A Mechanical Verifier for Supporting the Design of Reliable Reactive Systems

Tie-Cheng Wang and Allen Goldberg
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

Abstract

An automated verification system, Reacto-Verifier (RVF) developed for supporting the design of reliable reactive systems is described. In order to make the formal verification of large and/or complex systems tractable, RVF is enhanced by a knowledge-base manager, a proof manager, and a dependency maintenance procedure. The knowledge-base manager supports a flexible use of a large set of axioms and rules derived from the domain theory of the specification language. The proof manager helps handle verification failure and supports off-line development of proofs. The dependency maintenance procedure permits the user to trace the history of a derivation and supports efficient addition and/or retraction of assumptions. RVF can be used both for batch-style automated verification, and for incremental development of verified programs.

1 Introduction

We report here a formal verification system, Reacto-Verifier (RVF). RVF is the consistency checker for the Reacto system [9]. The Reacto system, developed at Kestrel Institute, is an environment for the executable specification of reliable reactive systems and the formal verification that such systems satisfy the given requirements. Reacto uses hierarchically-structured finite state machines to model reactive systems, and a functional language over abstract types, including predefined set-theoretic data types to specify the conditions and actions associated with transitions. Reliability is achieved by three distinct activities: one is the use of a transformation-based compiler and a formal proof of its correctness; another is the ability to prototype a specification using an execution simulator; and a third is formal verification of specification consistency with the mechanical verifier RVF described here.

There exists a number of factors that make the formal verification of large and/or complex programs tractable. Due to the explicit notion of states and transitions, the verification of a Reacto program is decomposed naturally into the verification of the correctness of each transition. The use of a very high-level functional language for specifying transitions and annotating states, permits the user to focus on the essential properties of the design for the purpose of formal verification. Implementation detail which adds to the complexity of verification condition generation and theorem proving is suppressed at this level. The verifier has been enhanced by a proof management facility, which helps extract the unproven verification conditions, and permits the user to make off-line development of proofs for them. A knowledge-base manager is designed to support a flexible use of a large set of axioms and rules derived from the domain theory of the specification language. A dependency maintenance procedure is incorporated which permits the user to trace the history of a derivation, and supports efficient addition and/or retraction of assumptions. The theorem prover is designed to support verification activities. The prover is based on a goal-oriented proof procedure hierarchical deduction incorporated with term-rewriting, partial-evaluation, and forward-inference procedures. The prover can be used as an automated system, or as an interactive proof checker.

Fig. 1 depicts the main components of RVF and a portion of the user interface. More details about these components and about their relations will be given in the following description.

RVF can be used both for batch-style verification and for supporting incremental development of specifications and proofs. For a simplest application of the verifier, the user needs only to apply it to the root state of the program or subprogram to be verified. The verifier tries to verify the correctness of each transition contained in the state according to the assertions associated with its originating and destination states. For each of those transitions that can not be verified automatically, the proof manager will save the
proof-objects associated with it into a proof-directory.
Upon termination, the verifier will give a summary of the unproven transitions and the associated unproven conjectures. For each unproven conjecture, the user can determine the reason of proof failure by examining the corresponding proof-object that has been saved. If the conjecture is indeed a theorem, then the user can help the prover to finally prove it by editing the proof-object with additional definitions, rules, intermediate lemmas, proving strategies, control parameters, etc.

Fig. 1: The Architecture of Reacto-Verifier

Section 2 briefly reviews Reacto language. Section 3 describes the verification condition generator. Section 4 describes the knowledge-base manager. Section 5 describes the proof manager. Section 6 describes the theorem prover. Section 7 describes applications of RVF. Section 8 concludes this report.

2 Overview of the Language

A Reacto program is a hierarchically structured finite state machine. States are either primitive states, which do not contain substates, or non-primitive states which do have substates, one of which is designated as the initial state. A transition to a non-primitive state is semantically equivalent to a transition to its initial state, and a transition from a non-primitive state is equivalent to a collection of identical transitions from each of the substates. The state hierarchy provides modularization that makes the specification of large systems easier [13]. Associated with each state is a collection of variables, whose scope is that state (and its substates) and whose extent is the complete computation, i.e. they remain allocated and their value is preserved upon exit and re-entrance into the state. Also associated with a state is an assertion, a first-order logic formula over the visible variables of the state which specifies an invariant true upon entrance to any substate of the state.

Associated with a transition is an enabling predicate which guards execution of the transition, and an action which is a collection of assignment statements. The expressions on the right-hand side of the assignment statements are specified by (terminating) functional programs. A special self transition (from a state back to itself) called a history transition is equivalent to self-transitions on each of its substates. Other details of the Reacto language can be found in [9].

Consider the task of programming a key assignment module for a bank security system. The system maintains a database of keys, $keydb$, and two identifiers, $keyloc$ and $keyval$. The task of the module is that, when invoked, it must prompt the user to either select an existing key from $keydb$, or select an unused entry of $keydb$, then define a new key in certain steps (not described here) and store it into the entry. The security requirement for the design is that any modification to $keydb$ should not affect its consistency, and when the process exits from the module, $keyloc$ must have a value which is an index of $keydb$, and $keyval$ must have a value which is a valid key identical to the one stored in the $keyloc$ entry of $keydb$.

