Implementing Extensible Theorem Provers

Kathi Fisler, Shriram Krishnamurthi, and Kathryn E. Gray

Department of Computer Science, Rice University
Houston, TX 77005-1892, USA
kfisler@cs.rice.edu
http://www.cs.rice.edu/~kfisler/

Abstract. The growing application of theorem proving techniques has increased the need for customized theorem provers. Powerful provers contain numerous interacting subsystems, each of which requires substantial time and expertise to build; constructing new provers from scratch is virtually prohibitive. Plug-and-play prover frameworks promise an alternative in which developers can construct provers by selecting logics, reasoning techniques, and interfaces. Realizing such frameworks cleanly requires specialized software architectures and particular language abstractions, even for frameworks supporting only simple interactions between logics. This paper explores architectural and linguistic issues in plug-and-play theorem prover development. It reflects our experience creating and using such a framework to develop several versions of a research prototype theorem prover.

Keywords: extensible theorem provers, plug-and-play theorem provers, software architectures, software components, programming languages

1 Introduction

Theorem provers are large bodies of software. A typical prover contains numerous interacting subsystems that implement decision procedures, reasoning methods, theory-declaration facilities, user interfaces, and more. As these subsystems mature, developing them demands greater expertise and labor. Simultaneously, the growing application of formal methods in general, and theorem proving in particular, is increasing demand for theorem provers. Extensions to existing provers can satisfy some of these applications; others require combinations of techniques from various provers. Still others may benefit from restricted versions of provers that, say, trade some functionality for a smaller memory footprint. Implementing each of these from scratch would be prohibitive. Instead, developers should be able to build such provers by wiring together components from existing provers. In this way, they can leverage off extant expertise and concentrate their efforts on value-adding customizations and extensions. Other researchers, developers, and users of theorem provers have also advocated a similar vision [16, 18, 29].

Designing and realizing such plug-and-play systems engenders many challenging technical problems. Some of these are strictly *logical*, such as how to semantically combine different logics, make multiple techniques inter-operate,

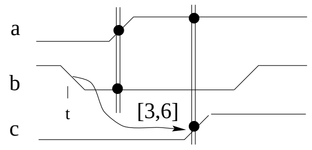


Fig. 1. Timing diagrams depict changes in boolean variables over time. Each change is called an *event*. Arrows indicate temporal ordering on events; optional annotations on arrows specify discrete bounds on the time passing between the two events. Vertical parallel lines specify event synchronization. The annotation on the falling transition on b (an anchor) indicates that the transition happens at time t. Fisler [11] defines a formal syntax and semantics, as well as inference rules, for timing diagrams.

or guarantee consistency after extensions. Others issues are *architectural*, such as how to integrate the code for existing subsystems. Architectural decisions greatly affect the maintainability and usability of a system. Thus, researchers must consider both logical and architectural features to build modular provers.

This paper discusses architectural issues in developing plug-and-play provers. The discussion goes beyond a standard high-level architectural description to include details about code organization within prover modules and the linguistic abstractions that support these organizations. Our observations and solutions arise from our experience in developing various plug-and-play systems, including an theorem prover framework called Ciderproof. Ciderproof lets developers

- build a new prover from selected logics and reasoning techniques,
- add new logics or reasoning techniques to an existing prover, and
- extend the expressive power of logics and techniques in an existing prover

without modifying or accessing the source code for existing prover modules.

Section 2 illustrates some forms of extensibility that arise in theorem provers. Section 3 presents our architecture for extensible provers. Section 4 discusses the role of linguistic constructs in extensible system implementation. Section 5 describes Ciderproof. The last two sections discuss related work and offer concluding remarks.

2 Extensible Provers: A Motivating Scenario

Consider the following scenario, which illustrates the vision of plug-and-play prover construction:

- Tom builds a natural deduction proof checker over timing diagrams (Figure 1) that have anchors but not timing constraints. He implements a simple interactive checker in which the user selects formulas and an inference rule to apply to them. He also implements a timing diagram data structure, a timing diagram editor, and functions corresponding to his inference rules. Tom distributes his tool on his web page as a set of libraries (Figure 3(a)).

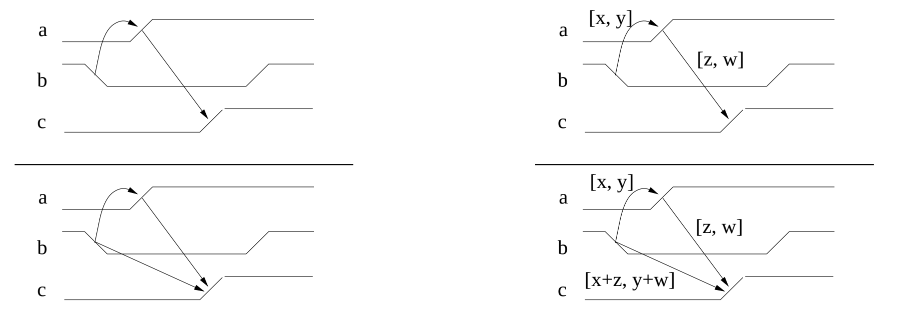


Fig. 2. How adding time bounds affects an existing inference rule. The original rule is on the left; the revised (extended) rule is on the right.

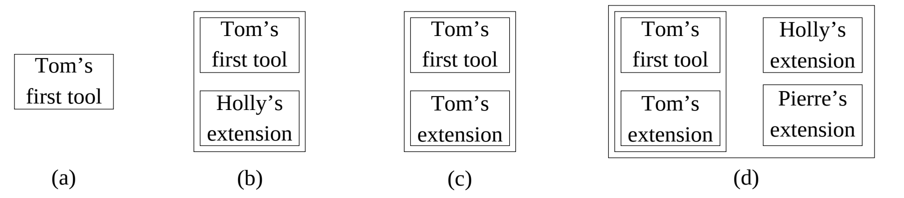


Fig. 3. The stages of the scenario. Each rectangle is an integrated set of libraries.

