Modular Redundancy for Theorem Proving

Miquel Bofill[†], Guillem Godoy[‡], Robert Nieuwenhuis[‡] and Albert Rubio[‡] *

[†]Universitat de Girona, Dept. IMA,
Lluís Santaló s/n, 17071 Girona, Spain
`mbofill@ima.udg.es`

[‡]Technical University of Catalonia, Dept. LSI,
Jordi Girona 1, 08034 Barcelona, Spain
`{ ggodoy,roberto,rubio }@lsi.upc.es`

Abstract. We introduce a notion of modular redundancy for theorem proving. It can be used to exploit redundancy elimination techniques (like tautology elimination, subsumption, demodulation or other more refined methods) in combination with arbitrary existing theorem provers, in a refutation complete way, even if these provers are not (or not known to be) complete in combination with the redundancy techniques when applied in the usual sense.

1 Introduction

The concept of *saturation* in theorem proving is nowadays a well-known, widely recognized useful concept. The main idea of saturation is that a theorem proving procedure does not need to compute the *closure* of a set of formulae w.r.t. a given inference system, but only the closure *up to redundancy*. Examples of early notions of redundancy (in the context of resolution) are the elimination of tautologies and subsumption. Bachmair and Ganzinger gave more general abstract notions of redundancy for inferences and formulae (see, e.g., [BG94]), where, roughly, something is redundant if it is a logical consequence of smaller logical formulae (w.r.t. a given ordering on formulae). In provers based on ordered resolution, paramodulation or superposition w.r.t. a total reduction ordering on ground terms $>$, apart from standard methods like removing tautologies, subsumption, or demodulation w.r.t. $>$, these abstract redundancy notions cover a large number of other practical techniques [NN93,GNN98].

However, in other contexts refutation completeness is lost if such redundancy elimination methods are applied. For example, the elimination of certain classes of tautologies is incompatible with logic programming with equality and arbitrary selection rules [Lyn97] and it is also incompatible with *strict superposition* [BG98]. Another situation where even less redundancy can be used is in the context of paramodulation with non-monotonic orderings [BGNR99], which is

* All authors partially supported by the ESPRIT Basic Research Action CCL-II, ref. WG # 22457 and the CICYT project HEMOSS ref. TIC98-0949-C02-01.

needed when monotonic orderings are not available, like in Knuth-Bendix completion of arbitrary terminating term rewrite systems, or in deduction modulo certain equational theories E for which no monotonic *E-compatible* orderings exist. Apart from these situations where it is known that the standard redundancy techniques cannot be used, there is also a large number of other cases where the compatibility with redundancy is unknown, or where for the completeness proof techniques applied it is unclear how to extend them to include redundancy, or where the theorem proving method is simply not based on the closure under a set of inference rules.

For all these cases our aim is to provide here modular notions of redundancy. Assume we have an unsatisfiable initial set of formulae S_0, and we want to prove its unsatisfiability by some refutation complete prover $\mathcal{P}$ that is a "black box", i.e., apart from its refutation completeness nothing is known about it. In particular, nothing is known about its compatibility w.r.t. redundancy elimination methods. In principle, the prover $\mathcal{P}$ may be based on any technique, but in our context it will be useful that $\mathcal{P}$ soundly generates new formulae that are consequences from its input (this happens, for instance, if $\mathcal{P}$ is based on a sound inference system).

Now the problem we address is: how, and up to which degree, can we still exploit redundancy elimination methods in this general context?

One instance of our ideas roughly amounts to the following. Apart from the prover $\mathcal{P}$, there is a *redundancy module* $\mathcal{R}$ that works with a given arbitrary well-founded ordering $>$ on formulae (this ordering $>$ may be completely different from the one $\mathcal{P}$ uses, if $\mathcal{P}$ is based on some ordered inference system). The redundancy module $\mathcal{R}$ keeps track of a set of formulae U that is assumed to be unsatisfiable. Initially, U is S_0. Each time a new formula is generated by $\mathcal{P}$, this is communicated to $\mathcal{R}$, who uses this information to try to prove the redundancy of some formula F of U, i.e., it checks whether F follows from smaller (w.r.t. $>$) formulae $G_1, \ldots, G_n$ that are in U or have been generated by $\mathcal{P}$ (or are logical consequences of these). Each time this is the case, a new set U is obtained by replacing F by $G_1, \ldots, G_n$. This fact is then communicated to $\mathcal{P}$, whose aim becomes to refute the new U instead of the previous one. In a naive setting, $\mathcal{P}$ could simply restart with the new U, but of course one can do better. Instead, $\mathcal{P}$ can apply several techniques (that may already be implemented in $\mathcal{P}$) in order to exploit the new situation. For example, $\mathcal{P}$ can apply *orphan murder* techniques: the descendants of F that have already been computed by $\mathcal{P}$ can be eliminated (although some of them can be kept if we want to completely avoid the repeated computation of formulae; for this, some additional bookkeeping is required). In this procedure, U is a finite set of formulae that decreases w.r.t. the well-founded multiset extension $\gg$ of $>$ each time it is modified. Hence at some point of the procedure U will not be modified any more and its limit U_∞ is obtained. If $\mathcal{P}$ computes all inferences from U_∞, then the refutation completeness of the general procedure follows.

