Qualitative Reasoning for Software Systems with Graph Homomorphisms

Constantinos Karapoulios and Spyros Xanthakis
Technological Institute of Larissa
Computer Science and Communications Department
Larissa Greece
karapoulios@teilar.gr

Abstract

In this paper we present an ontology for the analysis and the envisioning of software based systems. We first try to identify and solve the main difficulties of using qualitative reasoning in software engineering. Follow some results of a qualitative abstraction tool and some simple application examples. We then introduce the concept of a system qualitative graph, and the role of graph homomorphisms for modeling a software, and more generally a system, as a continuous phase transition map operating on an abstract data space. We finish with the presentation of the underlying mathematical framework and some properties of graph homomorphism invariants and their use in qualitative reasoning.

1 Introduction

As stated in [Bredeweg and Struss, 2004] Qualitative Reasoning (QR) research has focused on physics and engineering domains but there are many other areas of research and application that can benefit from using QR. According to Djikstra, programming is one of the most complex human engineering activities. The software engineering community has developed since many years, several (formal and informal) methods, tools and concepts to deal with the increasing complexity and size of software based systems, and recently with hybrid systems (software and hardware). Software reliability, maintainability and portability, constitute the main challenges for this engineering field. From the very beginning, software engineering has benefited from Artificial Intelligence (AI) contributions. And this for a very simple reason: the people who developed AI applications was in the same time, a software programmer himself, so he experienced the difficulty of designing, developing and testing a software program. AI concepts and ideas have therefore been successfully applied in this field: automatic programming from examples, conceptual models for software design, automatic program understanding of programmer's intent, case based reasoning for software maintenance and reusability, etc [Rich 1984], [Althof 2001]. With some exceptions in model based reasoning [Missier *et al*, 1993] and [Mayer and Stumptner, 2003], [Trave-Massuyes *et al*, 1997] QR has not been widely applied to software engineering.

The difficulty of applying QR concepts to software engineering is due to the fact that algorithmic behaviour cannot be described as a physical system governed by a simple set of differential equations and some system parameters. Data types are heterogeneous and are not always numerical: strings, records, lattices, vectors, trees, etc. A lot of work has been made in the software engineering field in order to predict and verify software behaviour by analysing statically (in a very detailed manner), the internal structure (i.e. the software source code) of the software system [F. Nielson *et al*, 1998], [Cousot 2005]. Those models are essentially based on formal methods and, like any other software testing method, have their own advantages and limitations due to the inherent complexity of the software programming process. However, static analysis approaches cannot be considered as qualitative since they are too analytic (even if a certain level of data abstraction is operated) and do not propose an ontology that envisions software behaviour as a whole. A software (or more generally a system) ontology must be qualitative and, in our understanding, must respect the following ontology specifications:

- It must propose a right level of *behavioural abstraction* (for designing and debugging) applicable to a wide range of software applications,
- It must be able to express *data type heterogeneity* and software compositionality (outputs of a software module can be used by another module),
- This ontology must be able to *envision software behaviour* when inputs *change*,
- It must contain the *concept of continuity* (even for non ordinal inputs) that is pervasive to any QR reasoning domain.

The paper is organised as follows. We shall expose in a first paragraph, a motivating example of a simple piece of software source code and its corresponding software qualitative graph. This graph summarises the global behaviour of the software system in response of its inputs. The construction of such graphs is completely automated by a tool. Qualitative graphs must respect some constraints that are

independent of the internal structure of the system they envision. Those constraints are uniquely and strictly related to the metric properties of the input space and not necessarily to its dimension. In other terms, qualitative graphs are homomorphic to the equivalence classes of the input space. This simple observation will constitute the grounds of a qualitative ontology based on graph homomorphisms (for oriented and not oriented graphs) that provide an elegant formal (and visual) framework for a qualitative envisioning. Some basic properties of graph homomorphisms and their connection with our qualitative framework will be given in the last section.

2 Motivation

Let's take a very simple piece of software source code written in the C programming language. For illustrative purposes the source code is given here, *Figure 1a*, but we must stress the fact that we do not need to know the internal structure for building the qualitative graph. All what we need to know is the inputs and their domain. In our case we have two integer inputs ***a*** and ***b***, which vary, say, from -100 to +100. We suppose in the same time that our software, when compiled and executed, produces an observable result (given by the **`return`** statement).

