ACUOS^2: A High-performance System for Modular ACU Generalization with Subtyping and Inheritance*

María Alpuente[1], Demis Ballis[2], Angel Cuenca-Ortega[1,3], Santiago Escobar[1], and José Meseguer[4]

[1] DSIC-ELP, Universitat Politècnica de València, Spain
{alpuente,acuenca,sescobar}@dsic.upv.es
[2] DMIF, University of Udine, Italy
demis.ballis@uniud.it
[3] Universidad de Guayaquil, Ecuador. angel.cuencao@ug.edu.ec
[4] University of Illinois at Urbana-Champaign, USA
meseguer@illinois.edu

Abstract. Generalization in order-sorted theories with any combination of associativity (A), commutativity (C), and unity (U) algebraic axioms is finitary. However, existing tools for computing generalizers (also called "anti-unifiers") of two typed structures in such theories do not currently scale to real size problems. This paper describes the ACUOS^2 system that achieves high performance when computing a complete and minimal set of least general generalizations in these theories. We discuss how it can be used to address artificial intelligence (AI) problems that are representable as order-sorted ACU generalization, e.g., generalization in lists, trees, (multi-)sets, and typical hierarchical/structural relations. Experimental results are also given to demonstrate that ACUOS^2 greatly outperforms the predecessor tool ACUOS by running up to five orders of magnitude faster.

1 Introduction

Computing generalizations is relevant in a wide spectrum of automated reasoning areas where analogical reasoning and inductive inference are needed, such as analogy making, case-based reasoning, web and data mining, ontology learning, machine learning, theorem proving, program derivation, and inductive logic programming, among others [5,17,18,20].

This work presents ACUOS^2, a highly optimized implementation of the order-sorted ACU least general generalization algorithm formalized in [3]. ACUOS^2 is a new, high-performance version of a previous prototype called ACUOS [4]. ACUOS^2 runs up to five orders of magnitude faster than ACUOS and is able to solve complex generalization problems in which ACUOS fails to give a response. Both systems are written in Maude [10], a programming language and system that implements rewriting logic [16] and supports reasoning modulo algebraic properties, subtype polimorphism, and reflection. However, ACUOS was developed with a strong concern for simplicity and does not scale to real-life problem sizes, such as the biomedical domains often used in inductive logic programming and other AI applications, with a substantial number of variables, predicates and/or operators per problem instance. Scalability issues were not really unexpected since other equational problems (such as equational matching, equational unification, or equational embedding) are typically much more involved and costly than their corresponding "syntactic" counterparts, and achieving efficient implementations has required years of significant investigation effort (see [11,13,14]).

* This work has been partially supported by the EU (FEDER) and the Spanish MINECO under grant TIN 2015-69175-C4-1-R, by Generalitat Valenciana under grant PROMETEOII/2015/013, and by NSF grant CNS-1319109. Angel Cuenca-Ortega has been supported by the SENESCYT, Ecuador (scholarship program 2013)

For the reader who is not acquainted with least general generalization modulo equations, Section 2 briefly summarizes the problem of generalizing two (typed) expressions in theories that satisfy any combination of associativity (A), commutativity (C) and unity axioms (U). In Section 3, we explain the main functionality of the ACUOS^2 system and describe the novel implementation ideas and optimizations that have boosted the tool performance. An in-depth experimental evaluation of ACUOS^2 is given in Section 4. In Section 5 we briefly discuss some related work. A nontrivial application of equational generalization to a biological domain is described in the Appendix A

2 Least General Generalization modulo A, C, and U Axioms

Roughly speaking, computing a *least general generalization* (lgg) for two expressions t_1 and t_2 means finding the least general expression t such that both t_1 and t_2 are instances of t under appropriate substitutions. For instance, the expression `olympics(X,Y)` is a generalizer of both `olympics(1900,paris)` and `olympics(2024, paris)` but their least general generalizer, also known as *most specific generalizer* (msg) and *least common anti-instance* (lcai), is `olympics(X,paris)`.

Syntactic generalization has two important limitations. First, it cannot generalize common data structures such as records, lists, trees, or (multi-)sets, which satisfy specific premises such as, e.g., the order among the elements in a set being irrelevant. Second, it does not cope with types and subtypes, which can lead to more specific generalizers.