Our techniques have been conceived with first-order clausal logic in mind. However, note that we do not rely on any particular logic.

This paper is structured as follows. In Section 2 we give a formal description of prover modules, redundancy modules, their combination and some first results. In Section 3 we explain how a redundancy module with the right properties can be built. Section 4 explains for three cases how to build a prover module in terms of an existing prover. First, this is done for an (almost) arbitrary prover, in a rather naive way where not much power of the redundancy module is exploited. Second, it is done for a prover based on the closure with respect to a complete inference system. And third, it is done for saturation-based provers (which is the case in most applications, like the ones of [Lyn97], [BG98], and [BGNR99]). Finally in Section 5 we give some conclusions.

2 Modular Redundancy

We now give a formal description of prover modules, redundancy modules, their combination and some first results.

Definition 1. *A* prover module *is a procedure satisfying:*

- *it takes as input a set S_0 of formulae*
- *at any point of its execution, it may receive messages of the form "replace F by $G_1, \ldots, G_n$", where F and $G_1, \ldots, G_n$ are formulae*
- *at any point of its execution, it may produce messages of the form "F has been deduced", where F is a formula*
- *it may halt with output "unsatisfiable".*

Definition 2. *Let $\mathcal{P}$ be a prover module, and consider a given execution of $\mathcal{P}$. We denote by S_0 the input set of formulae, and for each $k > 0$ by S_k we denote the set $S_k = S_{k-1} \cup \{G_1, \ldots, G_n\}$, where "replace F by $G_1, \ldots, G_n$" is the k-th message it receives.*

Furthermore, we denote by U_0 the input set of formulae, and for each $k > 0$ by U_k we denote the set $U_k = (U_{k-1} \setminus \{F\}) \cup \{G_1, \ldots, G_n\}$, where "replace F by $G_1, \ldots, G_n$" is the k-th message it receives.

Definition 3. *A prover module $\mathcal{P}$ is* correct *w.r.t. the given logic if:*

- *$S_k \models F$ whenever $\mathcal{P}$ emits an output message "F has been deduced" after receiving its k-th message.*
- *if $\mathcal{P}$ halts with output "unsatisfiable" after receiving its k-th message then S_k is unsatisfiable.*

Definition 4. *A prover module $\mathcal{P}$ is* refutation complete *if, for every execution where it receives exactly k messages, it halts with output "unsatisfiable" whenever U_k is unsatisfiable.*

Definition 5. *A* redundancy module *is a procedure that satisfies:*

- *it takes as input a set of formulae*

- *at any point of its execution, it may produce messages of the form "replace F by $G_1, \ldots, G_n$", where F and $G_1, \ldots G_n$ are formulae*
- *at any point of its execution, it may receive messages of the form "F has been deduced", where F is a formula.*

Definition 6. *Let $\mathcal{R}$ be a redundancy module, and consider a given execution of $\mathcal{R}$. We denote by R_0 the input set of formulae, and for each $k \geq 0$ by R_k we denote the set $R_k = R_{k-1} \cup \{F\}$, where "F has been deduced" is the k-th received message.*

Definition 7. *A redundancy module is* correct *if $R_k \models \{G_1, \ldots, G_n\}$ and G_1, ..., $G_n \models F$ whenever it outputs a message "replace F by $G_1, \ldots, G_n$" after receiving its k-th message.*

Definition 8. *A redundancy module is* finitary *if for every execution the number of output messages produced is finite.*

A prover module $\mathcal{P}$ can be combined with a redundancy module $\mathcal{R}$ in the expected way to obtain what we will call a theorem prover: both modules receive the same input set of formulae, and the output messages of each one of them are the received messages of the other one. The theorem prover halts if and only if $\mathcal{P}$ halts. Let us remark that it is of course not our aim to discuss priority or synchronization issues or the like between both modules; one can simply assume that $\mathcal{R}$ uses a finite amount of cpu time initially and after each received message; if this time is finished $\mathcal{R}$ (possibly after sending a message) transfers control to $\mathcal{P}$ until $\mathcal{P}$ emits the next message.