An automatic abstraction tool, developed by our team [Guiraud 2001], determines heuristically input values that respect input domains and are situated at the frontiers of the state space regions (i.e. the set of inputs that yield the same output value). This is analogous to the way qualitative phase portraits are defined in qualitative simulation; however, in our case, only the separatrices are relevant and not the trajectories (the search space dimension is lower than the map dimension). After several executions the following two dimension map with four distinct regions is built (*Figure 1b*). When two points belonging to different regions are found, the tool searches heuristically the execution path that links them. The variable *a* increases horizontally, and variable *b* vertically. The point with coordinates, say, **(20, 80)** belongs to the region numbered **3**, since the execution with inputs **a = 20** and **b = 80** produce the integer **3** as a return result. One can observe that the four regions are connected and separated by linear equations (automatically detected). This is due to the fact that the conditions appearing in the source code are linear functions of the inputs. It is often the case to have connected and even convex regions when we handle numeric parameters in software programming. This numerical abstraction mechanism adopted by the tool, is similar to the way qualitative and aggregative models are built and learnt [AI Magazine 2004] from quantitative data. Let's now replace each region by a graph vertex. A vertex ***x*** will be connected with a vertex ***y*** with an arc labelled *a* if there is a point belonging to the region represented by ***x*** where an "infinitesimal increase" (in our case all input variables are integers so the minimal change is 1) of the input variable *a* may lead the program to reach the region ***y***. We obtain a qualitative graph illustrated in *Figure 1c* wich visualizes the nominal behaviour. Since our variables are bounded we could draw an additional vertex (representing an *infinity*, an *error* or an *out of specification*s state).

```
int prem,sec;
if (a >= b) prem = 1;
else prem = 2;
if (a >= 48) sec = 1;
else sec = 2;
if ((prem == 1)&&(sec == 1)) return 0;
if ((prem == 1)&&(sec == 2)) return 1;
if ((prem == 2)&&(sec == 1)) return 2;
if ((prem == 2)&&(sec == 2)) return 3;
```

Figure 1a

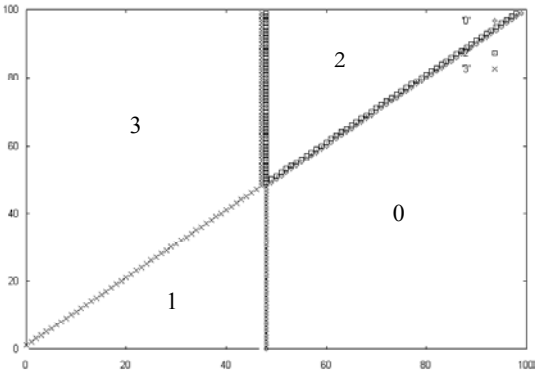


Figure 1b

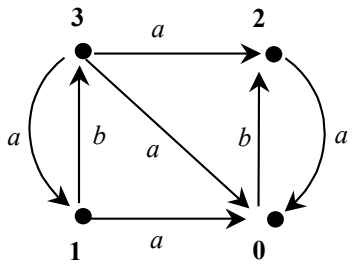


Figure 1c

This graph contains the same relevant information than the map but in a more compact form and does not depend on the map dimension. How this graph can be read? We can see for instance that, when we are in region **1** we cannot join directly the region **2**: we must first visit region **3** by increasing both inputs.

Sometimes we do not represent input labels on the arcs: in a non-oriented qualitative graph only the neighbourhood information is represented. All our graphs are reflexive (but we do not visualise loops on the vertices) since an infinitesimal change permits in most cases to stay in the same region. Regions with one isolated point do not admit loops and so constitute an exception but we don't wish to enter to those considerations in this presentation.

Visualising by means of a graph the closeness of the different software functional areas provides a sort of *software state space*, which permits a global understanding of the

software behaviour (*software phase transitions*). In some real time critical applications it is also important to know how the implemented software will react to some continuous modifications of its environmental inputs. But, expressing a sort of topological representation of input variations is not only relevant for applications where inputs vary continuously. Functional frontiers and transitions are of paramount importance in software testing. One of the most recurrent sources of programming errors [Zeil *et al*, 1992] resides in the fact that the programmer has badly programmed or misunderstood limit behaviour. In our example, a common programming error would consist for instance to write the first condition **(a<=b)**, instead of **(a>=b)** or to write a logical **or** instead of a logical **and** in a conditional statement. This sort of defects causes the deformation and/or the shifting of the surfaces separating the functional regions. Limit testing [Xanthakis *et al*, 2000] consists in stressing the software with input values that are close to or on the separating surfaces.