Consider the predicates `connected`, `flights`, `visited`, and `alliance` among cities, and let us introduce the constants `rome`, `paris`, `nyc`, `bonn`, `oslo`, `rio`, and `ulm`. Assume that the predicate `connected` is used to state that a pair of cities `C1;C2` are connected by transportation, with "`;`" being the *unordered pair constructor* operator so that the expressions `connected(nyc;paris)` and `connected(paris;nyc)` are considered to be equivalent modulo the commutativity of "`;`". Then, expressions `connected(nyc;paris)` and `connected(paris;bonn)` can be generalized to `connected(C;paris)`, whereas the syntactic least general (or most specific) generalizer of these two expressions is `connected(C1;C2)`.

Similarly, assume that the predicate `flights(C,L)` is used to state that the city `C` has direct flights to all of the cities in the list `L`. The *list concatenation* operator "`.`" records the cities[5] in the order given by the travel distance from `C`. Due to the associativity of list concatenation, i.e., `(X.Y).Z = X.(Y.Z)`, we can use the flattened list `rio.paris.oslo.nyc` as a very compact and convenient representation of the congruence class (modulo associativity) whose members are the different parenthesized list expressions `((rio.paris).oslo).nyc`, `(rio.(paris.oslo)).nyc`, `rio.(paris.(oslo.nyc))`, *etc.* Then, for the expressions `flights(rome,paris.oslo.nyc.rio)` and `flights(bonn,ulm.oslo.rome)`, the least general generalizer is `flights(C,L1.oslo.L2)`, which reveals that `oslo` is the only common city that has a direct flight from `rome` and `bonn`. Note that `flights(C,L1.oslo.L2)` is more general (modulo associativity) than `flights(rome,paris.oslo.nyc.rio)` by the substitution {`C/rome, L1/paris, L2/(nyc.rio)`} and more general than `flights(bonn,ulm.oslo.rome)` by the substitution {`C/bonn, L1/ulm, L2/rome`}.

Due to the equational axioms ACU, in general there can be more than one least general generalizer of two expressions. As a simple example, let us record the travel history of a person using a list that is ordered by the chronology in which the visits were made; e.g., `visited(paris.paris.bonn.nyc)` denotes that `paris` has been visited twice before visiting `bonn` and then `nyc`. Then, the travel histories `visited(paris.paris.bonn.nyc)` and `visited(bonn.bonn.rome)` have two incomparable least general generalizers: (a) `visited(L1.bonn.L2)` and (b) `visited(C.C.L)`, meaning that (a) the two travelers visited `bonn`, and (b) they consecutively repeated a visit to their own first visited city. Note that the two

[5] A single city is automatically coerced into a singleton list.

generalizers are least general and incomparable, since neither of them is an instance (modulo associativity) of the other.

Furthermore, consider the predicate `alliance(S)` that is used to check whether the cities in the set `S` have established an alliance. In Maude, we can introduce a new operator "`&`" that satisfies associativity, commutativity, and unit element $\emptyset$; i.e., `X & ∅ = X` and `∅ & X = X`. We can use the flattened, multi-set notation `alliance(nyc & oslo & paris & rome)` as a very compact and convenient representation (with a total order on elements given by the lexicographic order) for the congruence class modulo ACU whose members are all of the different parenthesized permutations of the considered cities. Such permutations include as many occurrences of $\emptyset$ as needed, due to unity [12]. In this scenario, the expressions (i) `alliance(nyc & oslo & paris & rome)` and (ii) `alliance(bonn & paris & rio & rome)` have an infinite set of ACU generalizers of the form `alliance(paris & rome & S1 & ⋯ & Sn)` yet they are all equivalent modulo ACU-renaming[6] so that we can choose one of them, typically the smallest one, as the class representative.

Regarding the handling of types and subtypes, let us assume that the constants `rome`, `paris`, `oslo`, `ulm`, and `bonn` belong to type European and that `nyc` and `rio` belong to type American. Furthermore, let us suppose that European and American are subtypes of a common type City that, in turn, is a subtype of the type Cities that can be used to model the typed version of the previous ACU (multi-)set structure. Subtyping implies automatic coercion: for instance, a European city also belongs to the type City and Cities. Note that the empty set, denoted by the unity $\emptyset$, only belongs to Cities.

