Smarter Features, Simpler Learning?

Sarah Winkler
University of Verona, Italy
sarahmaria.winkler@univr.it

Georg Moser
University of Innsbruck, Austria
georg.moser@uibk.ac.at

Earlier work on machine learning for automated reasoning mostly relied on simple, syntactic features combined with sophisticated learning techniques. Using ideas adopted in the software verification community, we propose the investigation of more complex, structural features to learn from. These may be exploited to either learn beneficial strategies for tools, or build a portfolio solver that chooses the most suitable tool for a given problem. We present some ideas for features of term rewrite systems and theorem proving problems.

1 Introduction

The idea to exploit machine learning in automated reasoning has been a natural one, given the success of AI methods in areas like Jeopardy[1] and Go.[2] Indeed, machine learning has already been investigated to improve heuristics for different tasks in theorem proving. Most prominently, these include learning parameters of the proof process [9], learning selection of the objects (like clauses) to process next [22, 14], premise selection [15], and learning proof search guidance in a more general sense, such as which branch to follow in a tableau proof [16].

Work done in the area so far typically exploited features reflecting the syntactic structure of the input; we mention some examples as representatives of different approaches. Used characteristics commonly include symbol counts, the number of clauses of a certain type, clause depth, and clause weight. For instance, such properties were considered by Bridge et al. [9] to determine tool parameters both for the initial problem (*static* features) and during the proof process (*dynamic* features). The strategy scheduling learner E-MaLeS by Kaliszyk et al. [18] also relies on features such as clause, literal, and term count, function symbol arity, and problem class (e.g., Horn or unit problems). Another approach used for clause selection in early work by Schulz [22] and Jakubův and Urban [14, 10] is to extract symbol strings (sometimes called *term walks*) from the tree representation of terms, and count occurrences of a predefined set of strings to obtain a feature vector. In a further experiment by Loos et al. [21] the entire problem was fed into a neural network. In the context of premise selection for a given conjecture, terms in the conjecture have often been compared to the terms in the premises, using the well-known notion of term frequency-inverse document frequency, but also by comparing prefixes of terms, see e.g. work by Jakubův and Urban [15].

These approaches tend to use elaborate machine learning models with rather simple features. A contrasting approach following the paradigm of *smart features, simple learning* has been pursued by Demyanova et al. to design a portfolio solver for the software verification competition (SV-COMP) [13]: Based on metrics of programs related to loop types, variable use, and control flow, the authors applied machine learning techniques to decide which solver to use. The resulting portfolio solver would have won the software competition in three consecutive years. Since the applied machine learning model is

[1] WATSON, IBM, `https://researcher.watson.ibm.com/researcher/view_group.php?id=2099`
[2] ALPHAGO, DeepMind, `https://deepmind.com/research/alphago`.

comparatively simple (support vector machines), the success of the method is attributed to the sophisticated metrics. Besides providing a powerful combined solver, the work thus also delivers a taxonomy of problems, along with insights on which techniques and tools work best on them.

We propose to experiment with a similar approach to the areas of term rewriting and theorem proving. Both in the termination[3] and the confluence competition[4] for term rewrite systems a variety of tools compete annually on thousands of benchmarks. Each tool has its own strengths on problems of a certain kind, but it can typically also be run with a vast variety of different strategies that accommodate better to some problems than others. We do not deny the relevance of syntactical characteristics: some tools and analysis techniques are obviously restricted to problems with certain syntactic properties. For instance, some tools are restricted to string rewrite systems, and the Knuth-Bendix order only applies to non-duplicating rewrite systems. However, many tools work on a wide range of problems and offer a variety of proof strategies. This holds for rewrite tools that attempt to establish termination, complexity bounds, and confluence; but also for theorem provers competing in the CADE Annual System Competition (CASC).[5] It is a common experience in this field that with the "right" strategy, a proof can be found within a few seconds or even fractions thereof while other strategies diverge hopelessly; but a suitable strategy is hard to predict. Many tools thus employ *strategy scheduling*, where a variety of strategies is run subsequently on a problem for a short time each, in the hope that one of them might succeed [18].

In this extended abstract we thus propose to investigate meaningful structural features that, possibly together with syntactic characteristics, allow machine learning models to predict which proof strategy and/or tool works best for a given problem. We do not fix a particular learning task, as we think that such structural properties could be of interest for different concrete applications.

