Collaborative Programming: Applications of logic and automated reasoning

Timothy L. Hinrichs

University of Chicago
tlh@uchicago.edu

Abstract. Collaborative Programming is characterized by groups of people issuing instructions to computer systems. Collaborative Programming languages differ from traditional programming languages because instruction sets can be incomplete and conflicting, and more of the burden for efficient execution is placed on the computer system. This paper introduces Collaborative Programming and through the discussion of two practical examples argues that tools from logic and automated reasoning form a good foundation for Collaborative Programming technology while at the same time illustrating the need for nonstandard automated reasoning techniques.

1 Introduction

Collaborative Programming comprises all those settings where groups of people issue instructions to computer systems. In contrast to traditional programming languages, Collaborative Programming languages must make combining instruction sets from different parties straightforward and may allow users to express incomplete and conflicting[1] instruction sets. An *incomplete* instruction set may only say what to do some of the time or what actions the system is forbidden from performing. A *conflicting* instruction set may simultaneously instruct the system to perform some action and forbid the system from performing that same action. Technology that supports Collaborative Programming must be able to combine independently authored instruction sets and be tolerant of incompleteness and conflicts.

The notion of Collaborative Programming was developed as a pedagogical device for explaining to researchers in traditional programming languages and systems (e.g. networks, operating systems) the benefits and limitations of logical languages and automated reasoning as compared to more traditional approaches. The word "collaborative" was chosen to capture situations where statements made by independent parties must be combined, a simple operation in logical languages. The word "programming" was chosen to capture situations in which the statements made by independent parties can be construed as instructions to

[1] Here we use the word "conflicting" as opposed to "inconsistent" to differentiate the informal notion of a disagreement and the proof and model theoretic notions of consistency.

a computer system, which is often the case when the statements are made in a formal language. The concept of Collaborative Programming covers situations that leverage the order-irrelevance and formal semantics of logical languages.

The connection between Collaborative Programming and logical languages was forged because of the need and ability to combine instruction sets; however, the connection runs deeper than that. In collaborative settings, it is very natural for users to submit incomplete and conflicting instruction sets. Sometimes people only have opinions on some issues; thus, for a language to reflect a user's true intentions, it must allow users to express incomplete instruction sets, a natural feature of many logical languages. Likewise, when collaborating, people rarely agree on everything; hence, a Collaborative Programming language must allow users to express disagreements, another feature of logical languages. Thus, logical languages are a natural foundation for Collaborative Programming languages, which means that Collaborative Programming language implementations rely on tools from automated reasoning.

Some of the most celebrated tools in automated reasoning, e.g. first-order theorem provers, are designed to detect a particular kind of conflict: a logical inconsistency. More precisely, they determine whether or not an inconsistency exists. As we illustrate in this paper, Collaborative Programming applications sometimes require knowing more than whether or not a conflict exists; they must act based on the type of conflict that occurred. To meet this requirement, theorem provers for Collaborative Programming applications must implement a paraconsistent entailment relation [14]: one that coincides with classical entailment for consistent theories but is more discerning for inconsistent theories.

Paraconsistent theorem provers must overcome an additional computational burden as compared to traditional programming languages. Given a set of instructions issued in a logical language, a computer system must determine which action to perform by analyzing those instructions, resolving conflicts, and filling in gaps. Unlike traditional programming languages, where computing the next action is guaranteed to be fast, computing the next action in a Collaborative Programming setting might require significant computation, which is especially worrisome for real-world applications where efficiency guarantees are important. To alleviate such concerns, we advocate custom-designing a Collaborative Programming language for each application so that it is expressive enough to be useful but no less efficient than is tolerable.

When custom-designing a Collaborative Programming language based on logic, one must choose which style of logic to use. In this paper we consider two specific logics, FHL and DATALOG$^{\neg}$, that from the perspective of Collaborative Programming represent two interesting language classes: classical logic and logic programming. FHL is a decidable fragment of first-order logic that allows arbitrary quantification (syntactically). DATALOG$^{\neg}$ is a language for describing and querying relational databases, perhaps the most successful application of logic in computer science. These two languages were chosen because FHL provides the opportunity to confront paraconsistency, while DATALOG$^{\neg}$ demonstrates that classical logics are not the only option for Collaborative Programming.