In this typed environment, the above expressions (i) and (ii) have only one typed ACU least general generalizer `alliance(paris & rome & C1:American & C2:European)` that we choose as the representative of its infinite ACU congruence class. Note that `alliance(paris & rome & S:Cities)` is not a least general generalizer since it is strictly more general; it suffices to see that the typed ACU-lgg above is an instance of it modulo ACU with substitution {`S:Cities/(C1:American & C2:European)`}.

Due to space restrictions, in this work we do not deal with higher-order generalization problems, in which operation symbols themselves are generalized by using function variables [15]. For a discussion on how to achieve higher-order generalization in Maude we refer to [4].

3 ACUOS²: A High Performance ACU Generalization System

ACUOS² is a new, totally redesigned, implementation of the ACUOS system presented in [4] that provides a remarkably faster and more optimized computation of least general generalizations. Generalizers are computed in an order-sorted, typed environment where inheritance and subtype relations are supported modulo any combination of associativity, commutativity, and unity axioms.

Both ACUOS and ACUOS² implement the generalization calculus of [3] but with remarkable differences concerning how they deal with the combinatorial explosion of different alternative possibilities; see [19] for some theoretical results on the complexity of generalization. Consider the generalization problem

$$\texttt{connected}(\texttt{paris};\texttt{bonn}) \stackrel{\triangle}{=} \texttt{connected}(\texttt{bonn};\texttt{paris})$$

that is written using the syntax of [3]. ACUOS already includes some optimizations but follows [3] straightforwardly and decomposes this problem (modulo commuta-

[6] i.e., the equivalence relation the equivalence relation $\approx_{ACU}$ induced by the relative generality (subsumption) preorder $\leq_{ACU}$, i.e., $s \approx_{ACU} t$ iff $s \leq_{ACU} t$ and $t \leq_{ACU} s$.

tivity of ";") into two simpler subproblems:

$$(P_1)\ \texttt{paris} \stackrel{\triangle}{=} \texttt{bonn} \wedge \texttt{bonn} \stackrel{\triangle}{=} \texttt{paris} \qquad (P_2)\ \texttt{paris} \stackrel{\triangle}{=} \texttt{paris} \wedge \texttt{bonn} \stackrel{\triangle}{=} \texttt{bonn}$$

According to [3], both are explored non-deterministically even if only the last subproblem would lead to the least general generalization. Much worse, due to axioms and types, a post-generation, time-expensive filtering phase is necessary to get rid of non-minimal generalizers. We have derived four groups of optimizations: (a) avoid non-deterministic exploration; (b) reduce the number of subproblems; (c) prune non-minimal paths to anticipate failure; and (d) filter out non-minimal solutions more efficiently.

(a) While ACUOS directly encoded the inference rules of [3] as rewrite rules that non-deterministically compute generalizers by exploring all branches of the search tree in a don't-know manner, i.e., each branch potentially leads to a different solution, ACUOS^2 smartly avoids non-deterministic exploration by using *synchronous rewriting* [7], also called *maximal parallel rewriting*, that allows ACUOS^2 to keep all current subproblems in a single data structure, e.g. $P_1 \mid P_2 \mid \cdots \mid P_n$, where all subproblems are simultaneously executed, avoiding any non-deterministic exploration at all. Synchronous rewriting is achieved in Maude by reformulating rewrite rules as oriented equations and, thanks to the different treatment of rules and equations in Maude [12], the deterministic encoding of the inference rules significantly reduces execution time and memory consumption. Also, built-in Maude memoization techniques are applied to speed up the evaluation of common subproblems, which can appear several times during the generalization process.

(b) Enumeration of all possible terms in a congruence class is horribly inefficient, and even nonterminating when the U axiom is considered. Therefore, it should not be used to effectively solve generalization problems when A, AC, or ACU axiom combinations are involved. For instance, if f is AC, the term $f(a_1, f(a_2, \ldots, f(a_{n-1}, a_n), \ldots))$ has $(2n-2)!/(n-1)!$ equivalent combinations; this number may grow exponentially for generalization problems that contain several symbols obeying distinct combinations of axioms.
ACUOS^2 avoids class element enumeration (specifically the expensive computation of argument permutations for *AC* operators). Instead, it relies on the extremely efficient Maude built-in support for equational matching to decompose generalization problems into simpler subproblems, thereby achieving a dramatic improvement in performance.