- Holly downloads Tom's tool, but her work needs both timing diagrams and first-order logic (henceforth FOL). She implements a FOL data structure and parser, as well as inference rules for FOL and rules that bridge timing diagrams and FOL. She integrates her libraries with Tom's, producing a new prover (Figure 3(b)).
- Tom enhances timing diagrams with timing constraints. He implements extensions to the data structure and editor. He also extends some of his original inference rules to handle the new syntax (as in Figure 2). Each extension reuses the functionality of his original system. He publishes a new composite library that integrates his original and new code (Figure 3(c)).
- Pierre wants a proof checker over timing diagrams and monadic second-order logic (henceforth MSOL). He downloads Tom's and Holly's code. He extends Holly's FOL data structure and parser with MSOL quantifiers. He also adds a substitution checker (henceforth SC) to help validate quantifier eliminations in his inference rules. Pierre integrates all four pieces of code to produce his tool (Figure 3(d)).

This scenario, comprised of extensions that we have performed with Ciderproof, motivates the benefits of plug-and-play provers. Holly and Pierre clearly saved substantial effort by enhancing existing code, rather than building their own systems from scratch. However, the scenario elides both how each person integrates their code with the others' and how each library must be structured for integration. This paper describes these protocols and requirements. We begin by motivating some criteria on the integration protocols.

– Developers can't rely on the ability to embed logics in one another or in a system-wide meta-logic; in this scenario, for example, the expressive power of the logic increases after each extension. Well-founded interactions between logics are possible, however, if every developer provides a formal syntax and semantics for the logic that he supports. The requirement ensures that our protocols apply to general classes of theorem provers.
– Developers can't modify the source code of existing libraries. If Pierre had to modify Holly's code in order to integrate it with his own, he would have to modify each new version that she produces to benefit from her changes. This is cumbersome. Pierre would rather program to an *interface* to Holly's code that she preserves across versions. This not only saves Pierre from managing multiple versions of Holly's libraries, but also prevents dependencies between his code and implementation details of Holly's code.
– Developers need not know which language a library has been implemented in. This allows developers to write each library in the language that is most appropriate for it (perhaps one for the GUI and another for the prover). This criterion is less idealistic than it seems if developers program to interfaces and do not modify existing source code.
– Each library should be separately compilable. Developers can therefore distribute object files instead of source libraries; this supports multi-language prover development and allows proprietary code in prover libraries.

Systems that adhere to these requirements allow developers to augment existing code without access to its source; such systems are called *extensible*. This paper discusses how to design extensible theorem provers for recursively defined logics. Starting from the scenario and the criteria on extensibility protocols, we derive an architecture for such systems. This architecture must go beyond a conventional, high-level description of how a system is organized; implementing extensible architectures without using cumbersome programming protocols also requires particular language constructs. Section 3 provides a high-level architectural description and Section 4 refines it with details about the language constructs that implement it.

3 The High-Level Architecture

The theorem provers in the scenario of Section 2 contain several types of information including representations, editors, inference rules, and tools (such as the proof engine and the SC). Multiple libraries may provide information of each type; for example, each stage in the scenario contributed inference rules. A prover engine, however, sees only one set of inference rules. The architecture therefore needs to provide a unified view of certain classes of information that may cross-cut several libraries. For now, we call these views collections; Section 4.4 describes collections in more detail.

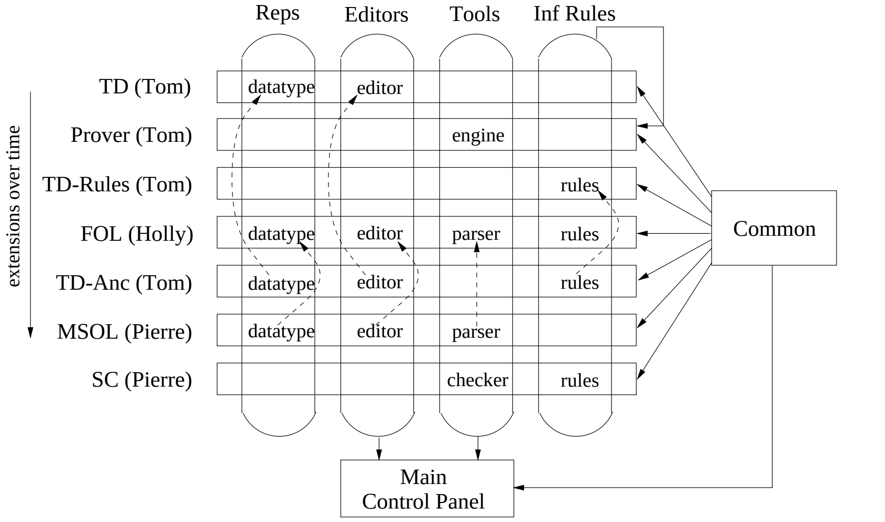
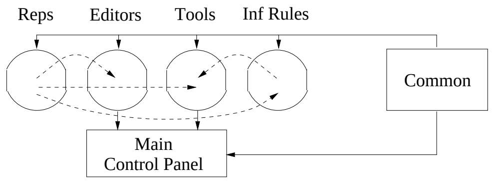


Fig. 4. The architecture at the end of the scenario in Section 2. Libraries at the target end of a solid arrow reference information from the library at the head of the arrow. Dashed arrows indicate that the source information extends the target information (unlike in the base architecture). In the annotations, TD stands for "timing diagram" and "Anc"stands for anchors.