Theorem 1. *A correct and complete prover module in combination with a correct and finitary redundancy module is a correct and refutation complete theorem prover, that is, a theorem prover that halts with output "unsatisfiable" if, and only if, the input set of formulae S_0 is unsatisfiable.*

Proof. From the correctness of both modules it follows that for all i, all sets S_i, U_i, and R_i are logically equivalent to S_0 (by induction on the subindex i: the new formulae coming from a message that is sent are a logical consequence of these sets with smaller index). Since $\mathcal{R}$ is finitary it sends only k messages, for some k. If S_0 is unsatisfiable, the corresponding set U_k is unsatisfiable, and by completeness of $\mathcal{P}$ the prover will halt with output "unsatisfiable". Conversely, if $\mathcal{P}$ halts with output "unsatisfiable" after receiving its j-th message, then, by correctness of $\mathcal{P}$, the set S_j and hence S_0 is unsatisfiable. □

3 Finitary redundancy modules

Here we show how to create a finitary redundancy module, given a well-founded ordering $>$ on formulae. In order to abstract from concrete algorithms and strategies that can be used in a redundancy module, i.e., to be as general as possible, we first model the execution of a redundancy module by the notion of a *redundancy derivation*, a sequence of pairs of sets of formulae. The sets R_i correspond to what we defined in the previous section. The U_i will play the role of sets of formulae that are known to be unsatisfiable as the set U mentioned in the introduction (hence the name U, for unsatisfiable).

Definition 9. *We model the execution of a redundancy module by the notion of a* redundancy derivation*, a sequence* $(U_0, R_0), (U_1, R_1), \ldots$ *where all* U_i *and* R_i *are sets of formulae, and where the transition from a pair* (U_i, R_i) *to a pair* (U_{i+1}, R_{i+1}) *takes place whenever a message is received or sent:*

- $U_0 = R_0 = S_0$*, where* S_0 *is the input set of formulae.*
- *if a message "*F *has been deduced" is received then* $U_{i+1} = U_i$ *and* $R_{i+1} = R_i \cup \{F\}$
- *if a message "replace* F *by* $G_1 \ldots G_n$*" is sent then* $U_{i+1} = U_i \setminus \{F\} \cup \{G_1 \ldots G_n\}$ *and* $R_{i+1} = R_i$*.*

We now impose conditions ensuring that the redundancy module satisfies the correctness and finiteness requirements:

Definition 10. *Let* $>$ *be a well-founded ordering on formulae. We say that a formula* F *is* redundant *w.r.t.* $>$ *by* $G_1 \ldots G_n$ *if* $\{G_1 \ldots G_n\} \models F$ *and* $F > G_i$ *for all* i *in* $1 \ldots n$*.*

Let $\mathcal{R}_>$ *be a redundancy module modeled by a redundancy derivation* (U_0, R_0)*,* (U_1, R_1)*,* $\ldots$ *such that if* $U_{i+1} = U_i \setminus \{F\} \cup \{G_1 \ldots G_n\}$*, i.e., a message "replace* F *by* $G_1 \ldots G_n$*" is sent, then*

- $R_i \models \{G_1 \ldots G_n\}$ *and*
- F *is a formula in* U_i *that is redundant by* $G_1 \ldots G_n$

A practical redundancy module implementing $\mathcal{R}_>$ can be a procedure that updates the set R whenever it receives a message, and keeps looking for formulae $G_1 \ldots G_n$ that can be used for proving the redundancy of some F in the set U, by means of (probably) well-known redundancy elimination techniques, and then replacing F by these $G_1 \ldots G_n$. For example, one can simplify F by rewriting (demodulating) applying some equational formula G_1 of U, and replace F by its simplified version G_2. In practice it will be useful to apply formulae G_k that are in U, since, on the one hand, this helps keeping the set U small, which will be useful, and on the other hand, by construction all formulae in each U_i are logical consequences of R_i, so this does not have to be checked.

Theorem 2. $\mathcal{R}_>$ *is a finitary redundancy module.*

Proof. Let $\gg$ denote the multiset extension of $>$, that is, the smallest ordering such that $S \cup \{F\} \gg S \cup \{F_1, \ldots, F_n\}$ whenever $F > F_i$ for all i in $1 \ldots n$. The ordering $\gg$ is a well-founded ordering on finite (multi)sets of formulae whenever $>$ is a well-founded ordering on formulae (see e.g., [Der87]).

Indeed every U_i is a finite set of formulae (by induction on i: initially it is, and at each step only a finite number of formulae is added). Moreover U decreases wrt. $\gg$ at each message sent. Hence the number of output messages must be finite. □

Theorem 3. *$\mathcal{R}_>$ is a correct redundancy module.*

Proof. From the definition of $\mathcal{R}_>$, whenever a message "replace F by $G_1 \dots G_n$" is sent then $R_i \models G_1, \dots, G_n$ and $G_1, \dots, G_n \models F$. This implies correctness. □

4 How do we obtain a prover module from a given prover

4.1 A prover module from any prover

It is easy to see that, in a naive way, a complete prover module can be obtained from any correct and refutation complete theorem prover A (with the name A for any): given an unsatisfiable set of formulae S as input, A halts with output "unsatisfiable" after finite time. The main idea is simply to restart A with the new set U_k every time a message "replace F by $G_1, \dots, G_n$" is received.

