequenceNode, and one ControlFlow edge will go from the SequenceNode to the ActivityFinalNode. The SequenceNode is a structured node, so it may contain other actions. Also, it will execute the actions in order, without a need for explicit ControlFlow edges.

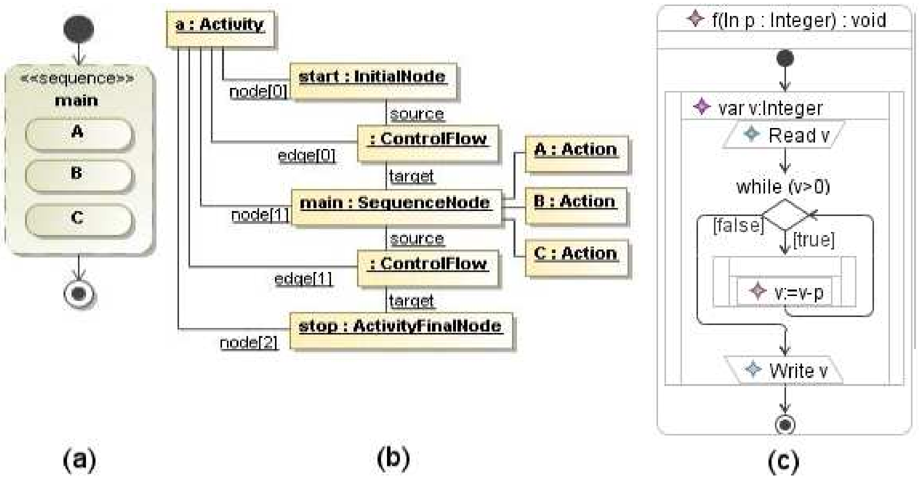


Figure 1. General Structure of an Activity

Figure 1 shows the general structure of an activity. Part (a) shows a possible representation inside an UML tool, part (b) shows the model structure of an activity, and part (c) presents the proposed representation of a simple sample activity. The tools should automatically arrange the elements in the diagram, using a top-to-bottom layout for the statements, optionally showing the implicit control flow with arrows.

We propose to represent the blocks of statements with rectangles, with a double edge on the left and right sides, as shown in Figures 1, 2 and 5.

2.2. **Variables.** The SequenceNode has a set of Variables, that may be used for computations inside the node. The Action Semantics package provides convenient actions to access the values of the variables: AddVariableValueAction (to set a value to the variable), ReadVariableValueAction (to read the value from a variable), and others. The proposed representation for the used variables is shown in Figures 1, 2 and 5. Each block will present its set of variables at the top, in a distinct compartment.

2.3. **Parameters.** The Operation owns a set of Parameters, that describe the inputs and outputs of the operation. We choose that the Activity that represents the behavior of an Operation will always have a similar set of owned Parameters as the Operation. One operation may be invoked from the behavior of another operation by using CallOperationAction [2]. The tools may automate keeping the Parameters of the Operation in sync with the Parameters of the Activity.

There are no actions in UML to access the values of the parameters. Instead, the standards ask for ActivityParameterNodes [1, 4] to be used to provide the parameter input values when the activity starts and to output values to the parameters when the activity ends. One ActivityParameterNode will be created for each *in* and *inout* parameter, only with outgoing edges, and one ActivityParameterNode will be created for each *inout*, *out* and *return* parameter, only with incoming edges.

The input parameter nodes will receive control when the activity starts, at the same time as the InitialNode, and they will provide their parameter values to the outgoing edges. Because the parameter data object may flow over only one outgoing edge (the least resistant one), the usual approach would be to use an intermediate ForkNode [3] to copy the value to all the InputPins of the actions that require it.

The output parameter nodes will copy the values that reached them to the parameters when the activity ends, at the same time when the ActivityFinalNode is executed. The values reaching the parameter nodes will overwrite each other, so, at the end, only the last value that reaches the parameter node will be set to the parameter. Because an action cannot start executing unless all incoming edges provide a token, the usual approach is not to set the edges from the activity actions to go directly in the parameter node, but to merge them before they reach the node, using a MergeNode [3].