The following diagram shows the base architecture. Ovals denote collections and rectangles denote code libraries. The base architecture contains two libraries: one for the main control panel and another for code that many parts of the system share. The solid arrows denote concrete information that flows between collections and libraries; for example, the control panel displays options to a user based on the installed sets of editors and tools. The dashed arrows denote information flows from one collection into some element of another: the prover, for example, needs a list of the available inference rules.



The base architecture provides an empty prover, with no logics or reasoning techniques. Integrating a library into a prover expands the collections and makes the concrete relationships between elements of the collections more explicit. Figure 4 shows the architecture at the end of the scenario in Section 2. Section 4 describes the protocols that transform the base architecture into Figure 4. The discussions refer to the several observations about the scenario in Figure 2:

1. Tom added new syntax to an existing timing diagram element.

2. Holly added inference rules to Tom's checker. Tom's checker had to update its list of available rules after the extension.
3. Tom couldn't predict the contexts in which other developers would use his code. As a result, his code had to be integratable into multiple contexts (his own extension in one case and Holly's in another). Furthermore, developers who use the code, not Tom, control where to integrate Tom's code.
4. In order to extend the timing diagram inference rules to handle timing constraints, Tom added functionality to his original inference rule functions.
5. Tom created a compound library from his original and extension libraries.
6. Pierre added an independent form (MSOL quantifiers) to the FOL syntax, and in parallel, extended the parser. We assume that both the syntax and the parser were defined recursively.

4 Language Constructs for Extensible Systems

The extensibility requirements affect the implementation of libraries, datatypes, tools, and databases. This section discusses these issues in turn, using the following terminology. A *datatype* specifies a new type with one or more *variants*. Each variant specifies a record-like structure with a fixed number of typed fields. A *client* of a library is any part of a program that references information from it. A program entity, such as a vector, is *first-class* if it can be treated as a value in all contexts (including being stored in data structures, passed as a parameter, and returned from a function). For example, functions are not first-class in Pascal because functions cannot return functions. Classes are not values at all, much less first-class, in most class-oriented languages, such as C++, Java, or Eiffel. Vectors are first-class in most languages. Closures in functional languages like ML and Haskell are also first-class, as are continuations in Scheme.

4.1 Libraries

Modules are a generally accepted mechanism for structuring code libraries. For his original tool, Tom would have written one module for the proof checker, one for the data structure, one for the editor, and one containing the inference rules. Connecting or *linking* these modules together forms his tool. Likewise, Holly and Pierre would have written modules for their extensions.

Module systems vary according to features such as type systems, linking styles, and support for nested modules. Sections 2 and 3 identified several desirable features in a module system for extensible provers: separate compilability, hierarchical linking (as in Pierre's prover, Figure 3(d)), the ability to use one module in multiple contexts, and client-controlled linking. A module with all of these characteristics is called a *component* [33].

Few existing module systems satisfy all the requirements to be components. Java packages do not because they hardwire their connections to other components [17]; clients therefore do not control linking in Java. Beta's module system is similar [25]. ML functors are components, but with a limitation: the type

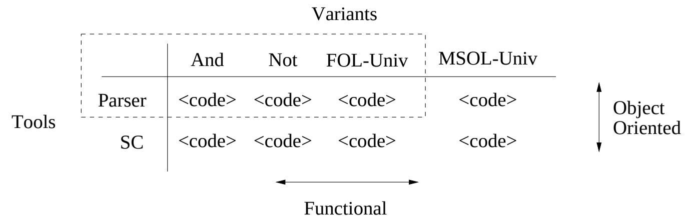


Fig. 5. Code fragments for two tools over a predicate logic datatype, and two common ways to organize those fragments into program structures.

system prohibits mutually-recursive dependencies between functors (*i.e.*, linking specifications must form trees instead of graphs) [26]. Such dependencies arise in practice. Consider a data structure component and a graphical editor component; the data structure component may use exception handlers from the editor to display messages to a user, while the editor invokes functions from the data structure component to create instances of the data structure. Object-oriented implementations can often encapsulate these dependencies in objects (*i.e.* use callbacks) to avoid circular graph-based linking. The MzScheme dialect [13] of Scheme provides components with full graph-based linking.

4.2 Parallel Datatype and Tool Extensions

Pierre's additions of MSOL quantifiers and the SC to Holly's code entailed adding a new variant to the FOL datatype and defining a new operation over the (resulting) MSOL datatype. Each operation over a datatype invokes some code fragment to process each variant of that datatype. The table in Figure 5 depicts the fragments for a FOL datatype with and, not, and universal quantification (FOL-Univ) variants. The dashed box encloses Holly's system; the MSOL universal quantifiers and the SC tool are Pierre's extensions. The protocols for extending datatypes and operations vary based on an implementation's organization of these code fragments. We now consider two common programming paradigms, how they would capture the dashed box in Figure 5, and how well they would support the extensions.

An Object-Oriented Attempt: A traditional object-oriented design would use an abstract base class for the FOL datatype; each variant of the datatype would be a separate subclass of this abstract base class. Each variant subclass would contain the parser code that processes that variant; this is known as the Interpreter pattern [14]. Thus, an object-oriented design would view the table in Figure 5 in terms of its columns. Under this organization, adding the MSOL quantifier variant is easy. We define a new subclass of the abstract base class that defines the parser code for MSOL expressions.