This paper examines Collaborative Programming languages for two practical applications: logical spreadsheets (Section 3) and authorization languages (Section 4). In each case, the strengths and weaknesses of FHL and DATALOG$^\neg$ are examined, and in the end one is chosen as the foundation of the language; additionally, issues surrounding conflicts and incompleteness for the chosen language are illustrated and resolved. Finally, we make some closing remarks (Section 5).

2 Preliminaries

The two languages studied in this paper, FHL and DATALOG$^\neg$, are well-known; we use common conventions for their syntax and semantics. FHL, a classical logic, is first-order logic with equality and the following three restrictions: no function constants, a domain closure assumption (DCA), and a unique names assumption (UNA). In this logic, the objects in the universe of every model are exactly the object constants in the language. We call this logic Finite Herbrand Logic (FHL) because the only models of interest are the finite Herbrand models.

The definitions for FHL's syntax are the same as for function-free first-order logic. The definitions for a model and for satisfaction are standard but are simplified to take advantage of the UNA and DCA. A model in FHL is a set of ground atoms from the language. A model satisfies all the ground atoms included in the set. Satisfaction is defined as usual for the Boolean connectives applied to ground sentences. Satisfaction of non-ground sentences reduces to satisfaction of ground sentences. Free variables are implicitly universally quantified. $\forall x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for every object constant a. $\exists x.\phi(x)$ is satisfied exactly when $\phi(a)$ is satisfied for some object a.

The other language of interest, DATALOG$^\neg$, is DATALOG with stratified negation. Again, the definitions for its syntax are standard, and we focus on semantics. A model for DATALOG$^\neg$ is the same as that for FHL: a set of ground atoms; however, in contrast to FHL where sentences may be satisfied by more than one model, a set of DATALOG$^\neg$ sentences is always satisfied by exactly one model. Without negation, that model is the smallest one (ordered by subset) that satisfies the sentences under the FHL definition of satisfaction. With negation, the stratified semantics [15] use minimality criteria to choose one model out of all those that satisfy the sentences under the FHL definition.

A set of sentences is satisfiable (or consistent) when there is at least one model that satisfies it. Logical entailment is defined as usual: $\Delta \models \phi$ if and only if every model that satisfies Δ also satisfies ϕ. Entailment for FHL is coNEXPTIME-complete [5], and entailment for DATALOG$^\neg$ is NEXPTIME-complete, e.g. [16].

FHL and DATALOG$^\neg$ are similar because they are both Herbrand-based logics. They are different in that FHL allows a sentence set to be satisfied by multiple models, whereas a DATALOG$^\neg$ sentence set is always satisfied by exactly one model. For the purposes of this paper, the most important consequence of this distinction is that FHL can express true disjunction (entailing/satisfying a disjunction without entailing/satisfying any disjunct) but DATALOG$^\neg$ cannot.

3 Use Case: Logical Spreadsheets

One area of research, popular enough to support a dedicated workshop in 2005 and a DARPA funding opportunity (in the small business sector) in 2004 [9], investigates the application of logic and automated reasoning to bring about the next generation of spreadsheets for the personal computer. These logical spreadsheets remove some of the limitations of traditional spreadsheets. Instead of equations that specify how to compute the value of one cell given the values of other cells, logical spreadsheeets accept arbitrary logical formulae, which allows updates to propagate in any direction and cells to be constrained to obey many-to-many relationships.

For example, using a logical spreadsheet one can require two cells to be assigned the same value; fill in the value of either cell, and the other one updates automatically. In addition, it is possible to constrain one cell to contain a postal code and another cell to contain a city. The postal code is not sufficient to compute the city, nor is the city sufficient to compute the postal code. Nevertheless, choosing a city restricts the possible postal codes, and vice versa.

Logical spreadsheets allow users to specify a set of constraints on the cells in the spreadsheet and then provide visual cues to indicate which values do not satisfy the constraints. Those visual cues include highlighting cells whose values conflict with the constraints and showing a list of values for any given cell that satisfy the constraints given the values of the other cells.