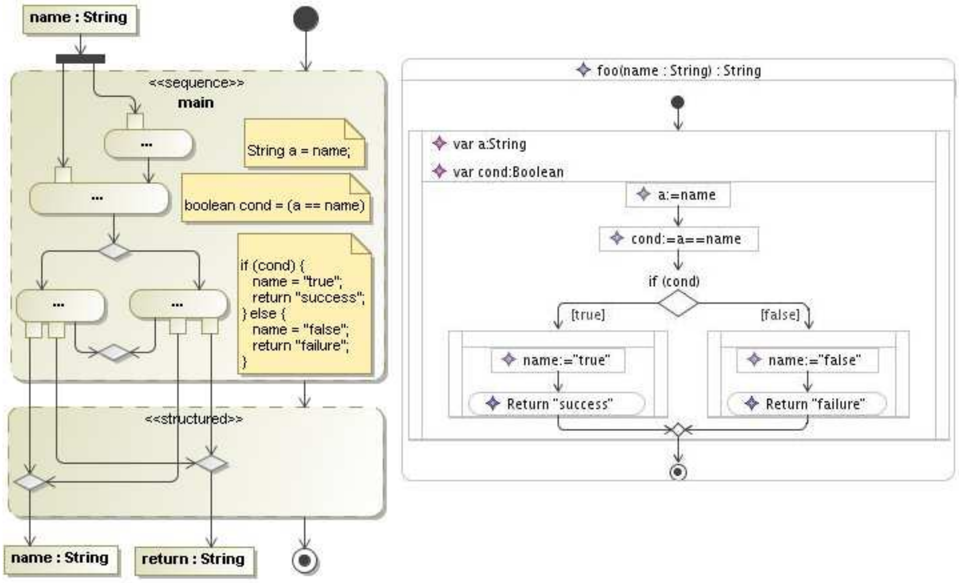


FIGURE 2. Activity with *inout* and *return* Parameters

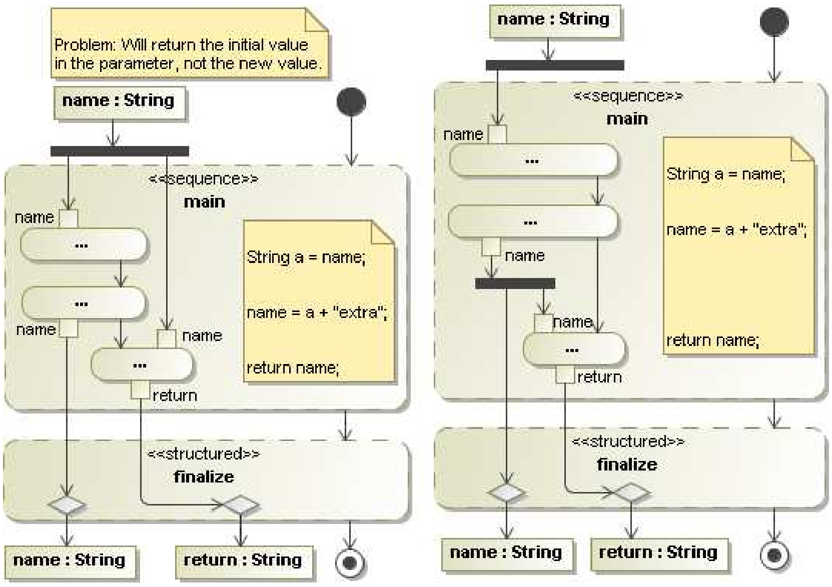


FIGURE 3. Problem (left) and Fix (right) for an Activity with an *inout* Parameter

Figure 2, on the left side, exemplifies the usual approach of working with parameters, for an *inout* and a *return* parameter. The figure uses the UML notations. The "..." actions represent an action or a group of actions that provide the functionality mentioned in the notes placed on the right side.