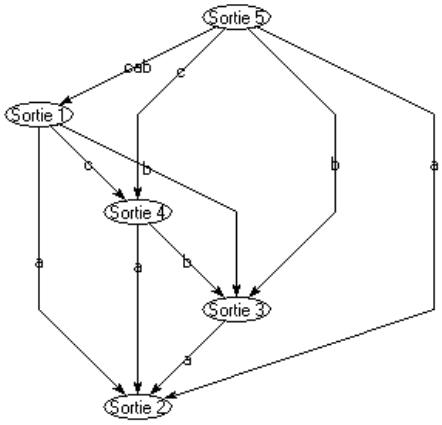
In our qualitative terminology, limit testing means that we shall try to increase or decrease input data in order to visit all the vertices of our qualitative graph. *Figure 2* illustrates another automatic analysis of a simple telecommunication protocol controller (with three control inputs). This qualitative analysis can be derived for any arbitrary program with ordinal parameters. The user has only to specify the name of the input parameters, their type, and their variation domain. Qualitative analysis is then automatically performed by heuristically changing the different input values. In that manner the analysis is completely independent on the internal software structure. This systemic approach differs from conventional static analysis approaches, which are based on slices and variable dependencies. It is also independent on the way the software has been specified, designed, implemented (using UML design, object or aspect programming languages, etc.) and can be applied at any level of the software development life cycle, as soon as we have an executable behaviour of our system.

3 A qualitative ontology based on graph homomorphisms

The previous qualitative graph inputs are organized in a multidimensional grid with a natural distance. However for more general applications, software types are not always ordinal and, due the complexity of the input data, the completeness of the envisionment cannot be ensured (as is the case with differential equations). Those two observations mean that we need to work on *equivalence classes* where a sort of *closeness* has been defined. Actually closeness relation can be defined in other data types than scalars: many software types a poset or lattice structure or a tree structure. Here too a closeness relation can be defined. In other cases, before testing, software engineers partition the input domain into separate classes (note that this a typical qualitative aggregation method), choose a representative test vector in each class, and execute the software system. Here too, we can say that some classes are close when they share some common attributes. Other times, software inputs are states of FSM (or products of FSM). In that case also the closeness between states or their products can be naturally be defined.

All those observations allow one to consider the input domain as a graph, the *input graph*, with its natural closeness relation. We conclude that is meaningful to study what happens when we *smoothly* change an input even if this input are complex data types. Of course, the structure of the equivalence classes as well as the closeness relation are domain (specification) dependent.

In order to illustrate our purposes let's now suppose that, before executing our software, we have found five kinds of input values. In *Figure 3*, the input graph G is transformed in an output qualitative graph H by means of an algorithm ***f.*** The edge **(1, 2)** means that we can "smoothly" change inputs to jump from class **1** to class **2**. For instance, we could decide that the class **2**, groups all input values that are negative but not both zero, class **4** could group values which are positive but not both zero, class **5** could group the point (0, 0), etc.



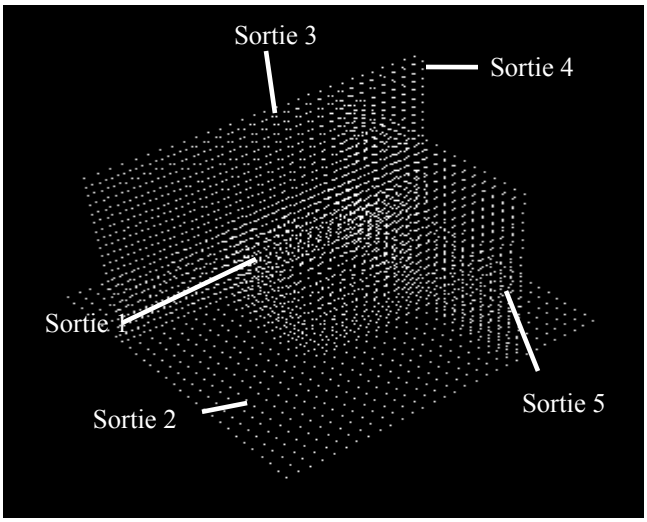


Figure 2

This is a sort aggregation of the bidimensional plane. Of course, G could represent a aggregation of any input space of any dimension. The edge *(x,y)* means that in an execution of ***f*** (taking an input from input class **1**) we get a result belonging to the class **x**.