(c) It is extremely convenient to discard as early as possible any generalization subproblem that will not lead to a least general generalization. For example, trivial generalization problems such as $\texttt{paris} \stackrel{\triangle}{=} \texttt{paris}$ are immediately solved once and for all without any further synchronous rewrite. Similarly, dummy generalization problems with single variable generalizers such as $\texttt{nyc} \stackrel{\triangle}{=} \texttt{paris}$ are solved immediately. However, note that $\texttt{paris.oslo} \stackrel{\triangle}{=} \texttt{nyc.oslo}$ is not a dummy problem. ACUOS^2 also checks whether a subproblem is more general than another during the whole process, discarding the more general one. For instance, P_1 above contains two dummy subproblems and P_2 above contains two trivial subproblems, which *safely* allows ACUOS^2 to discard P_1 as being more general than P_2.

(d) Getting rid of non-minimal generalizers commonly implies too many pairwise comparisons, i.e., whether a generalizer l_1 is an instance *modulo* axioms of a generalizer l_2, or viceversa. Term size is a very convenient ally here since a term t' being bigger than another term t prevents t from being an instance of t'. Note that this property is no longer true when there is a unit element. For instance, `alliance(nyc & rome & S1:Cities & S2:Cities)` is bigger (modulo ACU) than `alliance(nyc & rome & S:Cities)`; but the latter is an instance of the former by the substitution {`S1/S, S2/`$\emptyset$}. Term size can reduce the number of matching comparisons by half.

The ACUOS^2 backend has been implemented in Maude and consists of about 2300 lines of code. It can be directly invoked in the Maude environment by calling the generalization routine `lggs(M,t1,t2)`, which facilitates ACUOS^2 being integrated with third-party software. Furthermore, ACUOS^2 functionality can be accessed through an intuitive web interface that is publicly available at the ACUOS^2 web site [1]. The interface comes with a preloaded collection of generalization problems that is provided with the tool for demonstration purposes. New generalization problems can be specified from scratch by filling the dedicated edit areas.

4 Experimental Evaluation

To empirically evaluate the performance of ACUOS^2 and fairly compare it with ACUOS, we have considered the same generalization problems that were used to benchmark ACUOS in [4], together with some additional problems that deal with complex ACU structures such as graphs and biological models. All of the problems are available online at the tool web site [1] where the reader can also reproduce all of the experiments we conducted through the ACUOS^2 web interface. Specifically, the benchmarks used for the analysis are: (i) `incompatible types`, a problem without any generalizers; (ii) `twins`, `ancestors`, `spouses`, `siblings`, and `children`, some problems borrowed from the logic programming domain which are described in [4]; (iii) `only-U`, a generalization problem modulo (just) unity axioms, i.e., without A and C; (iv) `synthetic`, an involved example mixing A, C, and U axioms for different symbols; (v) `multiple inheritance`, which uses a classic example of multiple subtyping from [12] to illustrate the interaction of advanced type hierarchies with order-sorted generalization; (vi) `rutherford`, the classical analogy-making example that recognizes the duality between Rutherford's atom model and the solar system [15]; (vii) `chemical`, a variant of the case-based reasoning problem for chemical compounds discussed in [5]; (viii) `alliance`, the ACU example of Section 2; (ix) `graph`, the leading example of [9]; and (x) `biological`, a cell model borrowed from [21] (see Appendix).

We tested our implementations on a 3.30 GHz Intel(R) Xeon(R) E5-1660 with 64Gb of RAM memory running Maude v2.7.1, and we considered the average of ten executions for each test. Table 1 shows our experimental results. For each problem, we show the size (i.e., number of operators) of the input terms, the computation time (ms.) until the first generalization is found[7], and the number $\#S$ of different subproblems that were generated so far, as a measure of how much the complexity of the problem has been simplified (before the optimizations, the number of produced subproblems was typically in the thousands for a term size of 100). In many cases, we cannot compare the time taken by each system to compute the set of all lggs, since the previous prototype ACUOS times out (for a generous timeout that we set to 60 minutes). Indeed, when we increase the size of the input terms from 20 to 100, the generalization process in ACUOS stops for most of the benchmarks due to timeout.

Considering the high combinatorial complexity of the ACU generalization problem, our implementation is highly efficient. All of the examples discussed in [4], except for `incompatible types`, `twins (C)`, and `synthetic (C + AU)`, fail to produce a generalization in ACUOS when the problem size is 100, whereas the time taken in ACUOS^2 is in the range from 1 to 11067ms ($\sim 11s$). In all of the benchmarks, our figures demonstrate an impressive performance boost w.r.t. [4]: a speed up of five orders of magnitude for all of the ACU benchmarks.