Using this approach of accessing the parameter values from the actions inside the activity has some problems:

- the model and diagram get very complicated when the functionality is bigger, or when there are many parameters, or if the parameters are accessed many times. The diagram may be fixed if the tools would not show the edges from the parameter nodes.
- the values that are intermediately set to an *inout* parameter during the execution cannot be read, if this scheme is used, as the subsequent actions using the parameter value will receive the initial parameter value from the input parameter node. This can be fixed by passing the intermediate values to the subsequent actions that use the parameter, but this will lead to complicated structures. This issue is presented in Figure 3.
- the *out* parameters cannot be built incrementally, as the stored values cannot be accessed. This can be solved with schemes similar to the one mentioned for *inout* parameters.

To solve these problems, we propose using an alternative approach, presented in Figure 4. For each parameter, except the *return* parameter, there should be a similar Variable (with the same name and type) at the Activity level, and the actions that want to access the parameters will access the corresponding variable instead.

An initialization StructuredActivityNode is introduced between the InitialNode and the main sequence node, having initialization actions:

- for each of the variables corresponding to the *in* and *inout* parameters there will be an AddVariableValueAction that will set the value received directly from the corresponding input ActivityParameterNode to the variable
- for each of the variables corresponding to the *out* parameters there will be an AddVariableValueAction that will set LiteralNull value to the variable.

The actions from the main sequence node that need to access the parameters will simply connect themselves to AddVariableValueActions, ReadVariableValueActions and ClearVariableValueActions configured with the proper variables.

A finalize StructuredActivityNode is introduced between the main sequence node and the ActivityFinalNode, having actions that will read the