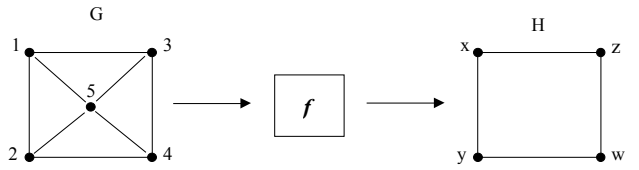


Figure 3

After what, we change "continuously" the input from the class **1** to the class **2** and, in a second execution, we obtain an output belonging to the class **y**. After several executions (or physical observations) we build the output graph H. All the executions are independent. The software system here has no memory. The software map ***f,*** transforms an input graph G to a qualitative one. We must stress the fact that H represents empirically the dynamic behaviour as it is observed externally by another program (as our qualitative tool) or a human user.

H contains the equivalence classes of the input graph G. Two vertices **x** and **y** of G are equivalent when *f(x)= f(y).* This natural equivalence relation means that the output qualitative graph H is isomorphic to the quotient graph ***G/f*** which is **homomorphic** to the input graph G by the homomorphism naturally induced by the equivalence relation. In other words, the qualitative H is built in such a manner, that becomes homomorphic to G and the software *f* is a graph homomorphism. Graph homomorphisms allow one to endow with the concept of continuity an inherently discontinuous field like software programming.

The output qualitative graph H envisions the global dynamic behaviour of the algorithm structure and, since it is homomorphic to the input domain, integrates the topological input constraints independently on the way the algorithm has been implemented. More formally, the qualitative graph must preserve **homomorphism invariants** that are present in the input graph: they allow one to use non homomorphism properties: if a graph does not respect an invariant, some vertices, labels or edges are lacking, or are misplaced.

For example, in *Figure 3*, did one observe all the possible changes of behaviour (all the edges of H)? Can one be sure that he will never follow a forbidden transition edge, i.e. (**x**, **w)** or **(y, z)**? Suppose, for instance, that **x** is the *normal* initial state of the system, **w** is the *error* state and **y** and **z** the *warning* states. Can one be sure that before visiting the *error* state he will always visit exactly one warning state? As we shall see in the next paragraph, graph invariants show that H cannot be homomorphic to G since it does not respect an important homomorphism invariant (*maximal hole number*). Homomorphism invariants filter the possible shapes of qualitative graphs in an analogous manner than physical properties (energy conservation, non intersection on phase transitions, etc.) filter spurious behaviour in qualitative simulation.

Suppose for instance that a system admits two integer inputs *a* and *b* and exhibits four possible classes of output behaviour. Are all qualitative graphs with four vertices admissible? Can one observe the qualitative graph of *Figure 4*?

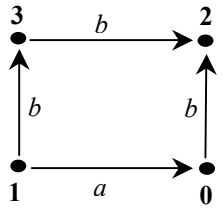


Figure 4

Without delving into further details (homomorphism invariants for oriented software qualitative graphs are discussed in [Karapoulios 1999]) we can say, for that example, that the region **3** must be necessarily connected with a label *a* to the region **2**. In fact, the qualitative graph H is homomorphic to a bi-dimensional oriented grid which has the isotropy property: two vertices which are connected by a path expression of the form $a^n.b^m$ must be also be connected by a path expression $b^m.a^n$. Isotropy is an homomorphism invariant which is not the case for the regions **1** and **2**. So a label *a* is <u>also</u> lacking between the regions **3** and **2** since this constitutes the only way to connect correctly the regions **1** and **2**. We conclude that the tester must design test cases in order to exhibit the specific output transition after an increase (with a sequence of independent executions) of the input *a.*

Graph homomorphisms also provide an elegant and coherent framework for *data abstraction and system composition*. Take for instance an input graph G aggregating equivalent functional classes taken as inputs. Since all computer values are discrete, the real plane is a huge grid where values are connected when there is no an intermediate value between two decimal points (the grid vertices). We conclude that even input graph G must be homomorphic to that original grid. Homomorphism composition can be further used in the case where the output graph H is an <u>input</u> graph of another system, ***say f'***, providing a new qualitative graph H'.