Particularly well-known examples of logical spreadsheets are the HTML forms found on the web. When ordering merchandise from e-commerce web sites, a form that asks for billing information often includes constraints on the combinations of values that can be entered, e.g. the city and postal code must be compatible. Often, web programmers use Javascript to check those constraints as the user enters information. When a constraint violation occurs, an error message appears somewhere on the page.

The difficulty with using Javascript to check constraints is that if the constraints change, the Javascript may require a substantial rewrite. Research into logical spreadsheets has the potential benefit that a web programmer could write down the necessary constraints for the web form elements in a logical language, and the Javascript for checking those constraints would be generated automatically. Small constraint changes that result in large Javascript changes would no longer be problematic because those large changes would be auto-generated.

Different approaches to logical spreadsheets expose different languages for users to express constraints. The language presented here is based on FHL and follows the presentation in [8]. Cells in the spreadsheet correspond to monadic predicates, and a (partial) cell assignment corresponds to a set of ground atoms. Constraints on a spreadsheet are FHL sentences.

For example, to require two cells named $cell_1$ and $cell_2$ to contain the same value, a user could enter the following sentence.

$$cell_1(x) \Leftrightarrow cell_2(x)$$

Likewise, to force the *postal* cell and the *city* cell to contain compatible values, one could write the implication

$$postal(x) \wedge city(y) \Rightarrow compatible(x, y),$$

where *compatible* is appropriately defined. Assigning cell *city* the value *paris* is represented by the atom $city(paris)$.

Conflicts in this language correspond to inconsistent FHL theories. This is problematic because using the traditional notions of satisfaction and entailment, there is no way to differentiate one conflict from another, which is vital information for visually indicating which cells fail to satisfy the constraints.

For example, consider again the web form where two cells are required to contain the same value, and a city cell and a postal code cell are required to contain compatible values. The constraints are the two sentences shown above. Assigning $cell_1$ and $cell_2$ different values causes an inconsistency, i.e. there are no models that satisfy the constraints together with the assignments to $cell_1$ and $cell_2$. This means that every sentence in the language is entailed. Compare this conflict with a conflict that occurs because the city and postal code cells were assigned incompatible values. Again, the theory is inconsistent, which means there are no satisfying models, and all sentences are entailed. Neither satisfaction nor entailment is sufficient for providing the user feedback as to which cells conflict with each other.

Such problems are addressed by work on paraconsistent logics, e.g. [14]. A paraconsistent logic is one in which an inconsistent theory does not entail all logical sentences. The approach described in [8], called *existential entailment* and denoted $\models_E$, combines the traditional notions of satisfaction and entailment in a simple way. In the case of consistent theories, traditional entailment and existential entailment coincide, but in the case of inconsistent theories, existential entailment isolates one conflict from another.

Intuitively, the problem with traditional entailment is that an inconsistent premise set entails every sentence, even if that inconsistency has nothing to do with the sentence in question. For example, the three premises below are inconsistent, which means that both the sentences $q(a)$ and $\neg q(a)$ are entailed.

$$\begin{array}{l} p(a) \\ \neg p(a) \\ q(a) \end{array} \tag{1}$$

However $q(a)$ would be entailed even without the inconsistency, but $\neg q(a)$ is only entailed because of the inconsistency. Existential entailment differentiates these two cases by requiring a satisfiable premise set for proving a conclusion.

Definition 1 (Existential Entailment [8]). *A set of sentences Δ existentially entails a sentence ϕ ($\Delta \models_E \phi$) if and only if there is some satisfiable Δ' that is a subset of Δ such that $\Delta' \models \phi$.*

Existential entailment can be employed as follows to pinpoint those cells in a spreadsheet that conflict with the constraints. Suppose that the constraints are

satisfiable and named Δ and that the assignments of values to cells is Γ. Recall that assigning cell p to value a is represented as $p(a)$. Cell value $p(a)$ conflicts with the constraints and other cell values whenever $\Delta \cup \Gamma$ existentially entails the negation of $p(a)$.