5 Related work

Related (but essentially different) problems of anti-unification for feature terms have been studied by [2], [5], and [6]. The minimal and complete unranked anti-

[7] The computation time for the benchmark called `incompatible types` is the same for any input term, since we provide two input terms that have a different, incompatible sort.

Benchmark	$\#S$	**Size**	ACUOS T1(ms)	ACUOS² T2(ms)	**Speedup** × (T1/T2)
incompatible types	0	20	30	1	30
	0	100	30	1	30
twins (C)	16	20	70	8	9
	42	100	23934	70	340
ancestors (A)	10	20	48	1	48
	31	100	TO	48	$>10^5$
spouses (A)	10	20	49	1	49
	31	100	TO	50	$>10^5$
spouses (AU)	10	20	531747	5	$\sim 10^5$
	61	100	TO	30	$>10^5$
siblings (AC)	16	20	TO	1	$>10^5$
	23	100	TO	150	$>10^5$
children (ACU)	12	20	TO	2	$>10^5$
	29	100	TO	3451	$>10^5$
only-U (U)	9	20	24	2	12
	9	100	TO	630	$>10^5$
synthetic (C+AU)	5	20	55	1	55
	5	100	31916	50	638
multiple inheritance (AC)	17	20	TO	10	$>10^5$
	31	100	TO	11067	$>10^5$
rutherford (AC+A+C)	5	20	48	1	48
	42	100	TO	320	$>10^5$
chemical (AU)	15	20	112	1	112
	31	100	TO	10	$>10^5$
graph (ACU+AU)	11	20	TO	1	$>10^5$
	31	100	TO	1002	$>10^5$
biological (ACU+AC+A)	22	20	TO	4	$>10^5$
	71	100	TO	50	$>10^5$
alliance (ACU)	11	20	TO	1	$>10^5$
	31	100	TO	9159	$>10^5$

Table 1: Experimental results

unification of [8] and the term graph anti-unification of [9] (together with the commutative extension) are also related to our work. The unranked anti-unification problems of [8,9] can be directly solved by using our techniques for associative anti-unification with the unit element by simply introducing sorts to distinguish between term variables and hedge variables (and their instantiations) [8]. Conversely, it is possible to simulate our calculus for associative least general generalization with the unit element in the minimal and complete unranked anti-unification algorithm of [8], but not the rules for associative-commutative least general generalization with the unit element.

As for the generalization of feature terms, this problem has two main similarities with computing (least general) generalizations modulo combinations of A, C, and U axioms: 1) feature terms are order-sorted (in contrast to the unsorted setting of unranked term anti-unification); and 2) there is no fixed order for arguments. However, the capability to deal with recursive, possibly cyclic data structures such as graphs in ACU anti-unification does not seem to have its counterpart in feature term anti-unification. Moreover, to generalize theories with a different number of clauses/equations (or a different number of atoms per clause), feature generalization algorithms have to resort to *ad hoc* mechanisms such as *background theories* and *projections* [15], whereas our approach naturally handles these kinds of generalizations by defining operators that obey the unity axiom.