Instead of formally defining the communications between A and the "layer" around it, that is, the remaining part of the prover module that uses A, we assume, without going into these details, that the layer knows whether A has inferred a formula, and that it is possible to tell A by means of the operation $Restart(S)$ that it has to restart from scratch with input the set of formulae S.

We now define the prover module in terms of the prover A. In this prover module there will be a set of formulae U that will play the same role as before (see Definition 2) and as the set U in the redundancy module. It will be maintained by this prover module according to this invariant. Hence, initially U is the input set S_0 and each time a received message "replace F by $G_1 \dots G_n$" is treated, the prover module will update U by doing $U := (U \setminus \{F\}) \cup \{G_1 \dots G_n\}$.

Summarizing, each time a message "replace F by $G_1 \dots G_n$" is received we do: $U := (U \setminus \{F\}) \cup \{G_1 \dots G_n\}$, and $Restart(U)$. Each time the prover A infers F, a message "F has been deduced" is sent.

Definition 11. *Let $\mathcal{P}_A$ denote the prover module as defined above.*

Theorem 4. *If A is a correct and refutationally complete prover, then $\mathcal{P}_A$ is a correct and complete prover module.*

Proof. We clearly have $U = U_k$ after the k-th message in $\mathcal{P}_A$. And moreover $S_k \models U_k$. Then, correctness of $\mathcal{P}_A$ follows from correctness of A and from the fact that A is restarted with input U_k after every received message. For completeness, suppose $\mathcal{P}_A$ receives exactly k messages. We have $U = U_k$ after the k-th message. Then, completeness of $\mathcal{P}_A$ follows from completeness of A, given that $\mathcal{P}_A$ just restarts A with input U_k after the k-th received message. □

It is doubtful that this naive use of A will lead to any practical usefulness, since restarting A from scratch is a very drastic measure that produces large amounts of repeated work. A possible alternative would be not to restart A after each received message, but restart it each n units of time for an arbitrary fixed n. Obviously this does not affect completeness. Another better solution is given in the next subsection, where no work is repeated due to restarts, and an important amount of unnecessary work is avoided.

4.2 A prover module built from a closure prover

Here we consider a given prover C (with the name C for closure) that is simply based on some correct and refutation complete inference system, and computes the closure w.r.t. this inference system using some strategy. Furthermore, it is assumed that this is done in such a way that the result is a complete prover: given an unsatisfiable set of formulae S as input, it halts with output "unsatisfiable", since it finds a contradictory formula, denoted by $\Box$.

In the following, this kind of prover will be called a *closure prover*. We will show that from C we can build a correct and complete prover module if C fulfils some minimal requirements needed in order to interact with it. The resulting prover module, in combination with a redundancy module, can reasonably well exploit our modular redundancy notions by means of *orphan murder*.

We now define the prover module in terms of the prover C.

In this prover module there will be two disjoint sets of formulae, U and P, that will be maintained by this prover module with the following invariants: the set U will play the same role as before (see Definition 2), and P is the set of remaining formulae stored by the prover C. Hence, initially U is the input set S_0, and P is empty, and each time a received message "replace F by $G_1 \ldots G_n$" is treated, the prover module will update P and U by doing $P := P \setminus \{G_1 \ldots G_n\}$ and $U := (U \setminus \{F\}) \cup \{G_1 \ldots G_n\}$ (note that this may cause F to disappear from $U \cup P$). Similarly, each time the prover C infers a new formula F, the prover module will update P by doing $P := P \cup \{F\}$.

Furthermore, we associate to each F in $U \cup P$ a set of sets of formulae, denoted by $Parents(F)$, with the invariant that the sets $\{F_1, \ldots, F_n\}$ in $Parents(F)$ correspond, one to one, to the inference steps with premises $F_1, \ldots, F_n$ and conclusion F that have taken place in the prover C and none of $F_1, \ldots, F_n$ have been removed after the inference step. In order to maintain these invariants, each time such an inference takes place, $\{F_1, \ldots, F_n\}$ will be added to $Parents(F)$ if F already existed in $U \cup P$, and $Parents(F)$ will be $\{\{F_1, \ldots, F_n\}\}$ if F is new. Similarly, since we require that all formulae occurring in $Parents(F)$ have not been removed afterwards from $U \cup P$, if $\{F_1, \ldots, F_n\} \in Parents(F)$ and some F_i is removed from $U \cup P$ then the set $\{F_1, \ldots, F_n\}$ is removed from $Parents(F)$.