Adding the SC is more difficult. The obvious approach, extending each variant class with a subclass that contains its SC code, raises several problems:

1. Clients such as the inference rule and prover components reference the original classes, not the extended ones. We must update the clients to reference the new classes, but without modifying their source code (as per our extensibility criteria). This is where hardwired links between components becomes problematic. If the implementation language allows only hardwired links, a developer must use Factory patterns [14] or other cumbersome (and hence error-prone) protocols to circumvent them.
2. There is no place for code that the new SC classes should share, such as functions for managing the variable bindings. Since we cannot edit the code for the abstract base class (which is inside the dashed box), we must duplicate all shared code in each SC class extension. This violates good programming style and prevents a single point of control.
3. The super-classes for the SC class extensions will not be known until link-time. A developer might link the SC component with either the original FOL datatype, or an extension thereof. This becomes problematic if the super-classes of the SC classes must be fixed when the classes are defined (as most object-oriented languages require). Avoiding this problem requires either parameterized super-classes [10] or complex programming protocols [21, 27] that simulate them.

Thus, a naïve object-oriented organization handles datatype extensions naturally, but is clumsy for tool extensions. Fixing super-classes at class definition time, rather than link-time, is the main problem in this approach.

A Functional Attempt: A traditional functional design would define a datatype for FOL and a function for the parser. Thus, it would view the table in Figure 5 in terms of its rows. Under this organization, adding the SC entails simply writing a new function. Code to be shared among the fragments takes the form of local definitions within the function; this solves one of the problems from the object-oriented approach.

Performing the MSOL extension, however, is problematic. First, statically-typed functional languages have limited support for extensible datatypes. Due to their existing type systems, the extensible datatype mechanisms for ML [9] and Haskell [24], for example, essentially seal off datatypes at compile-time: they cannot handle *dynamic* extensions. This is less problematic in object-oriented languages, which often give the effect of extensible datatypes through dynamic subclassing. Dynamic extensions are a nice feature because they enable extensions to a running system. If a user needs an additional proof technique in the middle of a proof, loading an extension dynamically saves her from having to recompile and resume execution (thereby losing her current work). However, we view this as a small matter of convenience, and therefore less fundamental than the other issues involved in developing extensible systems.

Another problem of a more subtle nature arises in the case of recursively defined logics. Consider Holly's original parser. Since the logic is defined recursively, the parser is likely recursive as well. The MSOL extension defines a parser for the new variant that invokes (reuses) the original parser on the original variants. If we attempt to parse the expression $\forall y\, \forall P.P(y)$, the new parser calls the

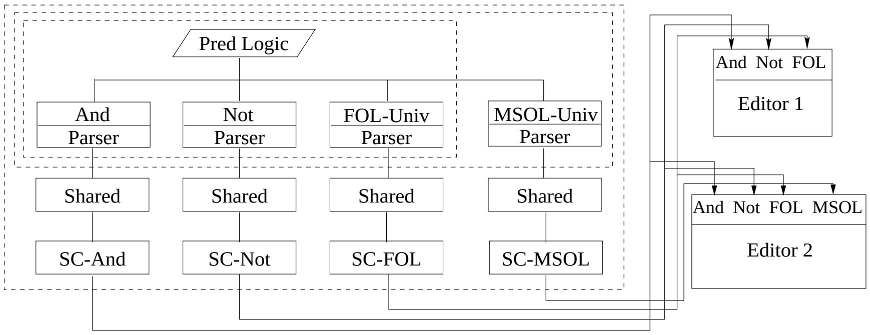


Fig. 6. Using mixins and units to extend the FOL datatype and add the SC in parallel. "Shared" is a mixin that extends several classes. The dashed boxes enclose Holly's original FOL datatype component and Pierre's respective extensions. The arrows link the datatype components with editor components.

original parser to process the outermost quantifier. The original parser processes $\forall P.P(y)$ recursively. But the recursive call invokes the original, not the extended, parser. As $\forall P.P(y)$ is in MSOL, which the original parser does not handle, a parsing error occurs. Again, the problem here is one of hardwiring: the recursive calls inside the original parser refer to the original parser [7, 20]. Object-oriented solutions based on the Interpreter pattern can suffer from similar problems when the operation creates new instances of a recursively defined datatype.

A Synthetic Solution: These problems with the object-oriented and functional styles, noted by several researchers [8, 21, 22, 27, 30], illustrate how hardwired relationships between program entities hinders extensibility. Ideally, an implementation framework for extensible systems should allow flexible connections between classes, between components, and between related code fragments; the underlying principle is to separate definitions from their connections [12]. Components provide this feature by definition. For classes, we need *mixins*, which are reusable class extensions with parameterized super-classes. In a language with mixins, a *class* is a composition of mixins onto an existing class (the language must provide an empty base class). Existing languages offer varied support for mixins. The idea appears to have arisen initially in Common Lisp [32]. VanHilst and Notkin [34] simulate mixins in C++ using templates, but, as the authors point out, their approach does not scale due to the amount of code duplication it entails. Java does not support mixins because it requires hardwired super-classes. OCaml [23] has "functorizable" classes, but the mixins obtained through this mechanism are less flexible than the notion used in this paper. Specifically, the type system demands that the super-class provided to a functor must provide *only* the services listed in the functor's import signature, which makes such mixins considerably less reusable. Beta and MzScheme both support mixins.