The next paragraph gives a more formal flavour to those observations with some basic properties of homomorphisms of non-oriented graphs.

4 Mathematical framework

We adopt conventional notations for graphs G(X, U) with X the set of vertices and U the set of edges. We note *x~y* the adjacency of the two vertices. Graphs are connected and reflexive but we do not visualize loops. We note $d_G(x, y)$ the natural distance in a connected graph G that is, the length of the shortest path, linking *x* to *y*. We note I_n as the path of length n, C_n are the cycles of length n. Grids noted $G_{m,n,p...}$ are cartesian products of paths.

An ***homomorphism*** [Godsil and Royle, 2001], is a map $h:G \rightarrow H$ preserving adjacency: i.e. *x~y* implies *h(x)~h(y)*.

Our graphs being reflexive, this definition is equivalent to a non expanding map: $d_G(x, y) \geq d_H(h(x),h(y))$.

Homomorphisms will always be onto. When $G = H$ we say that we have an endomorphism. Idempotent endomorphisms are also called **retractions**. So retractions are homomorphisms which leave invariant a subgraph G', called a **retract** of G. Graph homomorphisms, as well as retractions constitute a very active area of research in graph theory [Hahn and Tardiff, 1997] [Hahn and MacGillivray, 2002], [Hell and Nesetril, 2004], [Imrich and Klavzar, 2000], [Brightwell and Winkler, 2000].

*A **contraction** is an onto homomorphism $h:G \rightarrow H$ where the inverse image of every vertex of H is a **connected** subgraph of G.*

We note *G/h* the quotient graph induced by the kernel of *h*. A partition is *elementary* when all the equivalence classes contain only one element, except one class that contains exactly two adjacent vertices. More particularly a contraction is **elementary** when it induces an elementary partition.

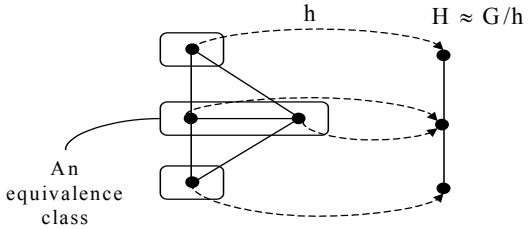


Figure 5

An elementary contraction can be viewed as gluing two adjacent edges following their common edge which disappears; the other adjacent edges follow the contracted vertex. *Figure 5* illustrates an elementary contraction *h* and its kernel *G/h*.

An homomorphism **invariant** is a non negative real valued function ∂ verifying: $\partial(G) \geq \partial(h(G))$ for any homomorphism *h*. The number of vertices, edges as well as the diameter are trivial invariants. It is easy to observe that any contraction is the commutative composition of elementary contractions. That means that if a property is an invariant for any elementary contraction it is also, by induction, a contraction invariant. An immediate property of that observation is that contractions preserve **planarity**.

For a connected subgraph G' of G, we define *discon(G')* as the number of connected components (possibly a single vertex) that we obtain when we remove G'. We call it the *disconnecting capacity* of G'. It is easy to prove that for any contraction *h* we have*: discon(G')≥discon(h(G'))*. This property yields an interesting corollary (that can easily generalized for higher dimensions): any bi-dimensional grid of an odd size *m* (i.e. $G_{m,m}$), with m≥3 cannot be contracted to any path I_{m+1}.

In *Figure 6* we illustrate this: the path I_3 cannot be contracted to the grid $G_{2,2}$. To have an idea of the general demonstration note, in *Figure 6*, that the central vertex *c* has a disconnecting capacity of 1 since its removal does not disconnect the graph. At the same time it is at a maximal distance of 2 from all the other vertices, so it cannot be homomorphically mapped to the two outer vertices of I_3. So, if a contraction exists, its image *h(c)* is necessarily a vertex in the middle of I_3, which disconnects I_3, thus increasing its disconnecting capacity, which is impossible.