In this context, the idea of orphan murder is the following. If for some formula F in P (and hence $F \notin U$) we have $Parents(F) = \emptyset$ then F appeared because it was inferred, in one or more ways, by the prover C, but, for each one of these ways, the premises do not exist any more. Hence F itself is not needed for the completeness of the prover C and can be removed as well. This triggers new eliminations of other formulae G of P if F occurs in all sets of $Parents(G)$, and so on. We assume that each time a formula is eliminated in this way, this fact can be communicated to the prover C by an operation $Remove(F)$, which has the effect that C simply removes F from its data structures. This operation has its counterpart $Add(S)$, which tells C to consider the formulae of a set of formulae S as additional input formulae.

Expressing this in an algorithmic way, we obtain the following. Initially U is the input set S_0, and P is empty. Each time a message "replace F by $G_1 \ldots G_n$" is received we do:

$\{$ Precondition: $F \in U$ **and** $Parents(F) = \emptyset$ $\}$
$Remove(F)$
$Add(\{G_1, \ldots, G_n\} \setminus (U \cup P))$
$P := P \setminus \{G_1 \ldots G_n\}$
$U := (U \setminus \{F\}) \cup \{G_1 \ldots G_n\}$
While $\exists F' \in (U \cup P)$ **and** $\exists S' \in Parents(F')$ **and** $S' \not\subseteq (U \cup P)$ **Do**
  $Parents(F') := Parents(F') \setminus \{S'\}$
  If $Parents(F') = \emptyset$ **and** $F' \in P$ **Then**
    $P := P \setminus \{F'\}$
    $Remove(F')$
  EndIf
EndWhile

As a precondition we assume that $F \in U$ and $Parents(F) = \emptyset$ since otherwise completeness may be lost. This precondition can be ensured in practice by an additional communication mechanism between both modules, for example, by making the redundancy module ask permission before it sends a message. On the other hand, each time the prover C infers F from premises $G_1 \ldots G_n$, a message "F has been deduced" is sent, and we do:

If $F \in (U \cup P)$ **Then**
  $Parents(F) := Parents(F) \cup \{\{G_1, \ldots, G_n\}\}$
Else
  $P := P \cup \{F\}$
  $Parents(F) := \{\{G_1, \ldots, G_n\}\}$
EndIf

Note that, instead of formally defining the interactions between C and the "layer" around it, that is, the remaining part of the prover module that uses C, we have assumed, without going into these details, that the layer knows whether C has inferred a formula from certain premises, and that it is possible to tell C by means of the operations $Remove(F)$ and $Add(S)$ that a certain formulae can be removed or have to be added.

Definition 12. *Let $\mathcal{P}_C$ denote the prover module as defined above.*

We will now prove that the prover module $\mathcal{P}_C$ is correct and complete. For this purpose we need a reasonable additional assumption on the prover C, namely a notion of fairness:

Definition 13. *A formula F is* persistent *in C if it is stored by C at some point and not removed by the operation $remove(F)$ later on.*

The prover C is called fair *if, whenever $G_1, \ldots, G_n$ become persistent formulae at some point of the execution, and there is a possible inference with them, then this inference is performed after that point.*

Lemma 1. *Given an execution of $\mathcal{P}_C$, if $G_1, \ldots, G_n$ are persistent formulae in C (and hence in $U \cup P$), and there is a possible inference with them producing F, then F is a persistent formula in C.*

Proof. By fairness, this inference is done at some point of the execution after $G_1, \ldots, G_n$ have become persistent. Furthermore $\{G_1, \ldots, G_n\} \in Parents(F)$ at that point. Moreover $\{G_1, \ldots, G_n\}$ will remain in $Parents(F)$. Then F will not be removed since $Parents(F)$ will never be empty. $\square$

Theorem 5. *$\mathcal{P}_C$ is a correct prover module.*

Proof. Let S_k be defined as in Definition 2. Then we will show that $S_k \models (U \cup P)$ after the k-th received message and before a new message is received, and the correctness of the procedure will follow trivially. It suffices to show that this property is satisfied initially, and it is preserved during the execution. Initially $U \cup P$ is S_0. When an inference is done, its conclusion F, which is added to P, is a consequence of $(U \cup P)$, and therefore, of the corresponding S_k. When a message is received, $S_k = S_{k-1} \cup \{G_1, \ldots, G_n\}$ and nothing else but $\{G_1, \ldots, G_n\}$ is added to $(U \cup P)$. $\square$

Lemma 2. *Let U_k be defined as in Definition 2. Then $U_k = U$ after the k-th received message in $\mathcal{P}_C$.*

Theorem 6. *$\mathcal{P}_C$ is a complete prover module.*

Proof. We have to show that, in an execution of our prover module with exactly k input messages, if U_k is unsatisfiable then the prover module halts with output "unsatisfiable". By the previous lemma, $U_k = U$ immediately after the k-th received message. Furthermore, by fairness all inferences with premises in U are performed after the point their premises become persistent. By Lemma 1 the corresponding conclusions are persistent as well. All these descendants will be persistent (by iterating this, i.e., by induction on the number of inference steps required to obtain a descendant F from U). Hence all formulae in the closure of U w.r.t. the given inference system are indeed retained by C, which implies that $\square$ will be inferred since the inference system of C is refutation complete. $\square$

In this subsection we have developed the prover module in terms of a prover that fulfils a reasonable fairness requirement. We could have followed an alternative approach, without this fairness requirement. Instead of explaining this alternative here for closure prover, we develop it for saturation provers in the next subsection. It will be clear that its application to closure provers will be a particular case.