In Section 2 we summarize the program metrics exploited in [13]. We propose corresponding metrics for term rewrite systems in Section 3. Subsequently, we consider possible extensions to problems from theorem proving in Section 4 and conclude in Section 5.

2 Program Metrics

The metrics to classify programs proposed in [13] are related to three different aspects: variable roles, loop patterns, and control flow.

For the former, the authors defined 27 variable roles, including e.g., pointers, loop bounds, array indices, counters, and variables of a particular type. An occurrence count vector holding the number of variables of each role, normalized by the total number of variables, is included in the features used for learning.

Second, four loop patterns are defined based on an estimate whether the number of iterations in a program execution can be bounded or not. In particular, the first two patterns correspond to loops with a constant number of iterations, and a certain type of FOR-loops where termination is guaranteed. Again, the number of loops of each kind gets counted, and an occurrence count vector (normalized by the number of loops overall) is included in the feature list.

Finally, control flow metrics include the number of basic blocks and the maximal indegree of a basic block for intraprocedural control flow, as well as the number of (recursive) call expressions and the involved parameters to account for interprocedural control flow.

[3] `http://termination-portal.org/wiki/Termination_Competition`

[4] `http://project-coco.uibk.ac.at`

[5] `http://www.tptp.org/CASC`

3 Metrics for Rewrite Systems

In this section we propose structural features for rewrite systems like the ones collected in the Termination Problems Database TPDB[6] or the collection of confluence problems CoPs.[7] Some basic notions of term rewriting are assumed in the sequel, see e.g. [5].

As mentioned above, there are a number of syntactic features of rewrite systems which naturally restrict applicable tools and strategies. These include the type of rewrite systems (string, conditional, constrained, classes of higher-order systems). On the other hand, one may want to recognize variable duplication (e.g. for the Knuth-Bendix order), linearity (e.g. for confluence criteria), or left-linear right ground systems (whose first-order theory is known to be decidable [12]).

That said, we next discuss structural features which might influence applicability of certain tools and methods, as outlined in Section 2. While programs are generally more structured than rewrite systems, we argue in the next paragraphs that several properties outlined in Section 2 actually remain relevant.

Variable Roles. On the level of term rewrite systems, a program variable corresponds to an *argument position* of a function symbol. We consider a function symbol f of arity n occurring in a TRS R, with argument position $1 \leq i \leq n$. The following properties of the ith argument can be considered:

- it is a *projection argument* if R contains a rule $f(t_1,\dots,t_n) \to t_i$,
- it is *decreasing* in presence of a rule $f(t_1,\dots,C[t_i],\dots t_n) \to f(t_1,\dots,t_i,\dots t_n)$ where C is a non-empty context (or *increasing* if the rule is reversed),
- it is *recursive* if some rule $f(t_1,\dots,t_n) \to f(t_1,\dots,f(u_1,\dots,u_n),\dots t_n)$ features a recursive call in the ith position,
- it is a *pattern matching argument* if R has rules $f(t_1,\dots,c_1,\dots,t_n) \to r_1$ and $f(t_1,\dots,c_2,\dots,t_n) \to r_1$ with different constructors c_1 and c_2 occurring at position i, and
- it is a *duplication argument* if in a rule $f(t_1,\dots,t_n) \to r$ term t_i occurs at least twice in r.

Some of these properties resemble roles of program variables: for instance, a decreasing or increasing argument position can be seen as the equivalent of a counter variable in a TRS, and a projection argument corresponds to a variable that is returned by a function. In the case of integer transition systems, many more of the variable roles from [13] have straightforward equivalents, for instance counters, argument positions involved in branching conditions, and arguments involved in arithmetic operations .

Moreover, for particular types of rewrite systems, further roles become important: In higher-order systems, argument positions can be divided into first- and higher-order ones. In the case of conditional or constrained TRSs, positions having *extra variables* can be recognized (i.e., variables that occur in the right-hand side of a rule but not in the respective left-hand side) .