Definition 2 (Logical Spreadsheet Conflict). *Cell p assigned to value a conflicts with the spreadsheet constraints Δ and the partial cell assignment Γ exactly when $\Delta \cup \Gamma \models_E \neg p(a)$.*

We can view Example 1 from above as a set of constraints and cell values for a spreadsheet with a cell p and a cell q. Cell p, having been assigned the value a, should be highlighted as a conflict because $\neg p(a)$ is existentially entailed (by the singleton, satisfiable premise set $\{\neg p(a)\}$). But, cell q assigned a should not be highlighted as a conflict because $\neg q(a)$ is not existentially entailed.

Using this definition of conflict, every time a user changes the value of a cell, the logical spreadsheet must compute existential entailment. Moreover, one cell assignment can cause other cells to violate constraints, meaning that multiple existential entailment queries must be answered for each cell assignment change. Thus, it is important that the computation of existential entailment runs efficiently enough for the spreadsheet to provide real-time visual cues to the user.

Our current implementation focuses on the web form application of logical spreadsheets. It converts a given set of constraints into conjunctive database queries that when evaluated compute existential entailment. Those queries are evaluated by the browser each time a cell value is changed using an in memory database implemented in Javascript. Preliminary testing appears promising both in ease of implementation and performance.

It is noteworthy that the choice to use FHL as the constraint language was not made arbitrarily. When compared to DATALOG$^\neg$, FHL is better suited as the foundation of the constraint language because it can express disjunction[2], whereas DATALOG$^\neg$ cannot. The importance of disjunction for logical spreadsheets can be seen in two ways.

First, FHL semantics is closer in spirit to a natural formalization of logical spreadsheets than is DATALOG$^\neg$. From a mathematical perspective, a logical spreadsheet maps a set of constraints and a partial assignment of cells to the set of all consistent extensions to that assignment. Similarly, FHL semantics maps a set of logical sentences to the set of models that satisfy those sentences. Both map the input to a set of alternatives. In contrast, DATALOG$^\neg$ semantics maps a set of sentences to a single model—to a single alternative.

Second, one of the features logical spreadsheets support that traditional spreadsheets do not, bidirectional update, is intimately tied to disjunction. A simple implication such as $cell_1(a) \Leftarrow cell_2(a)$ represents two possibilities: either the premises are false or the conclusion is true. For bidirectional update to be supported, falsifying the conclusion of the implication requires falsifying the premise, and satisfying the premise requires satisfying the conclusion. These

[2] Here we mean true disjunction: in FHL a theory can entail $p \vee q$ without entailing either p or q.

two equally plausible possibilities are represented succinctly by a disjunction: $cell_1(a) \vee \neg cell_2(a)$.

Logical spreadsheets exemplify Collaborative Programming because the instructions issued by users can conflict, can be incomplete, and can come from multiple sources. Collaboration comes about in a variety of ways. In the case of web forms, the form developers contribute the constraints and the users contribute data. In the case of a standalone application, constraints might originate from different people, each with expertise in different areas of the problem. Even if all of the constraints are created by a single individual, that individual might be collaborating with herself if over time she adds new constraints to the system. Collaboration breeds conflict, and because FHL, the constraint language, is a classical logic, the traditional notion of entailment does not support the functionality promised by the logical spreadsheet paradigm; hence, a paraconsistent entailment relation must be used and implemented efficiently.

4 Use Case: Authorization languages

An active area of research in security centers around logical languages for expressing authorization policies. An authorization policy says, for example, which users can access which resources in which ways, e.g. Alice has permission to write myfile.txt. Such policies are often written by several individuals, each of whom may want to operate independently of the others. The security systems that enforce authorization policies require that every request be either allowed or denied. There is no way to simultaneously allow and deny a request, and there is no way to neither allow nor deny a request. Thus, while authorization policies are defined in collaborative settings, neither conflicting nor incomplete policies can be tolerated by security systems. Formally, an authorization policy maps requests R to either *allow* or *deny*[3].

$$R \rightarrow \{allow, deny\}$$

Despite the fact that an authorization policy is developed for a system that cannot tolerate conflicts or incompleteness, there is no reason to believe that the people collaboratively defining such a policy will disagree less or know more than people in another Collaborative Programming setting. Thus, an authorization language should be able to express conflicts and incompleteness, less people encode instructions they do not intend, yet at the same time should hide conflicts and incompleteness from the security system. Hiding conflicts and incompleteness means that the language should include mechanisms for resolving conflicts and incompleteness when they occur.