Using components and mixins, we can avoid the problems that arise in the functional and object-oriented solutions; this demonstration is based on Findler

```
(define SC-extension
  (unit (import curr-And curr-Not curr-FOL curr-MSOL)
    (export And Not FOL-Univ MSOL-Univ)
    (define Shared (lambda (parent) (class parent ...)))
    (define SC-extend
      (lambda (curr-class SC-class)
        (SC-class (Shared curr-class))))
    (define SC-And (lambda (parent) (class parent ...)))
    (define SC-Not (lambda (parent) (class parent ...)))
    (define SC-FOL (lambda (parent) (class parent ...)))
    (define SC-MSOL (lambda (parent) (class parent ...)))
    (define And (SC-extend curr-And SC-And))
    (define Not (SC-extend curr-Not SC-Not))
    (define FOL-Univ (SC-extend curr-FOL SC-FOL))
    (define MSOL-Univ (SC-extend curr-MSOL SC-MSOL))))

(define make-PredLogic-SC
  (lambda (current-PredLogic-unit)
    (compound-unit (import)
     (link
      (PL (current-PredLogic-unit))
      (SC (SC-extension (PL And) (PL Not) (PL FOL-Univ) (PL MSOL-Univ))))
     (export (SC And) (SC Not) (SC FOL-Univ) (SC MSOL-Univ)))))
```

Fig. 7. Implementing the SC extension with first-class components and mixins.

and Flatt's [10] approach in MzScheme (where components are called *units*). Figure 6 shows the MSOL and SC extensions in terms of mixins and components. Each dashed box defines a component. The class hierarchy in the innermost dashed box implements Holly's original FOL datatype. The trapezoid labeled *PredLogic* denotes the abstract base class for the datatype; each original variant is a mixin that extends that base class. The middle dashed box shows the datatype after the extension for MSOL quantifiers. At extension time, the *MSOL-Univ* mixin extends the *PredLogic* base class.

The outermost dashed box shows the datatype after the SC extension; Figure 7 shows this extension as written in MzScheme. Component *SC-extension* imports classes for the four variants of the *PredLogic* datatype and exports extended classes for these variants. This component contains five mixins: one for the code to be shared across the variants (*Shared*), and one containing the SC code for each variant (*SC-And*, *SC-Not*, *SC-FOL*, and *SC-MSOL*). The extension occurs when a developer links the *SC-extension* component with the component defined by the middle dashed box in Figure 6. At that time, *SC-extension* uses function *SC-extend* to compose each SC variant mixin with the Shared mixin and the imported mixin for that variant; this yields the extended classes.

Figure 7 also illustrates the protocol that performs this extension. The function *make-PredLogic-SC* takes a component that exports variants for *And*, *Not*, *FOL-Univ*, and *MSOL-Univ* and links (compounds) it with *SC-extension*.[1] The resulting component has the same interface as the original, but its exported classes contain the SC extensions. By taking the current component as a parameter, *make-PredLogic-SC* gives developers control of where to link *SC-extension* into a system. This protocol, which relies on first-class components, is simple to use and shares the spirit of composition from functional languages.

Functions analogous to *make-PredLogic-SC* combine datatype components with other components in a prover. Figure 6 shows two client editor components. Editor 1 imports classes for the `and`, `not`, and FOL universal quantifier variants. Editor 1 can link to any of the three predicate logic components, because each provides versions of these classes. Editor 2 can link to any predicate logic component that also provides a MSOL quantifier class. In a language with first-class components, changing which components to link can be as simple as passing different arguments to a function.

This solution addresses all of the problems that arose in the standard object-oriented and functional approaches. Class hierarchies provide extensible datatypes. Using mixins for shared code avoids source code duplication. Invoking extension functions on revised components updates clients without modifying their source code. Putting the code for recursively defined tools (such as the parser) into the variant sub-classes solves the problem of hardwired recursive calls in the extended parser.[2] Thus, an approach based on components and mixins, which combines the object-oriented and functional programming styles, provides better support for extensibility than either style alone. For implementations in languages without units and mixins, abstract factories [14] or Smaragdakis and Batory's mixin layers [31] provide alternative approaches that require more complicated protocols and sacrifice some benefits such as separate compilation.

4.3 Extending Functions

When Tom added time bounds to successor edges in the scenario, he also had to extend the inference rules that operate on successor edges. Each rule corresponds to a function that checks whether an application of the rule is valid. Thus, extending inference rules requires extensions to functions. This kind of function extension differs from the parser extension discussed in Section 4.2. The parser extension involved only a single, new variant of the datatype. In this case, we must integrate new functionality into a function over an existing variant.

Function extensions require first-class functions and protocols for composing them [7, 20]. The original function must provide a parameter for an extension function. The passed function will process any extensions to the datatypes on

[1] Optional annotations on units specify and enforce names and types on imported and exported information; in order to simplify the code, we do not show them here.

[2] Extending functions that create recursively defined data requires a slightly more complicated solution.

```
(define check-rule
  (lambda (edge-1 edge-2 new-edge hash-table extra-check)
    (and (equal? (target-event edge-1) (source-event edge-2))
         (ht-equal? (source-event edge-1) (source-event new-edge) hash-table)
         (ht-equal? (target-event edge-2) (target-event new-edge) hash-table)
         (extra-check edge-1 edge-2 new-edge hash-table))))

(define extended-check
  (lambda (edge-1 edge-2 new-edge hash-table extra-check)
    (check-rule edge-1 edge-2 new-edge hash-table
                (lambda (e1 e2 newe ht)
                  (and (= (lower-bound newe)
                          (+ (lower-bound e1) (lower-bound e2)))
                       (= (upper-bound newe)
                          (+ (upper-bound e1) (upper-bound e2)))
                       (extra-check e1 e2 newe ht))))))
```

Fig. 8. Extending an inference rule checker function after adding time bounds.

which the function operates. The original function must therefore invoke the passed function at an appropriate point in its computation to process any additional variants or attributes. A simplified example based on the transitivity rule over timing diagrams appears in Figure 8; *check-rule* is the original inference rule and *extended-check* is the extension.[3] Both functions have the same types; they take the two original edges, the (asserted) transitive edge, a hash table that correlates information between the two timing diagrams, and an extension function, and return booleans. Passing *extended-check* as a parameter to *check-rule* allows us to pass the hash table to *extended-check*. Had we simply composed the extension and the original function, either the extension would have to recompute the hash table, or the original function would have to return the hash table; unless the implementation language provides native support for multiple return values, this would require an unwieldly protocol involving multiple values. In order to allow additional extensions to the rule, *extended-check* also needs a parameter for an extension function. The extension protocol consists of writing the function *extended-check*.