Recursion Patterns. In particular for termination and complexity analysis of rewrite systems, recursion constitutes the key difficulty. The first two loop patterns considered in [13] are strongly connected to the idea of tiering and safe recursion [23, 6, 19], where the essential idea is that the arguments of a function are separated into normal and safe arguments, and recursion is only allowed in safe arguments. Such a problem classification is also commonly used in runtime complexity analysis of term rewrite systems [4, 3]. More generally, restricted loop structures have been considered in the context of decidable

[6] `http://termination-portal.org/wiki/Termination_Competition#Termination_Problems_Data_Base`
[7] `http://termcomp-devel.uibk.ac.at/cops`

classes of resource or termination analysis, cf. [7, 8, 11]. Again, for the case of integer transition systems the loop patterns of [13] naturally carry over.

Control Flow. Just like call graphs depict calling relationships between functions in imperative programs, a call graph can be considered for term rewrite systems, and e.g. recursive calls and degrees of nodes can be counted to obtain numerical features.

The dependency graph [2] (DG) is another common illustration of control flow in TRSs. This graph is in general not computable, but estimations thereof are often used in termination tools. Natural features of this graph are given by the number of nodes, edges, and strongly connected components (SCCs). Though the control flow is in general less obvious for TRSs than for programs, some of the program metrics mentioned above have quite direct correspondents: The number of recursive calls is reflected by the number of paths in the DG between nodes with the same root symbol. Mutually recursive functions are reflected by edges between nodes and SCCs of different root symbols. The indegree of nodes in the DG could be considered related to the indegree of basic blocks.

We illustrate some of the proposed properties by means of an example.

Example 3.1. The following TRS R models a functional program to enumerate prime numbers:

$$\begin{array}{lrl lrl}
(1) & \mathsf{primes} & \to \mathsf{sieve}(\mathsf{from}(\mathsf{s}(\mathsf{s}(0)))) & (5) & \mathsf{sieve}(0:y) & \to \mathsf{sieve}(y) \\
(2) & \mathsf{from}(n) & \to n:\mathsf{from}(\mathsf{s}(n)) & (6) & \mathsf{sieve}(\mathsf{s}(n):y) & \to \mathsf{s}(n):\mathsf{sieve}(\mathsf{filter}(n,y,n)) \\
(3) & \mathsf{take}(0,y) & \to \mathsf{nil} & (7) & \mathsf{filter}(0,x:y,m) & \to 0:\mathsf{filter}(m,y,m) \\
(4) & \mathsf{take}(\mathsf{s}(n),x:y) & \to x:\mathsf{take}(n,y) & (5) & \mathsf{filter}(\mathsf{s}(n),x:y,m) & \to x:\mathsf{filter}(n,y,m)
\end{array}$$

For instance, the first arguments of take and filter are both decreasing and pattern matching arguments, while the argument of from is increasing. Both can be seen as analogues of counter variables. The last argument of filter is a duplication argument.

The call graph of this program depicts calling relationships between the different functions: an edge from function f to function g indicates that there exists a rule with f as the root symbol of a left-hand side where g occurs on the corresponding right-hand side.

$$\begin{tikzcd}
\mathsf{take} \arrow[loop below] & \mathsf{from} \arrow[loop below] & \mathsf{primes} \arrow[l] \arrow[r] & \mathsf{sieve} \arrow[loop below] \arrow[r] & \mathsf{filter} \arrow[loop below]
\end{tikzcd}$$

From the dependency pairs of R:

$$\begin{array}{lrl lrl}
(1) & \mathsf{primes}^\# & \to \mathsf{sieve}^\#(\mathsf{from}(\mathsf{s}(\mathsf{s}(0)))) & (5) & \mathsf{sieve}^\#(0:y) & \to \mathsf{sieve}^\#(y) \\
(2) & \mathsf{primes}^\# & \to \mathsf{from}^\#(\mathsf{s}(\mathsf{s}(0))) & (6) & \mathsf{sieve}^\#(\mathsf{s}(n):y) & \to \mathsf{sieve}^\#(\mathsf{filter}(n,y,n)) \\
(3) & \mathsf{from}^\#(n) & \to \mathsf{from}^\#(\mathsf{s}(n)) & (7) & \mathsf{sieve}^\#(\mathsf{s}(n):y) & \to \mathsf{filter}^\#(n,y,n) \\
(4) & \mathsf{take}^\#(\mathsf{s}(n),x:y) & \to \mathsf{take}^\#(n,y) & (8) & \mathsf{filter}^\#(0,x:y,m) & \to \mathsf{filter}^\#(m,y,m) \\
 & & & (9) & \mathsf{filter}^\#(\mathsf{s}(n),x:y,m) & \to \mathsf{filter}^\#(n,y,m)
\end{array}$$

the dependency graph can be constructed, where there is an edge from $\ell_1 \to r_1$ to $\ell_2 \to r_2$ if r_1 may rewrite to a term matching ℓ_2. It provides a more fine-grained analysis than the previous graph, just like a basic block flow graph shows more details than a call graph.