For conflicting authorization policies, where a request is both allowed and denied, there are at least two options for resolving that conflict. Deny might

[3] Depending on the setting, a request may contain a number of properties, e.g. the user, the resource, the action to be performed on the resource. For simplicity and generality, we treat a request as an opaque object.

take precedence over allow, or vice versa. It is important that the form of conflict resolution chosen is made known to users so that they can predict the results.

In FHL, it is natural to use a single distinguished predicate *allow*, and whenever an authorization request r is made, it is allowed if $allow(r)$ is entailed and denied if $\neg allow(r)$ is entailed. Conflicts amount to inconsistent theories where $allow(r)$ and $\neg allow(r)$ are both entailed. Conflict resolution is based on existential entailment as described in Section 3.

In DATALOG$^\neg$, a single predicate *allow* is insufficient for expressing conflicts. The language guarantees that if $allow(r)$ is entailed then $\neg allow(r)$ is not entailed. However, by using two distinguished predicates *allow* and *deny*, it is possible to encode conflicts and incompleteness. For any authorization request r, an authorization policy could entail $allow(r)$, $deny(r)$, both, or neither. Again, conflicts can be resolved by giving preference to either *allow* or *deny*.

For incomplete authorization policies, where a request is neither allowed nor denied, there are two separable cases. One form of incompleteness arises because the policy says nothing about a particular request. Similar to the case of conflict resolution, this form of incompleteness can be resolved by choosing either to allow or to deny the request, as the policy makes no commitment whatsoever. The other type of incompleteness, which is only possible in FHL-based languages, occurs when a request appears as a disjunctive consequence of the authorization policy. Resolving this type of incompleteness is more problematic than the first.

For example, consider an authorization policy with two FHL statements:

$$\begin{array}{l} allow(r_1) \vee allow(r_2) \\ \neg allow(r_1) \vee \neg allow(r_2). \end{array}$$

Together the statements say that either r_1 or r_2 must be allowed, and the other must be denied. Arguably, this policy is enforceable: simply make the choice. The problem is that the user may not be able to predict the result. It is imaginable that if the policy were written another way, the opposite choice might be made.

The resolution mechanism for disjunctive incompleteness requires making choices between requests, which is qualitatively different than making a choice between allow and deny. It is far easier to communicate a tie-breaking mechanism about allow and deny than about requests; moreover, it is unnatural to treat some requests differently than others when the authorization policy fails to do so. Thus, authorization languages should not be able to express disjunction.

While there are fragments of FHL that are guaranteed to be nondisjunctive, e.g. Horn clauses, DATALOG$^\neg$ has the benefit that it supports negation and limited recursion, which are difficult to support using nondisjunctive FHL. Thus, DATALOG$^\neg$ is the better choice for authorization languages.

Formalizing and implementing the conflict and incompleteness resolution mechanisms for a DATALOG$^\neg$-based language is straightforward. For example, if the conflict resolution mechanism deems that deny should override allow (a reasonable choice in the context of security), and policy completion allows all unspecified requests, the semantics for the authorization language ($\models'$) would

be defined as follows, where $\models$ is the usual DATALOG$^\neg$ semantics.

$$\Delta \models' deny(r) \text{ iff } \Delta \models deny(r)$$
$$\Delta \models' allow(r) \text{ iff } \Delta \not\models deny(r)$$

This layered approach to language design has two benefits. The core of the language ($\models$) is defined using traditional means and hence can leverage well-known tools. Those tools can be used to analyze a policy according to $\models$, identify conflicts, and inform the authors who contributed the conflicting statements; yet, at the same time, a security system can use $\models'$ to make authorization decisions using a policy without conflicts or incompleteness.