4.3 A prover module from a saturation prover

Here we consider a standard saturation-based prover D (with the name D for prover with *deletion* of redundant formulae). Without loss of generality, such a prover D can be assumed to consist of some *strategy* controlling the application of only two basic operations: adding a new formula inferred by some correct deduction system, and removing a formula F whose redundancy follows from other formulae $F_1, \ldots, F_n$ w.r.t. some *internal* redundancy criterion with the minimal requirement that $F_1, \ldots, F_n \models F$ and a kind of transitivity property, namely if F is redundant by the set of formulae $F_1, \ldots, F_n$ and F_1 is redundant by the set of formulae $G_1, \ldots, G_m$ then F is redundant by the set $G_1, \ldots, G_m, F_2, \ldots, F_n$. Note that redundancy techniques like demodulation of F into F' can be expressed by a combination of these two steps: first add F' (which is a logical consequence), then remove F (which has become redundant).

As usual, a prover like D can be modelled by a derivation, i.e., a sequence of sets of formulae, of the form $D_0, D_1, \ldots$ where at each step the strategy decides by which concrete instance of one of the two basic operations D_{i+1} is obtained from D_i. Of course, this is done in such a way that the result is a complete prover. For technical reasons that will become clear soon, here we will make a reasonable additional assumption, namely that if, for a finite number of basic steps of the derivation (i.e., for a finite number of decisions about which basic step is performed next), the strategy is overruled by an external event, like user interaction, the prover D remains complete. Note that the usual internal fairness requirements of saturation-based provers are not affected by such a finite number of "wrong" decisions.

As in the previous subsection, here we assume that the prover D also provides the operations *Add* and *Remove*, and provides information each time it infers a formula F from a set of premises $F_1, \ldots, F_r$. In addition, here we will assume that each time the prover removes a formula F that is redundant by a set $F_1, \ldots, F_r$, this will be communicated as well.

In a practical prover, its strategy can be based on many different types of information (stored in data structures, like priority queues based on weighting functions) and that may depend on the current state as well as on historical information about previous states. Hence it has to be discussed how to deal with the external operations *Add* and *Remove* that were not foreseen when the strategy was developed. For instance, in a practical prover where a new formula F is added with the operation $Add(F)$, this F has to be inserted in the different data structures on which the strategy is based.

In the most general setting the strategy can be based on the whole *history* of the current derivation. Then the situation is as follows. Suppose that an operation *Add* or *Remove* takes place after step n of a derivation $D_0, \ldots, D_n$, and this operation converts D_n into another set S. Of course the strategy of the prover cannot be expected to behave well any more if it continues based on the history of the derivation $D_0, \ldots, D_n$. But fortunately, our prover module will only perform operations *Add* and *Remove* in such a way that another very similar derivation $D'_0, \ldots, D'_m$ exists where D'_m is S. Hence it is reasonable to expect that the prover D will behave well if it continues with the strategy it would have followed if it had reached S by this alternative similar derivation.

Therefore, we will assume that it is possible to update the internal state of the prover (on which its strategy is based) into the state it would be in after m basic steps of the alternative derivation $D'_0, \ldots, D'_m$. This is how we will proceed.

However, note that, although this alternative sequence consists of j correct steps, it may not be a derivation with the strategy. But since we know that the prover D is complete even if, for a finite number of decisions, the strategy is overruled by an external event, intuitively, the completeness of the prover module follows, since there will only be a finite number of such external events.

We now define the prover module in terms of the prover D. Again there will be sets U and P and $Parents(F)$ for each F in $U \cup P$. As before $U \cup P$ contains the formulae kept by the prover D, but, due to the internal redundancy, U may not be U_k but a subset of it. In addition, each formula in $U \cup P$ can also be *blocked* for external removals, i.e., it will never be removed by a $Remove(F)$ operation, and we denote by $Blocked(U \cup P)$ the set of blocked formulae in $U \cup P$. In this case we only need the following invariant: $U \cup Blocked(U \cup P) \models U_k$. No assumption is needed about the set $Parents(F)$ in order to prove correctness and completeness of the prover module. Nevertheless, as we will see at the end of this section, the way we deal with these sets in combination with the way we construct the alternative derivations will allow us in most of the cases to avoid repeated work. Expressing the previous ideas algorithmically, we obtain the following. Initially U is the input set S_0, and P is empty.