$$\begin{tikzcd}
(1) \arrow[r] \arrow[d] \arrow[dr] & (7) \arrow[r] \arrow[rr, bend left] & (8) \arrow[r, bend left] \arrow[loop below] & (9) \arrow[l, bend left] \arrow[loop below] & \\
(6) \arrow[r, bend left] \arrow[ur] \arrow[loop below] & (5) \arrow[l, bend left] \arrow[u] \arrow[loop below] & (2) \arrow[r] & (3) \arrow[loop below] & (4) \arrow[loop below]
\end{tikzcd}$$

Finally, we mention some further features of term rewrite systems which express structural properties that may be relevant. These include orthogonality, the number of critical pairs, or whether it is a constructor system. But also more complex properties such as simple (non-)termination and local (non-)confluence can be easily decided for some cases, and might be indicative for the complexity of an input TRS and hence appropriate strategies. Moreover, also the presence of algebraic substructures such as associative and commutative operators, groups, lattices, or distributivity can be taken into account.

4 Metrics for Theorem Proving Problems

Here we consider theorem proving problems as collected in the TPTP benchmarks [24]. This collection is quite diverse. Given a theorem proving problem, it thus seems natural to consider membership of this problem in different syntactic problem classes, as in the case of rewrite systems. In the case of TPTP problems, one might consider the unit equality class (UEQ), the Bernays-Schönfinkel class (EPR) [20], higher-order classes, or decidable first-order fragments like the monadic, the Ackermann, or the guarded fragment [1], description logics, or problems where symbols can be sorted in an acyclic way [17].

Membership of the problem in classes like UEQ or EPR is already indicated in the metadata of TPTP files, as are in fact many other syntactic properties: the number of (unit) clauses, literals, and terms; the number of occurrences of different logical operators; the number of function symbols, predicates, and (existentially and universally quantified) variables; formula and term depth; as well as information about arithmetic operators and types. In learning suitable strategies for a given problem, it cannot hurt to include these features, since non-influential problem properties can be ignored by the model.

Next, we mention some structural properties which can be included to classify problems. In presence of equality literals, all features described in Section 3 can be considered for theorem proving problems as well. Besides, the presence of substructures like group, list, lattice, or array axioms can be exploited.

For typed theorem proving problems, we can moreover take the number of arguments of boolean, integer, real, and set type in every function symbol and predicate into account. These features would resemble variable types being taken into account as done in [13].

In Section 3 the dependency graph was used to keep track of dependencies between function symbols. A similar approach could be used to analyze the dependencies between literals. For problems in CNF without equality, the following notion of a clause dependency graph could be used for this purpose.

Definition 4.1. *Let S be a set of variable-disjoint clauses without equality. The* clause dependency graph *has as node set all pairs of literals (l, l') such that there is a clause in S containing $\neg l$ and l', and an edge from (l_1, l_1') to (l_2, l_2') if and only if l_1' and l_2 have the same polarity and are unifiable (where we identify literals $\neg\neg p$ and p, for all atoms p.)*

The clause dependency graph captures opportunities for resolution. For instance, if this graph has no cycles then the problem admits only finitely many resolution steps and is hence decidable. We illustrate this idea by means of an example.