References

1. The ACUOS^2 Website (2018), http://safe-tools.dsic.upv.es/acuos2
2. Aït-Kaci, H., Sasaki, Y.: An axiomatic approach to feature term generalization. In: Machine Learning: EMCL 2001, 12th European Conference on Machine Learning, Freiburg, Germany, September 5-7, 2001, Proceedings. pp. 1–12 (2001)
3. Alpuente, M., Escobar, S., Espert, J., Meseguer, J.: A Modular Order-sorted Equational Generalization Algorithm. Information and Computation 235, 98–136 (2014)
4. Alpuente, M., Escobar, S., Espert, J., Meseguer, J.: ACUOS: A System for Modular ACU Generalization with Subtyping and Inheritance. In: Proc. of the 14th European Conference on Logics in Artificial Intelligence (JELIA 2014). Lecture Notes in Computer Science, vol. 8761, pp. 573–581. Springer-Verlag (2014)
5. Armengol, E.: Usages of Generalization in Case-Based Reasoning. In: Proc. of ICCBR 2007. LNCS, vol. 4626, pp. 31–45. Springer-Verlag, Berlin, Heidelberg (2007)
6. Armengol, E., Plaza, E.: Symbolic explanation of similarities in case-based reasoning. Computers and Artificial Intelligence 25(2-3), 153–171 (2006)
7. Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
8. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: A variant of higher-order anti-unification. In: van Raamsdonk, F. (ed.) Proc. of 24th International Conference on Rewriting Techniques and Applications, RTA 2013. LIPIcs, vol. 21, pp. 113–127. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2013)
9. Baumgartner, A., Kutsia, T., Levy, J., Villaret, M.: Term-graph anti-unification. In: FSCD. LIPIcs, vol. 108, pp. 9:1–9:17. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2018)
10. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.7.1), July 2016, http://maude.cs.illinois.edu
11. Clavel, M., Durán, F., Eker, S., Escobar, S., Lincoln, P., Martí-Oliet, N., Talcott, C.L.: Two decades of Maude. In: Martí-Oliet, N., Ölveczky, P.C., Talcott, C.L. (eds.) Logic, Rewriting, and Concurrency - Essays dedicated to José Meseguer on the Occasion of His 65th Birthday. Lecture Notes in Computer Science, vol. 9200, pp. 232–254. Springer-Verlag (2015)
12. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
13. Durán, F., Eker, S., Escobar, S., Martí-Oliet, N., Meseguer, J., Talcott, C.L.: Associative unification and symbolic reasoning modulo associativity in Maude. In: Rusu, V. (ed.) Rewriting Logic and Its Applications - 12th International Workshop, WRLA 2018. Lecture Notes in Computer Science, vol. 11152, pp. 98–114. Springer (2018)
14. Eker, S.: Associative-Commutative Rewriting on Large Terms. In: Nieuwenhuis, R. (ed.) Rewriting Techniques and Applications, 14th International Conference, RTA 2003. LNCS, vol. 2706, pp. 14–29. Springer (2003)
15. Gentner, D.: Structure-Mapping: A Theoretical Framework for Analogy*. Cognitive Science 7(2), 155–170 (1983)
16. Meseguer, J.: Conditioned rewriting logic as a united model of concurrency. Theor. Comput. Sci. 96(1), 73–155 (1992)
17. Muggleton, S.: Inductive Logic Programming: Issues, Results and the Challenge of Learning Language in Logic. Artif. Intell. 114(1-2), 283–296 (1999)
18. Ontañón, S., Plaza, E.: Similarity measures over refinement graphs. Machine Learning 87(1), 57–92 (Apr 2012)
19. Pottier, L.: Generalisation de termes en theorie equationelle: Cas associatif-commutatif. Tech. Rep. INRIA 1056, Norwegian Computing Center (1989)
20. Schmid, U., Hofmann, M., Bader, F., Häberle, T., Schneider, T.: Incident Mining using Structural Prototypes. In: Proc. of IEA-AIE 2010. LNCS, vol. 6097, pp. 327–336. Springer-Verlag, Berlin, Heidelberg (2010)
21. Talcott, C.: Pathway logic. Formal Methods for Computational Systems Biology 5016, 21–53 (2008)

A An Application of ACU Generalization to a Biological Domain

In this section, we show how ACUOS^2 can be productively used to analyze biological systems, e.g., to extract similarities and pinpoint discrepancies between two cell models that express distinct cellular states. To illustrate our example, we consider cell states that appear in the MAPK (Mitogen-Activated Protein Kinase) metabolic pathway that regulates growth, survival, proliferation, and differentiation of mammalian cells.

Our cell formalization is inspired by and slightly modifies the data structures used in Pathway Logic (PL) [21] —a symbolic approach to the modeling and analysis of biological systems that is implemented in Maude. Specifically, a cell state can be specified as a typed term as follows.

We use sorts to classify cell entities. The main sorts are `Chemical`, `Protein`, and `Complex`, which are all subsorts of sort `Thing`, which specifies a generic entity. Cellular compartments are identified by sort `Location`, while `Modification` is a sort that is used to identify post-transactional protein modifications, which are defined by the operator "`[-]`" (e.g., the term `[EgfR - act]` represents the Egf (epidermal growth factor) receptor in an active state). A complex is a compound element that is specified by means of the associative ad commutative (AC) operator "`<=>`", which combines generic entities together.