Each time a message "replace F by $G_1 \ldots G_r$" is received we do:

{ Precondition: $F \in U$ **and** $Parents(F) = \emptyset$ **and** F is not blocked }
$Remove(F)$
$Add(\{G_1, \ldots, G_r\} \setminus (U \cup P))$
$P := P \setminus \{G_1 \ldots G_r\}$
$U := (U \setminus \{F\}) \cup \{G_1 \ldots G_r\}$
While $\exists F' \in (U \cup P)$ **and** $\exists S' \in Parents(F')$ **and** $S' \not\subseteq (U \cup P)$ **Do**
    $Parents(F') := Parents(F') \setminus \{S'\}$
    If $Parents(F') = \emptyset$ **and** $F' \in P$ **and** F' is not blocked **Then**
        $P := P \setminus \{F'\}$
        $Remove(F')$
    EndIf
EndWhile

The precondition is ensured as in the previous subsection. On the other hand, as said at the beginning of this subsection, after each $Remove(F)$ or $Add(F)$ operation, we have to update the internal state of the prover into a new state which can be reached by a correct derivation with the prover D ending with the same final set of formulae. Below, such an alternative derivation is given for each one of these operations. The prover is then updated into the state it would be in after producing the alternative derivation.

The alternative derivation can be built in several different ways. As a general rule, for provers keeping more historical information it pays off to build alternative derivations that mimic the past computations quite accurately in order to avoid repeated computations; for others, similar results will be obtained by shorter derivations leading to the current clause set. In any case, for the main results, it suffices for the moment to assume that the alternative derivation is defined correctly, in the sense that its final state corresponds to the current set of formulae stored in the prover.

Each time the prover D infers F from premises $G_1 \dots G_r$, a message "F has been deduced" is sent, and we do:

If $F \in (U \cup P)$ **Then**
  $Parents(F) := Parents(F) \cup \{\{G_1, \dots, G_r\}\}$
Else
  $P := P \cup \{F\}$
  $Parents(F) := \{\{G_1, \dots, G_r\}\}$
EndIf

Finally, each time the prover D removes a formula F due to its redundancy by a set $F_1 \dots F_r$, we do:

Foreach F_i in $\{F_1 \dots F_r\}$ **Do**
  $Block(F_i)$
EndForeach
$U := U \setminus F$
$P := P \setminus F$

Note that, although at this point we do not apply any orphan murder, it will be applied only if, later on, another "replace" message is received by the redundancy module, which triggers the actual *Remove* operations. This is done because we can ensure completeness only if no more than a finite number of *Remove* operations occur, and there may be an infinite number of internal redundancy steps.

Definition 14. *Let $\mathcal{P}_D$ denote the prover module as defined above.*

Theorem 7. *$\mathcal{P}_D$ is a correct prover module.*

Proof. Analogous to the correctness proof of the previous section. □

Theorem 8. *$\mathcal{P}_D$ is a complete prover module.*

Proof. We have to show that, in an execution of our prover module with exactly k input messages, if U_k is unsatisfiable then the prover module halts with output "unsatisfiable".

Since there are finitely many input messages, there are finitely many, say n, operations *Add* or *Remove*. Let $D_0, \ldots, D_m$ be the n-th alternative derivation. By the invariant of the process we have $U \cup \mathit{Blocked}(U \cup P) \models U_k$, therefore $U \cup P$ is unsatisfiable.

Finally, since there are only a finite number of steps (m) contradicting the strategy of the prover D, it remains complete, and hence the prover module halts with output "unsatisfiable". □

Now it remains to define the alternative derivations. Before it was mentioned that for some provers it pays off to build alternative derivations that mimic the past computations quite accurately in order to avoid repeated computations and for others perhaps not. Here we explain a way that attempts to preserve as much as possible the previous history.

Suppose $D_1, \ldots, D_n$ is the current derivation. The following additional invariant, which is necessary to prove the correctness of the alternative derivation, is always satisfied: all formulae in $(\bigcup_{i \leq n} D_i) \setminus (U \cup P)$ are internally redundant by $\mathit{Blocked}(U \cup P)$.