4.4 Collections Revisited

The architecture in Section 3 contains collections of related information corresponding to datatypes, editors, inference rules, and tools. We perform several operations on these collections in the course of extending the system: adding new items to them, extending particular items (such as inference rules), and asking for all items of a particular type (as the prover does for inference rules). From

[3] The actual function operates over the full timing diagrams, not just the edges.

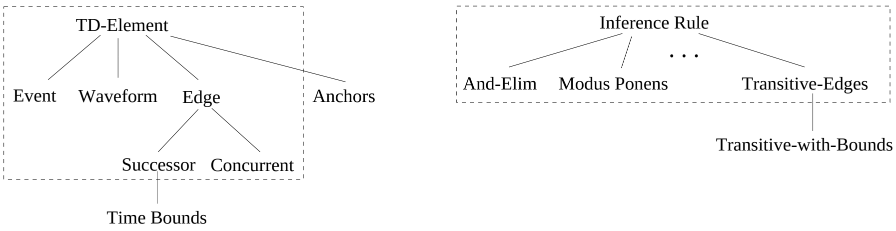


Fig. 9. The database trees for the timing diagram datatype and the inference rules.

this description, these collections resemble databases that we query and update at each extension to the system.

Our databases have two interesting features. First, they cross-cut components. Many extension components provide inference rules, yet the database lets us view them as a unified data structure. Second, databases act as a version control system. When the prover queries the rules database for the available rules, the database must return the most extended versions. Models for handling cross-cutting and version control are therefore candidates for implementing databases. We currently view a database as a tree of definitions; Figure 9 shows trees for the timing diagram representation and the inference rules. These trees resemble class hierarchies, but they may contain items other than classes (such as inference rule functions). The tree for the timing diagram elements has a branch for each variant, while the inference rule tree has a branch for each rule. The required operations on databases, such as adding items, finding elements, and getting lists of elements, are simply operations on trees. Each database provides a traversal method for searching and obtaining its leaves. Storing classes in databases, such as those for the timing diagram datatype, therefore requires first-class classes.

4.5 Summary and Perspective

Our discussions identified several language constructs that naturally express extensible theorem prover frameworks:

- Extensible datatypes
- First-class functions
- First-class classes
- First-class components
- Mixins (externally connected classes)
- Components (externally linked modules)
- Databases
- Compoundable components

Automatic memory management is also critical in extensible frameworks. The dataflows in an extensible system are quite complicated, and hence difficult to manage manually. Even worse, extensions may break existing code by altering when data may be released [35]. Furthermore, our experience in attempting to combine program fragments that use different memory management strategies indicates that this is not feasible in practice. Using automatic memory management avoids these issues and also reduces programming overhead.

Extensible systems are, in the end, the products of careful design, not just the choice of an appropriate programming language. As the second author's prior work [20, 21] has shown, object-oriented and functional design are strongly related, and this relationship is manifest in their strengths and weaknesses at handling extensions. This work also showed that functional design more effectively synthesizes these strengths, and that the Visitor pattern [14] is really a manual, object-oriented encoding of functional programming [21].[4] The main remaining benefit we get from object-oriented languages is their support for dynamic type extensions. (Functional programming in uni-typed languages, like core Scheme and Erlang [4], also offers this benefit.) In this paper, we have attempted to describe our design in abstract terms, and then discuss how the design elements map to various concrete programming languages, identifying areas where the mappings break down in certain languages.

It is nevertheless much easier to program in languages that directly support the linguistic elements used to describe the design. Languages that have both functional and object-oriented constructs, such as Beta, OCaml, and MzScheme, are good candidates for extensible system development. Beta's main limitation appears to be the module system, which hinders extensibility through hardwired connections. OCaml's main limitation lies with mixins, as discussed in Section 4. MzScheme provides adequate support for the features we have needed to implement extensible systems in practice.

5 Ciderproof

Using the techniques described in this paper, we have built an extensible theorem prover called Ciderproof (see Figure 10). Ciderproof supports a logic of hardware design representations, including circuit diagrams, state machines, timing diagrams, linear temporal logic, and monadic second-order logic [11]. Such logics, which support multiple syntactic representations (without embedding them in one another) are called *heterogeneous* [5] or *multi-language* [15]. They are ideal for problem domains, such as hardware design, in which people use multiple notations on individual problems. Since the notations that people use when reasoning about particular domains can evolve over time, reasoning tools for heterogeneous logics need to be extensible in order to evolve accordingly.

Ciderproof currently supports additions and extensions to representations, editors, inference rules, and reasoning tools. Developers could add databases of and extensions to other kinds of information. We have performed every extension demonstrated in this paper in Ciderproof, using similar protocols to those described in Section4. We have implemented Ciderproof in MzScheme, mainly because it supports first-class mixins and units [13]. Ciderproof is a real research tool, currently containing roughly 12,000 lines of MzScheme code. Using the techniques described in the paper, a team of two undergraduates and a part-time programmer implemented a prototype system in only three months. During that

[4] That paper also discusses the type problems that arise when implementing these solutions in a language without generic types (like Java).