The maximal disconnecting capacity of a graph, *mdc*(G), is the maximum *discon*(G') that we can obtain from a subgraph G'. Since *discon*(G') does not increase, mdc(G) is a contraction invariant. In Figure 7, we have *mdc*(G)=2 and *mdc*(H)=3. A cycle contains a **chord** when two not subsequent vertices of the cycle are connected. Chordless cycles that are also retracts are called **holes**. For instance, *in Figure 3*, the cycle [1, 2, 4, 3] is a chordless cycle since opposite vertices (like 1 and 4) are not adjacent, but is not a hole, since there is no possible retraction on this cycle. Let *hole*(G) be the greatest length of a hole in G. For instance, the only holes of grids are the cycles of length 4. So the hole number of any grid, of any dimension, is 4. It can be proved by induction that elementary contractions do not increase the hole number, so *hole*(G) is also a contraction invariant.

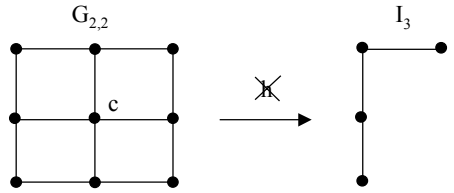


Figure 6

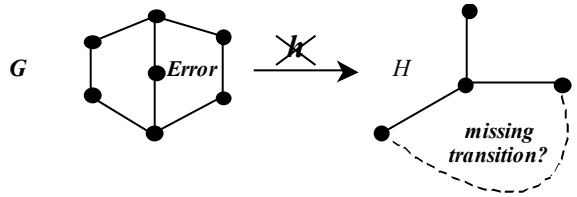


Figure 7

As we said in the previous sections, if we assume the connectivity of functional regions, contraction invariants can be used to constraint qualitative graphs. For instance, in *Figure 6*, the non homomorphism property based on the ***disconnecting capacity*** invariant says that if we observe a software with two integer inputs partitioned in 9 classes of a bidimensional plane (combination of negatives and positive coordinates) it is impossible to observe a completely linear behaviour. The ***mdc***(G) invariant, in *Figure 7*, permits to say that when inputs of a software system follow a cyclic finite state machine with a central error state, then the observed output region transitions cannot have a star-like topology. A transition edge is missing. The ***hole*** number permits to conclude that any system with any number of scalar inputs cannot exhibit a 5-cycle behaviour without a missing transition among the states of the cycle.

Moreover, contraction invariants can also be used to deduce some important properties of the input graph. Let's take an interesting example. Suppose one has a system (like a robot) that he controls with two integer variables forming

a planar input graph. Suppose now that this robot exhibits a good behaviour (this is a vertex of the output qualitative graph H). One wants to be sure that small perturbations on input variables (the graph G) will not suddenly change robot's behaviour. That is, we want to be sure that whatever would be the output behaviour, there is *the possibility* of maintaining the robot in the same configuration while smoothly changing its control variables (a sort of controllability). That simply implies that all the control surfaces of our system must be connected since we wish to visit all the points of a region without jumping into another. In other words there must be a contraction between G and H. Suppose now that, during the testing of our robot, we observe a graph H containing a 5-cycle hole, or, a graph with so many transitions that make it no planar. In both cases, we can conclude that there is no possibility of contraction. Thus, inverse images are not connected and the system is not controllable.

Conclusion

Qualitative reasoning has been applied for the testing of protocol oriented software components and proved to be very beneficial. Software designing, programming, testing and debugging are very complex and error prone human activities. Conventional software engineering methods and tools are very powerful but lack of a global qualitative ontology to help the engineer to understand the system behaviour. An ontology based on graph homomorphisms has been presented. Software is viewed as a sort of continuous map transformation between two abstract data spaces likewise a phase transition system. The qualitative graph envisions the global of the software system and respects some constraints that are independent of its internal structure. Those constraints, called invariants, express topological properties of graph homomorphisms. They can be used to infer the possible shapes of the qualitative graph. An automatic abstraction tool has been presented. Many questions may arise: can we abstract all common input data structures with a proximity relation? Is it possible to express more quantitative information in the qualitative graph labels? How do we handle state machines, time and memory? How this ontology can be extended to physical and/or artificial systems? Do endomorphisms or retractions express some specific classes of software behaviour? Can we classify software applications according to the properties of their endomorphisms (that is, the properties of the generated monoid)? How the composition of homomorphisms can express system integration? Is it possible to express some software errors as the composition of the correct map with an error map that one could study in more details? All those questions are open but we think that the main contribution if our ontology resides in the fact that it proposes a bridge between qualitative reasoning and a very seminal area of applied mathematics.