Now, a *cell state* is represented by a term of the form `[cellType | locs]`, where `cellType` specifies the cell type [8] and `locs` is a list (i.e., an associative data structure whose constructor symbol is "`,`") of cellular compartments (or locations). Each location is modeled by a term of the form { `locName | comp` }, where `locName` is a name identifying the location (e.g., `CLm` represents the cell membrane location), and `comp` is a soup (i.e., an associative and commutative data structure with unity element `empty`) that specifies the entities included in that location. Note that cell states are built by means of a combination of A, AC, and ACU operators. The full formalization of the considered cell specification is given in Section B.

Example 1. The term c_1

```
[ mcell | { Clc | Gab1 Grb2 Plcg Sos1 },
          { CLm | EgfR PIP2},
          { CLi | [Hras - GDP] Src } ]
```

models a cell state of the MAPK pathway with three locations: the cytoplasm (`CLi`) includes five proteins `Gab1`, `Grb2`, `Pi3k`, `Plcg`, and `Sos1`; the membrane (`CLm`) includes the protein `EgfR` and the chemical `PIP2`; the membrane interior (`CLi`) includes the proteins `Hras` (modified by `GDP`) and `Src`.

In this scenario, ACUOS^2 can be used to compare two cell states, c_1 and c_2. Indeed, any ACUOS^2 solution for the problem of generalizing c_1 and c_2 is a term whose non-variable part represents the common cell structure shared by c_1 and c_2, while its variables highlight discrepancy points where the two cell states differ.

Example 2. Consider the problem of generalizing the cell state of Example 1 plus the following MAPK cell state c_2

```
[ mcell | { CLc | Gab1 Plcg Sos1 },
          { CLm | PIP2 Egf <=> [EgfR - act] },
          { CLi | Grb2 Src [Hras - GDP] } ]
```

For instance, ACUOS^2 computes (in 4ms) the following least general generalizer

```
[ mcell | { CLc | Gab1 Plcg Sos1 X1:Soup },
          { CLm | PIP2 X2:Thing },
          { CLi | Src X3:Soup [Hras - GDP] } ]
```

[8] To simplify the exposition, we only consider mammalian cells denoted by the constant `mcell`.

where X1:Soup, X2:Thing, and X3:Soup are typed variables. Each variable in the computed lgg detects a discrepancy between the two cell states. The variable X2:Thing represents a generic entity that abstracts two distinct elements in the membrane location CLm of the two cell states. In fact, c_1's membrane includes the (inactive) receptor EgfR, whereas c_2's membrane contains the complex Egf <=> [EgfR - act] that activates the receptor EgfR and binds it to the ligand Egf to start the metabolic process. Variables X1:Soup and X3:Soup indicate a protein relocation for Grb2, which appears in the location CLc in c_1 and in the membrane interior CLi in c_2. Note that the computed sort Soup is key in modeling the absence of Grb2 in a location, since it allows X1:Soup and X3:Soup to be bound to the empty soup.

B A Simplified Maude Specification for a Mammalian Cell

```
fmod CELL-STRUCTURE is
  sorts Protein Thing Complex Chemical .
  subsorts Protein Complex Chemical < Thing .

  op _<=>_ : Thing Thing -> Complex [assoc comm] .

  op Egf : -> Protein .
  op EgfR : -> Protein .
  op ErbB2 : -> Protein .
  op Pi3k : -> Protein .
  op Gab1 : -> Protein .
  op Grb2 : -> Protein .
  op Hras : -> Protein .
  op Plcg : -> Protein .
  op Sos1  : -> Protein .
  op Src : -> Protein .
  op Tgfa   : -> Protein .
  op PIP2 : -> Chemical .
  op PIP3 : -> Chemical .

  sort Soup .
  subsort Thing < Soup .
  op empty : -> Soup .
  op __ : Soup Soup -> Soup [assoc comm id: empty ] .

  sorts Site Modification .
  op act : -> Modification .
  ops GTP GDP : -> Modification .
  op [_-_] : Protein   Modification -> Protein   .

  sort Location LocName .
  op {_|_} : LocName Soup -> Location .
  ops CLm CLi CLc : -> LocName .

  sort Locations .
  subsort Location < Locations .
  op _,_ : Locations Locations -> Locations [assoc] .

  sorts Cell CellType .
  op [_|_] : CellType Locations -> Cell .
  op mcell : -> CellType .
endfm
```