Thus, unlike FHL, which requires nonstandard automated reasoning tools for handling conflicts, conflict resolution in DATALOG$^\neg$ can be built on top of well-known techniques. This could explain the popularity of DATALOG$^\neg$ for authorization languages in the security literature [7, 2, 12, 17, 3, 13, 1, 6]. The drawback is that DATALOG$^\neg$ can only express conflicts in settings where all possible conflicts are known ahead of time. Keywords must be introduced into the language and built into the algorithms for processing that language.

5 Conclusion

Kowalski is famous for illustrating that logic can be used as a programming language, the result of which was the Logic Programming paradigm [10]. Today, the term "Logic Programming" has come to mean a particular type of logic and automated reasoning, syntactically based on implication and semantically concerned with negation as failure. Logic Programming today is consistent with Kowalski's original vision but is more narrowly defined than he intended.

Logic Programming (in Kowalski's original intent) is the right choice for industrial applications only in certain situations. The notion of Collaborative Programming was developed to explain to non-experts what those situations are and to reinvigorate Kowalski's original idea. Other similarly motivated work includes Golog [11], which includes nondeterministic choice operators, and Partial Programs [4], which enable programmers to express incomplete instruction sets.

Collaborative Programming differs from similar initiatives because of its commitment to conflicts. Because instruction sets are issued by multiple people, and people often disagree with one another, a Collaborative Programming language must allow conflicts to be expressed, simply so that the language is capable of capturing peoples' true intentions. Consequently, automated reasoning tools for processing instruction sets must be aware of and tolerate conflicts. In the case of classical logic, this requires automated reasoning tools that implement a paraconsistent entailment relation. In the case of logic programming languages, it requires making ontological commitments within the language and employing algorithms that adhere to those commitments. Each language class has strengths and weaknesses, making the right choice for any particular Collaborative Programming application dependent on the demands of that application.

References

1. Moritz Y. Becker, Cedric Y. Fournet, and Andrew D. Gordon. Design and semantics of a decentralized authorization language. In *Proceedings of the IEEE Computer Security Foundations Symposium*, pages 3–15, 2007.
2. Elisa Bertino, Barbara Catania, Elena Ferrari, and Paolo Perlasca. A logical framework for reasoning about access control models. *ACM Transactions on Information and System Security*, 6(1):71–127, 2003.
3. John DeTreville. Binder, a logic-based security language. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.
4. Michael R. Genesereth and J. Y. Hsu. Partial programs. In *Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning*, pages 238–249, 1991.
5. Timothy L. Hinrichs. *Extensional Reasoning*. PhD thesis, Stanford University, 2007.
6. Timothy L. Hinrichs, Natasha Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Design and implementation of a flow-based security language. Under review, 2008.
7. Sushil Jajodia, Pierangela Samarati, and V S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 31–42. IEEE Press, 1997.
8. Michael Kassoff and Michael R. Genesereth. PrediCalc: A logical spreadsheet management system. *Knowledge Engineering Review*, 22(3):281–295, 2007.
9. Michael Kassoff and Andre Valente. An introduction to logical spreadsheets. *Knowledge Engineering Review*, 22(3):213–219, 2007.
10. Robert Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.
11. Hector J. Levesque, Ray Reiter, Yves Lesperance, Fangzhen Lin, and Richard B. Scherl. Golog: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–83, 1997.
12. Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of the Symposium on Practical Aspects of Declarative Languages*, 2003.
13. Ninghui Li, John C. Mitchell, and William H. Winsborough. Design of a role-based trust-management framework. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2002.
14. G. Priest. Paraconsistent logic. In *Handbook of Philosophical Logic*, volume 6, pages 287–293. Kluwer Academic Publishers, 2002.
15. Jeffrey Ullman. *Principles of Database and Knowledge-Base Systems*. Computer Science Press, 1989.
16. Sergei Vorobyov and Andrei Voronkov. Complexity of nonrecursive logic programs with complex values. In *Proceedings of the ACM SIG for the Management of Data*, pages 244–253, 1998.
17. Marianne Winslett, Charles C. Zhang, and Piero A. Bonatti. Peeraccess: A logic for distributed authorization. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 168–179, 2005.