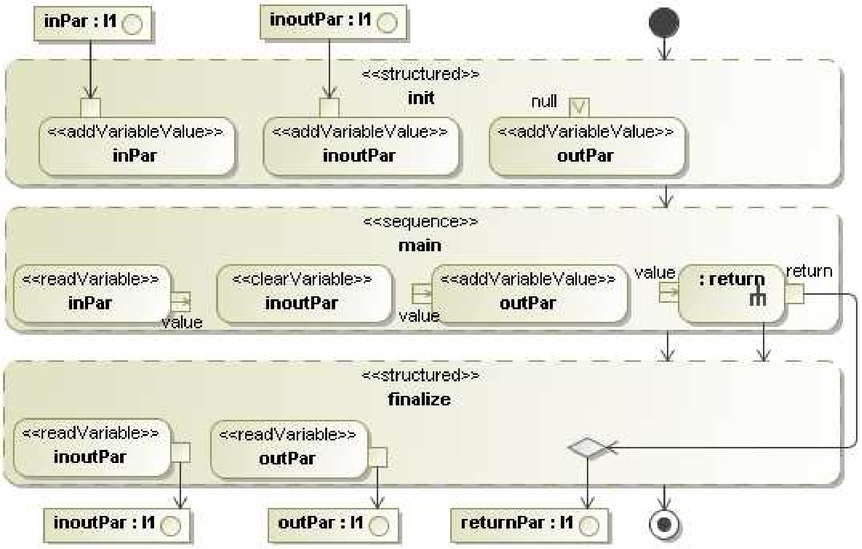


FIGURE 4. Proposed Structure of an Activity

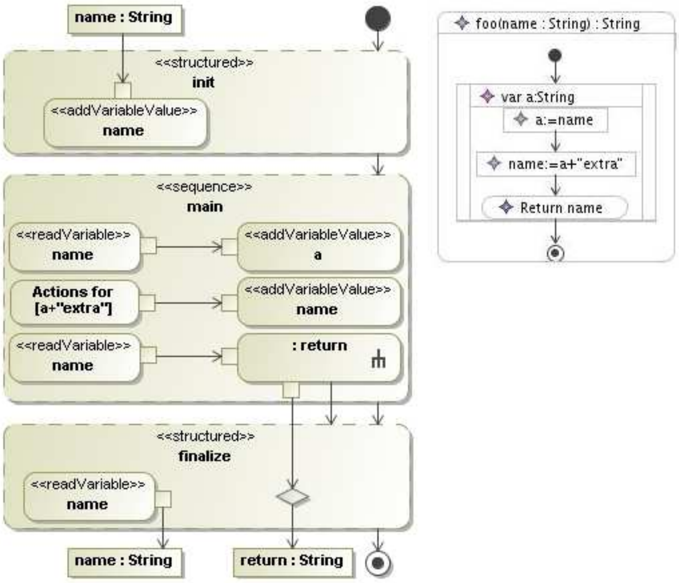


FIGURE 5. Solution and Representation for an Activity with *inout* Parameters

variable values from the *inout* and *out* variables and send them directly to the corresponding output ActivityParameterNodes.

The *return* parameter is handled using the usual approach, but this is described in a subsection below (Return / Output Statement).

In Figure 5 we present the example above with the *inout* parameter problem, solved with this approach (the structured nodes marking the statements are omitted, for brevity). The creation of the variables that correspond to the parameters, along with the *init* and *finalize* nodes containing the variables initialization / output actions, should be automated by the tools. The right side of the figure shows our proposed representation for the activity.

The parameters are presented graphically as part of the activity signature. A distinct compartment containing all the parameters may also be present at the activity level, similar to the compartment for the block variables.

2.4. **Model for Statements and Expressions.** The actions that form each statement may be composed using either the default *push* style model (data tokens will be pushed using ObjectFlow edges from OutputPins to InputPins), or, by using the *pull* style model (data tokens will be pulled by ActionInputPins from Actions with exactly one OutputPin) [5]. The expressions needed in conditions, for instance, are constructed in the same manner. The difference between a statement and an expression is that an expression provides an output value, which is used by a statement (for instance, the *test* node of the LoopNode is an expression that provides a boolean value).

In the *push* style model, the actions are all contained in the same node. The control will arrive at the actions with no input edges, and the data will be pushed through the actions, to the root action. This is not a very intuitive flow for the developers used to structured programming, but UML provides graphical notations and the UML tools have graphical support for it.

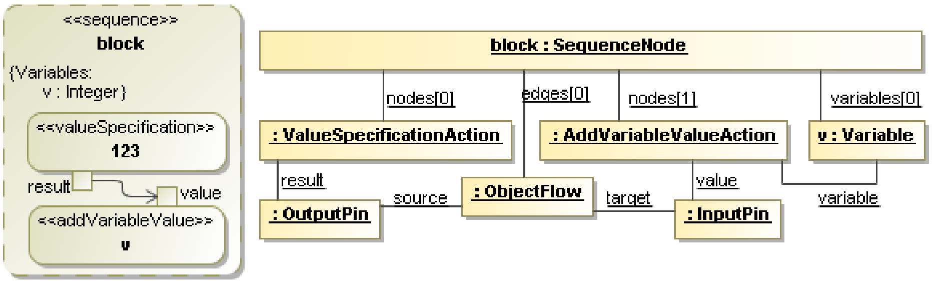


Figure 6. Assignment Statement (v:=123) With *push* Style Model

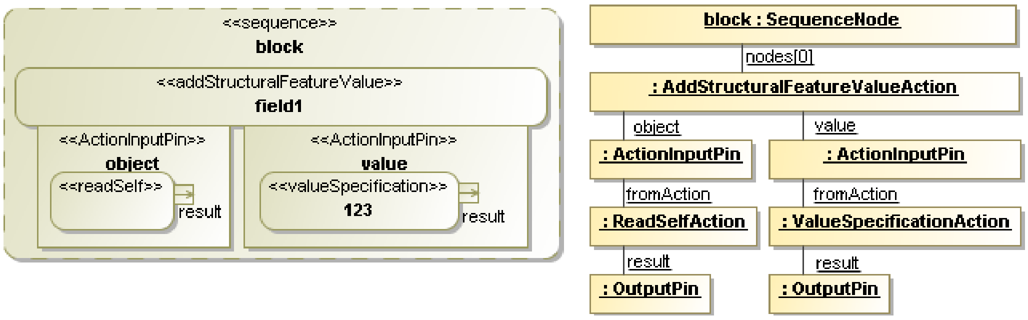


Figure 7. Assignment Statement (self.field1:=123) With *pull* Style Model
(Graphical representation on the left is not UML compliant!)