Example 4.1. Consider the following two sets of clauses

$$\begin{array}{llll} S_1: & \neg\mathsf{P}(x) \vee \mathsf{Q}(x) & \neg\mathsf{Q}(y) \vee \neg\mathsf{R}(\mathsf{f}(y),y) & \mathsf{R}(u,z) \vee \mathsf{P}(\mathsf{f}(u)) \\ S_2: & \neg\mathsf{P}(x) \vee \mathsf{Q}(x) & \neg\mathsf{Q}(y) \vee \neg\mathsf{R}(\mathsf{f}(y),y) & \mathsf{R}(u,u) \vee \mathsf{P}(\mathsf{f}(u)) \end{array}$$

The clause dependency graphs look as follows, where we use colors to indicate from which clauses the literal pairs originate:

$$\begin{tikzcd} S_1: & (\neg\mathsf{R}(u,z),\mathsf{P}(\mathsf{f}(u))) \arrow[d] & (\mathsf{R}(\mathsf{f}(y),z),\neg\mathsf{Q}(y)) \arrow[dr] & (\neg\mathsf{P}(\mathsf{f}(u)),\mathsf{R}(u,z)) \arrow[l] \\ & (\mathsf{P}(x),\mathsf{Q}(x)) \arrow[r] & (\mathsf{Q}(y),\neg\mathsf{R}(\mathsf{f}(y),y)) \arrow[ul] & (\neg\mathsf{Q}(x),\neg\mathsf{P}(x)) \arrow[u] \end{tikzcd}$$

$$\begin{tikzcd} S_2: & (\neg\mathsf{R}(u,u),\mathsf{P}(\mathsf{f}(u))) \arrow[d] & (\mathsf{R}(\mathsf{f}(y),y),\neg\mathsf{Q}(y)) \arrow[dr] & (\neg\mathsf{P}(\mathsf{f}(u)),\mathsf{R}(u,u)) \\ & (\mathsf{P}(x),\mathsf{Q}(x)) \arrow[r] & (\mathsf{Q}(y),\neg\mathsf{R}(\mathsf{f}(y),y)) & (\neg\mathsf{Q}(x),\neg\mathsf{P}(x)) \arrow[u] \end{tikzcd}$$

The graph for S_1 has cycles and the accumulated substitutions in the cycles are not just renamings. Indeed, S_1 offers infinitely many opportunities for resolution. The graph for S_2 on the other hand is free of cycles, hence its saturation is finite.

Properties of the clause dependency graph such as its number of nodes, degrees of nodes, and cycles can thus be used as features indicating the proof complexity. One could also consider to include further properties about these cycles related to the accumulated substitution.

All the features mentioned so far in Sections 3 and 4 were described as *static* properties, in the sense that they are computed from the input problem before starting a proof attempt. They could also be considered as *dynamic* features, meaning they are re-evaluated after a certain number of proof steps. This idea was already used in [9]. However, besides revising properties already considered for the initial problem, also characteristics of the proof steps themselves could be considered.

5 Conclusion

In this extended abstract we proposed structural features that can be used in machine learning tasks to infer appropriate strategies or tools for term rewriting or theorem proving problems.

We believe that the development of meaningful features is interesting in multiple respects. One aim is to use machine learning to better predict suitable strategies and tools for a given problem. But once a machine learning model is obtained, it can also be analyzed for its most significant features. If a nontrivial feature turns out to be important, it constitutes also an interesting property of problems that helps to discriminate between different problem classes. This should allow for interesting conclusions about which characteristics are pivotal in the choice of a strategy or a strategy component, and consequently lead to a taxonomy of problems.

The properties described here are, however, not meant to be comprehensive. On the contrary, they rather constitute a collection of initial ideas. In future work, we aim to test the expressibility of the proposed feature set experimentally, also attempting to complement it with further useful properties.

References

[1] H. Andréka, I. Németi & J. van Benthem (1998): *Modal logics and bounded fragments of predicate logic*. *JPL* 27(3), pp. 217–274, doi:10.1023/A:1004275029985.

[2] T. Arts & J. Giesl (2000): *Termination of term rewriting using dependency pairs*. *Theoretical Computer Science* 236(1), pp. 133–178, doi:10.1016/S0304-3975(99)00207-8.

[3] M. Avanzini, N. Eguchi & G. Moser (2015): *A new order-theoretic characterisation of the polytime computable functions*. Theoretical Computer Science 585, pp. 3–24, doi:10.1016/j.tcs.2015.03.003.

[4] M. Avanzini & G. Moser (2013): *Polynomial Path Orders*. Log. Meth. Comput. Sci. 9(4), doi:10.2168/LMCS-9(4:9)2013.

[5] F. Baader & T. Nipkow (1998): *Term Rewriting and All That*. Cambridge University Press, doi:10.1017/CBO9781139172752.