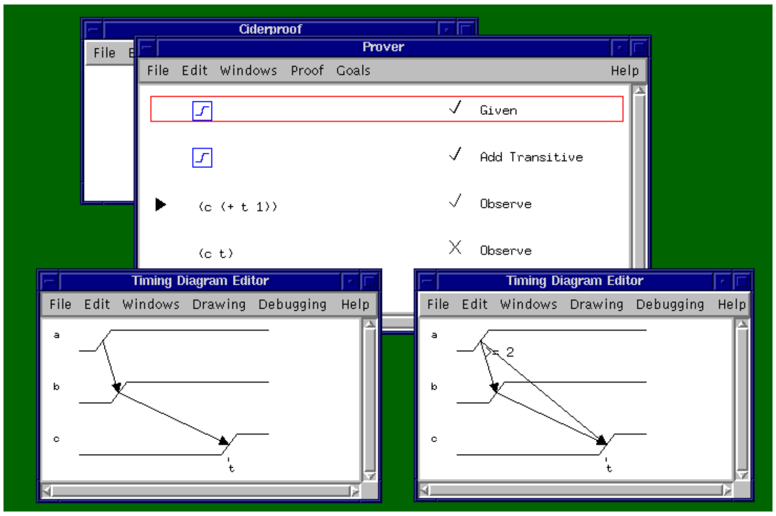


Fig. 10. A view of Ciderproof with MSOL and timing diagram representations.

time, we built several versions of Ciderproof by composing various components as described in this paper. Based on this experience, we believe our architecture and protocols are robust and provide a viable base for extensible prover design.

The main limitation of our system is the same as for any plug-and-play prover framework: interactions between subsystems must be expressible as interfaces. Boyer and Moore's integration of a decision procedure into an existing prover shows that this is not trivial [6]. Whether we could develop some of the integrated reasoning tools in the literature [1] in our framework depends on whether we could express them in this manner. This remains a problem for future work.

6 Related Work

Plug-and-play provers are a popular research topic. The Open Mechanized Reasoning Systems (OMRS) project members have written several papers on logical issues in extensible theorem prover development: Giunchiglia, Pecchiari and Talcott provide a general architecture for the logical components of such systems [16], while Armando and Ranise present a methodology for lifting specialized reasoning tools out of existing provers [3]. The architectures discussed throughout the OMRS project address logical issues, but neither implementation architectures nor aspects of the protocols needed to realize their systems. PROSPER [29], a joint project of several theorem proving research groups, also concerns how to design open prover frameworks, though we are not aware of any published documents on their architecture.

Gravell and Pratten propose a Java-based architecture for open and extensible theorem provers [18, 19]. They addresses syntactic extensions to recursively-defined languages, as well as some logical issues. Their system handles syntactic extensions using a combination of Java packages and the Abstract Factory pattern. We have already discussed how hardwiring in Java hinders extensibility. Pattern-based solutions expect programmers to manually maintain complex invariants and intertwine functionality with scaffolding from the pattern. Furthermore, such solutions often have to subvert the Java type system [21]. Gravell and Pratten also acknowledge a limitation in Java, namely that they "would like to separate the identification of the subclass relationship [...] from the definition of the class" ([19], page 14). Mixins provide exactly this ability.

Openproof [2], a multi-syntactic and diagrammatic reasoning tool supporting Venn diagrams, Hasse diagrams, position diagrams, and blocks-world diagrams, is similar in spirit to Ciderproof. It uses JavaBeans to support dynamic addition of new representations, but it does not support extensions to existing representations. While commercial component frameworks, such as JavaBeans, COM, and CORBA, support dynamic linking, they do not address issues such as function extensions and parallel extensions to recursively defined datatypes and tools. The resulting protocols for these operations therefore require the programmer to maintain more complicated invariants than in frameworks that provide native constructs for expressing extensions.

Isabelle [28] is a widely-used theorem prover development framework. Developers can embed (at shallow or deep levels) object logics into its core meta-logic; the core prover engine then operates over the new object logics. Isabelle and our framework have different design goals. Isabelle has been crafted primarily to support logical extensibility. Our work emphasizes software engineering aspects of component-based theorem prover design. Isabelle appears to support customizations to monolithic, rather than component-based, provers.

7 Conclusions

Plug-and-play theorem provers are a challenging, exciting, and extremely practical research direction for the theorem proving community. Implementing these provers requires research into both logic (*e.g.* interoperability) and engineering (*e.g.* how to add features to a logic and its tools with minimal or no source code modification). The complexity of these issues and their potential interactions, coupled with the recursive structure of many of our logics, makes theorem provers a potent case study for extensible systems.

This paper proposes an architecture for extensible prover frameworks and explores the language constructs that naturally express its features. We have intentionally designed our architecture for systems with minimal logical features but sophisticated kinds of extensibility. Minimizing the logical features allows us to identify fundamental architectural issues for plug-and-play prover design. In the long term, we plan to study how we must refine this architecture and its protocols to handle more complicated logical interactions (including embeddings), as well as different kinds of logics. In general, however, our experience shows that such extensible frameworks are naturally expressed in languages that support a combination of declarative programming styles and allow flexible connections between program entities, such as components, classes, and even pieces of functions. These observations have significant impact on how we select or design programming languages for implementing extensible provers.

We have used our framework to implement an extensible theorem prover called Ciderproof. Ciderproof allows users to add syntactic representations, extend existing representations in expressive ways, and add reasoning tools that operate on those representations. Developers repeatedly compose separately compilable prover components to create new instances of Ciderproof. The composition protocols involve no source code modifications or cumbersome programmer-maintained invariants. Our team has built several versions of Ciderproof in this manner. We are confident, based on our experience building these and other extensible systems, that our architecture is both viable and robust.