In the *pull* style model, the root action contains the action input pins, which, in turn, contain the *from* actions, and so on. The control will arrive at the root action, which will begin its execution by trying to get the data tokens from the ActionInputPins, which, in turn, will pull the data tokens from the contained actions, by executing them. The control will arrive in this way at the leaf actions. After the actions are executed and the data tokens are placed in the OutputPins, these data tokens are used as the values for the ActionInputPins. The problem with the *pull* style is that UML provides no graphical notations for the ActionInputPins, as these are meant to be used in textual representations. And this means that most of the UML tools do not have graphical support for the ActionInputPins.

UML provides a special kind of ActionInputPin, called ValuePin [4], that is a shorthand for an ActionInputPin providing the value from a ValueSpecificationAction. The ValuePin provides the value directly from a ValueSpecification. The UML tools might have graphical support for the ValuePin, though UML doesn't propose a graphical notation. However, using ValuePin only, in conjunction with InputPins, is insufficient for more complex statements.

The *pull* style model is chosen, as it fits better to our purpose, and it produces fewer objects, grouped in a well nested structure. However, in order to be able to exchange the models between UML tools, a conversion tool between the two styles is needed, so that a *pull* style model may be viewed and edited inside a UML compliant tool, as a *push* style model.

2.5. Assignment / Input Statement.

- The AssignmentStatement from the PAL model is represented with an action structure that has the root an AddVariableValueAction (if the statement assigns a value to a Variable), or an AddStructuralFeatureValueAction (if the statement assigns a value to a Property of a Classifier). The isReplaceAll boolean property of the action will be set to true. The proposed representation for this statement is shown in Figures 1, 2 and 5 (a simple rectangle containing the textual representation).
- The InputStatement from PAL is represented with the same actions, with the difference that the input value is obtained from a CallBehaviorAction using a FunctionBehavior called *read*, with one return parameter. The proposed representation for this statement is shown in Figure 1.

2.6. Return / Output Statement.

- The *return* parameter is handled using the usual approach (in a non-structured fashion), because the *return* parameter is set only once in an execution path, and after it is set, the execution of the activity has to end. A *return* action sends its result value to the MergeNode found in the $finalize$ StructuredActivityNode, which forwards it to the *return* ActivityParameterNode. Also, the *return* action gives the control to the $finalize$ node, so that the values from the variables are copied to the corresponding parameter nodes and forcing the execution of the activity to end. A CallBehaviorAction is used as the root action of the *return* statement, which means a special *return* FunctionBehavior needs to exist. This behavior should have one *in* parameter (the value to be returned) and one *return* parameter (the same value, that is returned). The action needs one output pin, in order to forward the value to be returned to the *return* ActivityParameterNode. The proposed representation for this statement is shown in Figures 2 and 5.
- The OutputStatement from PAL is represented in a similar fashion, by using a CallBehaviorAction as the root action of the statement. The used FunctionBehavior is called *write* and it has only an *in* parameter. The proposed representation for this statement is shown in Figure 1.

2.7. Branch Statement. The BranchStatement from PAL is represented with a ConditionalNode, with one Clause object if only *then* branch is present, or with two Clause objects if *else* branch is also present. The clauses will be properly ordered by using their successor / predecessor properties. The ConditionalNode will contain all the *test* and *body* executable nodes, and the clauses will properly reference them as *test* or *body* nodes. The decider pin for a *test* clause will always be the output pin of its *test* node. The *else* clause will always have a *true* clause test, meaning that the test node will consist of one ValueSpecificationAction for the *true* LiteralBoolean.