- **an operation** $\mathit{Add}(F)$ **takes place.** Then, a new derivation, with $D_n \cup F$ as final set, is constructed as follows. There are two cases to be considered. If $F \notin D_i$ for all $i \leq n$ then we consider the derivation $D_0 \cup \{F\}, \ldots, D_n \cup \{F\}$. Otherwise, let D_i be the set with maximal i such that $F \in D_i$. Note that $i < n$. Hence D_{i+1} is obtained by an internal redundancy of F. Then the alternative derivation is $D_0, \ldots, D_i, D_{i+2} \cup \{F\}, \ldots, D_n \cup \{F\}$.
- **an operation** $\mathit{Remove}(F)$ **takes place.** Then, a new derivation, with $D_n \setminus \{F\}$ as final set, can be constructed. The new derivation $D'_0, \ldots, D'_m$ consists of two parts $D'_0, \ldots, D'_k$ containing only inference steps and $D'_k, \ldots, D'_m$ containing only internal redundancy steps.
  1. We take D'_0 as $(D_0 \setminus \{F\}) \cup \mathit{Ch}(F)$, where $\mathit{Ch}(F)$ denotes the set of children of F in inferences along the sequence $D_0, \ldots, D_n$. Let $D_{i_1}, \ldots, D_{i_k}$ be the sets of the derivation obtained by an inference step where F is neither a premise nor a conclusion. Then $D'_0, \ldots, D'_k$ is the derivation where D'_j is obtained from D'_{j-1} by the same inference step between D_{i_j-1} and D_{i_j}. Note that all steps are correct since we have added $\mathit{Ch}(F)$ to the initial set. Now, we have $D'_k = (\bigcup_{i \leq n} D_i) \setminus \{F\}$, and hence $D_n \setminus \{F\} \subseteq D'_k$.
  2. The second part of the new derivation consists of eliminating the formulae in $D'_k \setminus D_n$ by internal redundancies. This can be done since $D_n \setminus \{F\}$ contains all blocked formulae (F is not blocked since a $\mathit{Remove}(F)$ is done), and, by the invariant above, all formulae in $D'_k \setminus D_n$ are redundant by the blocked formulae.

Regarding the aim of avoiding repeated work, a last invariant, that is also satisfied, shows the reason for mantaining the sets $\mathit{Parents}(F)$: for all inference steps in the derivation with premises $F_1, \ldots, F_r$ and conclusion F, s.t.

none of $F_1, \ldots, F_r, F$ are redundant by $Blocked(U \cup P)$, we have $\{F_1, \ldots, F_r\} \in Parents(F)$. This invariant ensures that the only missing information in the sets $Parents(F)$ (about performed inferences) involves either a premise or a conclusion that is redundant by the blocked formulae. Therefore, it is clear that some repeated work might be done, but, in any case, the prover could avoid it by means of internal redundancy steps.

5 Conclusions

A notion of modular redundancy for theorem proving has been introduced, that can be used to exploit redundancy elimination techniques in combination with arbitrary existing theorem provers, in a refutation complete way, even if these provers are not (or not known to be) complete in combination with the redundancy techniques when applied in the usual sense.

We have explained for three cases how to build a prover module in terms of an existing prover: for arbitrary provers, closure provers and saturation-based provers. The latter case seems to be the most general and useful one, since there are several practical applications where the standard general redundancy notions cannot be used, like [Lyn97] [BG98], and [BGNR99].

Especially in the case of [BGNR99], where the completeness of paramodulation w.r.t. non-monotonic orderings is proved, we believe that our modular redundancy results will be the method of choice. The reason is that the standard redundancy notions should be applied w.r.t. an ordering that is built (at the completeness proof level) based on semantic information and hence cannot even be approximated in a strong enough way during the saturation process. Moreover, in many practical applications the existence of a monotonic ordering is too strong a requirement. For example, one may want to apply completion to a terminating set of rules with a given orientation by a reduction ordering that cannot be extended to a total one, like $f(a) \to f(b)$ and $g(b) \to g(a)$, for which a and b must be incomparable in any monotonic extension. Another typical situation is deduction modulo built-in equational theories E, where the existence of a total *E-compatible* reduction ordering is a very strong requirement. We plan to do some experiments in this context with the Saturate prover[NN93,GNN98].

References

[BG94] Leo Bachmair and Harald Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217–247, 1994.

[BG98] Leo Bachmair and Harald Ganzinger. Strict basic superposition. In Claude Kirchner and Hélène Kirchner, editors, *Proceedings of the 15th International Conference on Automated Deduction (CADE-98)*, volume 1421 of *LNAI*, pages 160–174, Berlin, July 5–10 1998. Springer.

[BGNR99] Miquel Bofill, Guillem Godoy, Robert Nieuwenhuis, and Albert Rubio. Paramodulation with non-monotonic orderings. In *14th IEEE Symposium on Logic in Computer Science (LICS)*, pages 225–233, Trento, Italy, July 2–5, 1999.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[GNN98] Harald Ganzinger, Robert Nieuwenhuis, and Pilar Nivela. The Saturate System, 1998. Software and documentation maintained at `http://www.mpi-sb.mpg.de`.

[Lyn97] C. Lynch. Oriented equational logic programming is complete. *Journal of Symbolic Computation*, 23(1):23–46, January 1997.

[NN93] Pilar Nivela and Robert Nieuwenhuis. Practical results on the saturation of full first-order clauses: Experiments with the saturate system. (system description). In C. Kirchner, editor, *5th International Conference on Rewriting Techniques and Applications (RTA)*, LNCS 690, pages 436–440, Montreal, Canada, June 16–18, 1993. Springer-Verlag.