References

[Althoff 2001] Althoff K.-D. Case-Based Reasoning. In *Handbook on Software Engineering and Knowledge Engineering*. Vol. 1 "Fundamentals", Chang, S. K., Editor, World Scientific. pages 549-588, 2001.

[AI Magazine 2004] *AI Magazine*, Winter 2004. , pages 47 and 107.

[Bredeweg and Struss, 2004] Bert Bredeweg and Peter Struss. Current Topics in Qualitative Reasoning. *AI Magazine*, Winter 2004.

[Brightwell and Winkler, 2000] G. R. Brightwell and P. Winkler, Gibbs measures and dismantlable graphs*, J. Graph Theory* 11 (1987) 71-79.

[Cousot 2005] Patrick Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Sixth International Conference on Verification, Model Checking and Abstract Interpretation* (VMCAI'05), Paris, France, January 2005. LNCS 3385, (c) Springer, Berlin, pages 1-24.

[Guiraud 2003] Virginie Guiraud. Visualisation du comportement dynamique des logiciels numériques. *Rappot de stage, société SOPRA*, 2001-2003.

[Godsil and Royle, 2001] Chris Godsil and Gordon Royle. *Algebraic Graph Theory*. Springer Verlag, No 207, 2001.

[Hahn and Tardif, 1997] Gena Hahn and Claude Tardif in. Graph homomorphisms structure and symmetry, in *Graph Symmetry* (*eds*. Hahn and G. Sabidussi)., Kluwer Academic Publishers, 1997.

[Hahn and MacGillivray, 2002] Gena Hahn and Gary MacGillivray. Graph homomorphisms: computational aspects and infinite graphs. *Research report*, Université de Montreal, June 2002.

[Hell and Nesetril, 2004] Pavoll Hell and Jaroslav Nesetril. Graph and Homomorphisms. *Oxford Lecture Series in Mathematics and its Applications*, Oxford University Press, 2004.

[Imrich and Klavzar, 2000] Wilfried Imrich and Sandi Klavzar, Product Graphs, structure and recognition, *Wiley Interscience Series in Discrete Mathematics*, 2000.

[Mayer and Stumptner, 2003] Wolfgang Mayer and Markus Stumptner. Model-Based Debugging using Multiple Abstract Models. In *Proceedings of the 5th International Workshop on Automated and Algorithmic Debugging*, pages 55-70.

[Missier *et al*., 1994] Antoine Missier, Spyros Xanthakis and Louise Trave-Massuyes. Qualitative Algorithmics using Order of Growth Reasoning. In *Proceedings ECAI 94*, pages 750-754, 1994.

[Nielson *et al*., 1998] Flemming Nielson, Hanne Riis Nielson and Chris Hankin. *Principles of Program Analysis*, Springer, 1998.

[Karapoulios 1999] Constantinos Karapoulios. Raisonnement Qualitatif Appliqué au Test Evolutif des Logiciels.

Thèse de Doctorat, I.R.I.T, Université Paul Sabatier, Toulouse, France, Juillet 1999.

[Rich 1984] Charles Rich. Artificial intelligence and software engineering: the programmer's apprentice project. In *Proceedings of the 1984 annual conference of the ACM*, 1984.

[Trave-Massuyes *et al*., 1997] Louise Travé-Massuyès, Phillipe Dague and Francois Guerrin. *Le raisonnement qualitatif pour les sciences de l'ingénieur* (coll. diagnostic et maintenance), chapter 12. Editions Hermès, France, 1997.

[Xanthakis *et al*., 2000] Spyros Xanthakis, Pascal Régnier and Constantinos Karapoulios. *Le test des logiciels, Etudes et logiciels informatiques*. Editions Hermès, France, 2000.

[Zeil *et al*., 1992] Steven J. Zeil Faten H. Afifi Lee J. White. *Detection of linear errors via domain testing*. ACM Press, New York, NY, USA, 1992.