The *body* node is a block of statements and is represented with a single SequenceNode, which will contain the actions for the statements.

The ConditionalNode is not *assured*, meaning that it is possible that no test will succeed (this is needed when the *else* clause is missing). And it is *determinate*, meaning that at most one test will succeed (this is needed when *else* clause is present, so that, if the test of *then* clause passes, the body of *else* clause will not be executed, as the test of *else* clause will always succeed).

An example for the proposed graphical representation is given in Figure 2, on the right side. If *else* branch is missing, there will be a control edge shown, with no statements, going to the merge node at the bottom.

2.8. **While / Do While / For Statement.** WhileStatement and ForStatement from PAL are represented with a tested first LoopNode. DoWhileStatement (a variant of RepeatStatement) is represented with a tested last LoopNode.

The LoopNode is a StructuredActivityNode, so it may have variables, which may be used as iterators for the ForStatement, as opposed to using the built-in system of loop input/output pins, which is hard to use. The LoopNode will contain all the actions for the *setupPart*, *test* and *bodyPart*, which will simply reference the used actions. The iterator may be initialized in the *setupPart* actions. The *test* actions will have to output a boolean value. The decider pin for the *test* will always be the output pin of its *test* actions. The *bodyPart* needs to contain both the actual body actions (inside a SequenceNode) and, if needed, the actions that update the iterator variables.

For the While and Do While statements, the iterator parts are omitted, and only the *test* and *bodyPart* (without the actions that update the iterator variables) will be present.

The loop node has a set of *setupPart* nodes, each one being represented in the model by actions corresponding to a single statement. ControlFlow edges will be set between the *setupPart* nodes, so that the statements are executed in order. The *bodyPart* node includes the main block of statements, which is represented with a single SequenceNode. This node is the first node (has no incoming ControlFlow edges) and will contain the actions for the statements. The statements that update the iterator variables are kept in the *bodyPart* node, also. A ControlFlow edge will go form the main block node to the first iterator update statement, and the rest of the statements are ordered using ControlFlow edges, similar to the *setupPart* nodes.

The graphical representations are similar to those provided in [14], as they help the user understand the flow of the algorithm [15]. A sample for WhileStatement is provided in Figure 1 (c). The tools might support different layouts for the loop nodes, allowing the users to choose the preferred one. The layout used in [14] for ForStatement is chosen, as it does a good job in visually separating the four parts of the statement, while keeping the occupied space to a minimum and providing an intuitive flow.

2.9. Extra Object related actions.

- Reading $self$ (or *this*) instance will be done using ReadSelfAction. This instance will need to be provided whenever the invoked operation or the accessed property is not static and no other instance is explicitly specified by the user.
- Creating a new object instance will be done using CreateObjectAction. This action will not invoke any operation, or behavior, so the created instance could be uninitialized. To obtain the *constructor* behavior found in programming languages, the tools could also execute, if needed, an operation that has the same name as the Classifier and one return parameter of the same Classifier type. The CallOperationAction becomes the root, obtaining its *target* input value from the CreateObjectAction. The CallOperationAction will provide the initialized object to the action that needs the instance, not the CreateObjectAction.

2.10. **Primitive Functions.** Similar to the other FunctionBehaviors mentioned before, a FunctionBehavior needs to be created for each primitive operation (==, +, -, ...) between Integer, Boolean and String typed operands, to be used in expressions. The $primitive functions$ are limited, at this point, to having only operands of the data types defined in UML. All these $primitive functions$ should be packaged in a separate model resource, so that they may be easily reused in different UML tools and different projects.

3. Conclusions and Future Work

Using SequenceNode (sequence), ConditionalNode (decision) and LoopNode (loop) from UML's CompleteStructuredActivities package, the chosen subset of actions is computationally complete.

The new level at which the user creates the executable models is raised from actions to statements and expressions, increasing user efficiency. The tools should take care of a lot of redundant steps while creating the model, as well as properly arranging the diagram, allowing the user to focus on the actual algorithm.