[6] S. Bellantoni & S. Cook (1992): *A new recursion-theoretic characterization of the polytime functions*. Comput. Complex. 2(2), pp. 97–110, doi:10.1145/129712.129740.

[7] A. Ben-Amram (2011): *Monotonicity Constraints for Termination in the Integer Domain*. Log. Meth. Comput. Sci. 7(3), doi:10.2168/LMCS-7(3:4)2011.

[8] A. Ben-Amram & A. Pineles (2016): *Flowchart Programs, Regular Expressions, and Decidability of Polynomial Growth-Rate*. In: Proc. 4th VPT, EPTCS 216, pp. 24–49, doi:10.4204/EPTCS.216.2.

[9] J. P. Bridge, S. Holden & L. Paulson (2014): *Machine Learning for First-Order Theorem Proving*. JAR 53(2), pp. 141–172, doi:10.1007/s10817-014-9301-5.

[10] K. Chvalovský, J. Jakubův, M. Suda & J. Urban (2019): *ENIGMA-NG: Efficient Neural and Gradient-Boosted Inference Guidance for E*. In: Proc. 27th CADE, LNCS 11716, pp. 197–215, doi:10.1007/978-3-030-29436-6_12.

[11] T. Colcombet, L. Daviaud & F. Zuleger (2017): *Automata and Program Analysis*. In: Proc. 21st FCT, LNCS 10472, pp. 3–10, doi:10.1007/978-3-662-55751-8_1.

[12] M. Dauchet & S. Tison (1990): *The Theory of Ground Rewrite Systems is Decidable*. In: Proc. 5thIEEE Symposium on Logic in Computer Science, pp. 242–248, doi:10.1109/LICS.1990.113750.

[13] Y. Demyanova, T. Pani, H. Veith & F. Zuleger (2017): *Empirical Software Metrics for Benchmarking of Verification Tools*. Form. Methods Syst. Des. 50(2-3), pp. 289–316, doi:10.1007/s10703-016-0264-5.

[14] J. Jakubův & J. Urban (2017): *ENIGMA: Efficient Learning-Based Inference Guiding Machine*. In: Proc. CICM 2017, LNCS 10383, pp. 292–302, doi:10.1007/978-3-319-62075-6_20.

[15] J. Jakubův & J. Urban (2018): *Hierarchical invention of theorem proving strategies*. AI Commun. 31(3), pp. 237–250, doi:10.3233/AIC-180761.

[16] C. Kaliszyk & J. Urban (2015): *FEMaLeCoP: Fairly Efficient Machine Learning Connection Prover*. In: Proc. LPAR 2015, LNCS 9450, pp. 88–96, doi:10.1007/978-3-662-48899-7_7.

[17] K. Korovin (2013): *Non-cyclic Sorts for First-Order Satisfiability*. In: Proc. FroCoS 2013, LNCS 8152, pp. 214–228, doi:10.1007/978-3-642-40885-4_15.

[18] D. Kühlwein, S. Schulz & J. Urban (2013): *E-MaLeS 1.1*. In: Proc. 24th CADE, LNCS 7898, pp. 407–413, doi:10.1007/978-3-642-38574-2.

[19] D. Leivant (1994): *Ramified recurrence and computatinal complexity I: Word recurrence and poly-time*. In P. Clote & J. Remmel, editors: Feasible Mathematics II, Birkhäuser, pp. 320–343, doi:10.1007/978-1-4612-2566-9_11.

[20] H. R. Lewis (1980): *Complexity results for classes of quantificational formulas*. JCSS 21(3), pp. 317–353, doi:10.1016/0022-0000(80)90027-6.

[21] S. M. Loos, G. Irving, C. Szegedy & C. Kaliszyk (2017): *Deep Network Guided Proof Search*. In: Proc. 21st LPAR, pp. 85–105, doi:10.29007/8mwc.

[22] S. Schulz (2001): *Learning Search Control Knowledge for Equational Theorem Proving*. In: Proc. KI 2001, LNCS 2174, pp. 320–334, doi:10.1007/3-540-45422-5_23.

[23] H. Simmons (1988): *The realm of primitive recursion*. Archive for Mathematical Logic 27(2), pp. 177–188, doi:10.1007/BF01620765.

[24] G. Sutcliffe (2009): *The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts*. JAR 43(4), pp. 337–362, doi:10.1007/s10817-009-9143-8.