References

1. Aagaard, M. D., R. B. Jones and C.-J. H. Seger. Combining theorem proving and trajectory evaluation in an industrial environment. In *Proceedings of the 35th Design Automation Conference*, 1998.
2. Allwein, G. Private commmunication, 1998.
3. Armando, A. and S. Ranise. From integrated reasoning specialists to "Plug-and-Play" reasoning components. Technical Report MRG/DIST 97-0049, Dipartimento Informatica Sistemistica Telematica, Università di Genova, November 1997.
4. Armstrong, J., R. Virding, C. Wikström and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, 1996.
5. Barwise, J. and J. Etchemendy. Heterogeneous logic. In Glasgow, J., N. H. Narayanan and B. Chandrasekaran, editors, *Diagrammatic Reasoning: Cognitive and Computational Perspectives*, pages 211–234. MIT Press, 1995.
6. Boyer, R. S. and J. S. Moore. Integrating decision procedures into heuristic theorem provers: A case study with linear arithmetic. *Machine Intelligence*, 11, 1988.
7. Cartwright, R. S. and M. Felleisen. Extensible denotational language specifications. In Hagiya, M. and J. C. Mitchell, editors, *Symposium on Theoretical Aspects of Computer Software*, pages 244–272. Springer-Verlag, April 1994. LNCS 789.
8. Cook, W. R. Object-oriented programming versus abstract data types. In *Foundations of Object-Oriented Languages*, pages 151–178, June 1990.
9. Duggan, D. and C. Sourelis. Mixin modules. In *International Conference on Functional Programming*, pages 262–273, May 1996.
10. Findler, R. B. and M. Flatt. Modular object-oriented programming with units and mixins. In *Proceedings of the International Conference on Functional Programming*, 1998.
11. Fisler, K. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams*. PhD thesis, Indiana University, 1996.
12. Flatt, M. *Programming Languages for Reusable Software Components*. PhD thesis, Rice University, Department of Computer Science, 1999.
13. Flatt, M. and R. B. Findler. PLT MrEd: Graphical toolbox manual. Technical Report TR97-279, Rice University, 1997.
14. Gamma, E., R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Personal Computing Series. Addison-Wesley, Reading, MA, 1995.

15. Giunchiglia, F. Multilanguage systems. In *Proceedings of the AAAI-91 Spring Symposium on Logical Formalizations of Commonsense Reasoning*, 1991.
16. Giunchiglia, F., P. Pecchiari and C. Talcott. Reasoning theories: Towards an architecture for open mechanized reasoning systems. Technical Report STAN-CS-TN-94-15, Stanford University Department of Computer Science, September 1994.
17. Gosling, J., B. Joy and G. L. Steele, Jr. *The Java Language Specification*. Addison-Wesley, 1996.
18. Gravell, A. M. and C. H. Pratten. A prototype generic tool supporting the embedding of formal notations. In Grundy, J. and M. Newey, editors, *Theorem Proving in Higher Order Logics: Emerging Trends*, pages 63–72. Australian National University, 1998.
19. Gravell, A. M. and C. H. Pratten. Using Java to build an open and extensible theorem prover component. University of Southamption, 1998.
20. Krishnamurthi, S. and M. Felleisen. Toward a formal theory of extensible software. In *ACM Symposium on the Foundations of Software Engineering*, 1998.
21. Krishnamurthi, S., M. Felleisen and D. P. Friedman. Synthesizing object-oriented and functional design to promote re-use. In *European Conference on Object-Oriented Programming*, pages 91–113, July 1998.
22. Kühne, T. The translator pattern—external functionality with homomorphic mappings. In *Proceedings of TOOLS 23, USA*, pages 48–62, July 1997.
23. Leroy, X. *The Objective Caml system, documentation and user's guide*, 1997.
24. Liang, S., P. Hudak and M. Jones. Monad transformers and modular interpreters. In *Symposium on Principles of Programming Languages*, pages 333–343, 1992.
25. Madsen, O. L., B. Møller-Pedersen and K. Nygaard. *Object-oriented programming in the BETA programming language*. Addison-Wesley, 1993.
26. Milner, R., M. Tofte and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.
27. Palsberg, J. and C. B. Jay. The essence of the Visitor pattern. Technical Report 05, University of Technology, Sydney, 1997.
28. Paulson, L. C. *Isabelle : a generic theorem prover*. Number 828 in Lecture Notes in Computer Science. Springer-Verlag, Berlin; New York, 1994.
29. PROSPER: Proof and specification assisted design environments. Project Programme, available at `http://www.dcs.gla.ac.uk/prosper/`, March 1998.
30. Reynolds, J. C. User-defined types and procedural data structures as complementary approaches to data abstraction. In Schuman, S. A., editor, *New Directions in Algorithmic Languages*, pages 157–168. IFIP Working Group 2.1 on Algol, 1975.
31. Smaragdakis, Y. and D. Batory. Implementing layered designs with mixin layers. In *Proceedings of the 12th European Conference on Object-Oriented Programming*, pages 550–570, 1998.
32. Steele, G. L., Jr., editor. *Common Lisp: the Language*. Digital Press, Bedford, MA, second edition, 1990.
33. Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley/ACM Press, 1998.
34. VanHilst, M. and D. Notkin. Using C++ templates to implement role-based designs. In *Second JSSST International Symposium on Object Technologies for Advanced Software*, pages 22–37. Springer-Verlag, 1996.
35. Wilson, P. R. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*. Springer-Verlag, September 1992.