The Action Semantics subset was chosen in such a way that the resulting models are as simple and clear as possible, while preserving the abstract syntax and the execution semantics of the UML elements. This has great benefits, as the resulting models are small and well structured, which makes it easier for an user to analyze them, if needed. It is also not that hard to create conformant models for small operations using existing UML tools.

There is no UML profile defined for the Action Semantics subset chosen in this article, which means that the executable models can be built without having to apply stereotypes. Instead, the article provides exact operational semantics for the selected elements, so that there is an exact interpretation of the model.

Formal OCL [8] constraints need to be defined, so that the UML models can be statically analyzed for conformance with the proposed action language, before being executed. In order to be conformant with this action language, the models must not contain other UML elements, except those proposed here, and they must also comply with the extra operational semantics defined in this article.

This article provides an exhaustive description for the core of an action language using UML Action Semantics. There are many elements remaining to be considered in the future: preconditions, postconditions, ...; re-analyze the support for arrays; switch statement; in-line if statement (with output value): $a > b\ ?\ a : b$; other non-structured statements: break, continue; exception handling; threads; synchronized blocks; operations for associations; events.

FunctionBehaviors for common utility operations may also be defined in the future, and packaged together with the primitive functions. Also, more data types could be defined, as the existing ones are far from being enough.

ACKNOWLEDGMENTS

This work was supported by the grant ID_546, sponsored by NURC - Romanian National University Research Council (CNCSIS).

References

[1] Conrad Bock. UML 2 activity and action models. *Journal of Object Technology*, 2(4):43–53, 2003.

[2] Conrad Bock. UML 2 activity and action models, part 2: Actions. *Journal of Object Technology*, 2(5):41–56, 2003.

[3] Conrad Bock. UML 2 activity and action models, part 3: Control nodes. *Journal of Object Technology*, 2(6):7–23, 2004.

[4] Conrad Bock. UML 2 activity and action models, part 4: Object nodes. *Journal of Object Technology*, 3(1):27–41, 2004.

[5] Conrad Bock. UML 2 activity and action models, part 6: Structured activities. *Journal of Object Technology*, 4(4):43–66, 2005.

[6] I.-G. Czibula, C.-L. Lazăr, I. Lazăr, S. Motogna, and B. Pârv. Comdevalco development tools for procedural paradigm. *Studia Univ. Babeş-Bolyai*, III, 2008.

[7] Object Management Group. *Semantics of a Foundational Subset for Executable UML Models RFP*. http://www.omg.org/docs/ad/05-04-02.pdf, 2005.

[8] Object Management Group. *Object Constraint Language Specification, version 2.0*. http://www.omg.org/docs/formal/06-05-01.pdf, 2006.

[9] Object Management Group. *UML 2.1.2 Superstructure Specification*. http://www.omg.org/docs/formal/07-11-02.pdf, 2007.

[10] I. Lazăr, B. Pârv, S. Motogna, I.G. Czibula, and C.-L. Lazăr. An agile MDA approach for executable UML structured activities. *Studia Univ. Babeş-Bolyai*, LII(2):101–114, 2008.

[11] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.

[12] Bazil Pârv. Comdevalco - a framework for software component definition, validation, and composition. *Studia Univ. Babeş-Bolyai*, LII(2):59–68, 2007.

[13] Tim Schattkowsky and Alexander Förster. On the pitfalls of UML 2 activity modeling. *International Workshop on Modeling in Software Engineering*, 2007.

[14] Tia Watts. *A Structured Flow Chart Editor*. http://watts.cs.sonoma.edu/SFC/.

[15] Tia Watts. The SFC editor a graphical tool for algorithm development. *Journal of Computing Sciences in Colleges*, 4(1):73–85, 2004.

Department of Computer Science, Faculty of Mathematics and Computer Science, Babeş-Bolyai University, 1, M. Kogălniceanu, Cluj-Napoca 400084, Romania

E-mail address: ilazar@cs.ubbcluj.ro