The non-primitive state that specifies this behavior is called define-key.

\[
\begin{align*}
\text{state} & \quad \text{define-key} \\
\text{vars} & \quad \text{given-kdata: string, given-kname: string, given-keyloc: int, display-line1: string, display-line2: string} \\
\text{assertion} & \quad V (\text{loc}) \ (\text{loc} \in [0..15]) \\
& \quad \land (\text{keydb}(\text{loc})).\text{keyname} \neq \text{"available"} \\
& \quad \land \exists (u, v, w) \ \text{keydb}(\text{loc}) \equiv \langle u, v, w \rangle \\
& \quad \land \text{valid-kname}(u) \\
& \quad \land \text{valid-kdata}(v) \\
& \quad \land \text{int}(w) \\
\text{substates} & \quad \text{define-key-init, keydb-display, define-kloc, kloc-defined, define-kname, define-kdata, define-key-return} \\
\text{initial-state} & \quad \text{define-key-init} \\
\end{align*}
\]

The identifiers, ‘$keydb$: map(int, key-type)’, ‘$keyval$: key-type’ and ‘$keyloc$: int’ are local variables of a superstate of the define-key state, where $keydb$ has been initialized to be a map with a total of 16 entries of keys. Each key is a tuple consisting of three fields: $kname$, $kdata$, $update-no$. An entry of $keydb$ is unused if the $kname$ field of the entry is the string "available."

Note that since the assertion annotated to a state applies to all of its substates, then the assertion given in the above state is exactly the consistency requirement about $keydb$ for the entire module. Each substate may have its own assertion. For example, the special requirement for the process to exit from the module is
specified by the assertion given in the following define-key-return state.

\[\text{state define-key-return} \]
\[\text{assertion kloc in [9 .. 15]} \land kvalue = \text{keydb}(\text{kloc})\]
\[\land 3 \big( n, v, w \big) \land (kvalue = \llcorner v, v, w \lrcorner) \land \text{valid-kdata}(v) \land \text{int}(w)\]

Functions and types are defined globally with the functional language.

\[\text{type key-type} = \text{tuple(kname string, kdata string, update-no int)}\]
\[\text{function valid-kdata (input) =} \]
\[\text{if string(input) then} \]
\[\text{if} \ \text{string(input)} \text{then} \]
\[\text{else false}\]

The entire module has been specified by 10 states and 12 transitions. We list one transition below as an illustration. (define-kdata-3 mentioned in the transition is a substate of the define-kdata state, which is in turn a substate of define-key-return state).

\[\text{transition define-kdata-3-transition-1} \]
\[\text{from define-kdata-3 to define-key-return}\]
\[\text{predicate valid-kdata} \{\text{"keyboard-input"}\}\]
\[\text{action given-kdata} \] 
\[\text{= concat(given-kdata, "keyboard-input"));} \]
\[\text{kloc} \] 
\[\text{= given-kloc;} \]
\[\text{kvalue} \] 
\[\text{= given-kname, given-kdata, 0);} \]
\[\text{keydb(kloc)} \] 
\[= \text{kvalue}\]

3 Verification Condition Generator

Reacto-Verifier (RVF) is based on an extension of Floyd's inductive assertion method [6]. One use of RVF is to prove that a Reacto program is consistent with respect to the annotated assertions, that is, a proof that the assertions associated with each state of a Reacto program holds each time a state is entered. According to Floyd's method, this can be done by verifying that, for each transition \(t\) into \(s\), if the assertion associated with its originating state is assumed to be true, then the assertion associated with \(s\) must be true whenever \(t\) is enabled and then terminated. In verifying the correctness of each transition, RVF is designed to proceed in two steps. First it deduces a verification condition \(vc\) by a verification condition generator (VCG), then it proves \(vc\) by a theorem prover. The semantics of Reacto has been kept simple and so its specification by the VCG is straightforward.

A basic step of verifying a transition involves proving a statement of the form, "if an assertion \(p\) holds and an action \(\alpha\) is taken then an assertion \(q\) holds after \(\alpha\) terminates." Using classical techniques, this statement is proved by first deducing the weakest liberal precondition \(\text{wlp}(\alpha, q)\) from \(p\) and \(q\), then proving that \(p \Rightarrow \text{wlp}(\alpha, q)\) holds. A condition \(c\) is called a weakest liberal precondition of \(\alpha\) and \(q\) if \(c\) holds if and only if \(\alpha\) does not terminate or \(\alpha\) terminates and \(q\) is true after \(\alpha\) terminates. The formula \(p \Rightarrow \text{wlp}(\alpha, q)\) is called the verification condition (VC) of \(\alpha\) with respect to \(p\) and \(q\). If \(\alpha\) is a multiple-assignment statement,

\[\alpha : x_1, \ldots, x_n := e_1, \ldots, e_n,\]

where no variables \(x_1, \ldots, x_n\) are common, then we have

\[\text{wlp}(\alpha, q) \Rightarrow q^{x_1, \ldots, x_n}_{e_1, \ldots, e_n}\]

where \(q^{x_1, \ldots, x_n}_{e_1, \ldots, e_n}\) is obtained from \(q\) by a simultaneous substitution of \(x_i\) with \(e_i\), for \(1 \leq i \leq n\). In Reacto, the variable \(x_i\) may be a map entry, a tuple field, the \(n\)th element of a sequence, etc. An advanced substitution algorithm, based on existing methods [2, 12], has been used for the substitution of this type of variables and for handling the case in which \(\alpha\) is a sequence of assignment statements, and/or the right hand side of an assignment statement is an 'if-then' or 'if-then-else' statement.

Let \(\text{pds}(s)\) be the set of primitive descendant states covered by \(s\),

\[\text{pds}(s) = \{ \text{if } \text{substate}(s) = \emptyset \}
\text{then (if } s \text{ is a return-state}
\text{then } \{ \}
\text{else } \{ s \}\}
\text{else } \bigcup_{s' \in \text{substate}(s)} \text{pds}(s').\]

Let \(\text{super}^*(s)\) be the set of all ancestor states of \(s\), including \(s\) itself, and let \(\text{init}^*(s)\) be the primitive initial-state of the state \(s\). Let \(\text{ass}^*(s)\) be the assertion associated with \(s\),

\[\text{ass}^*(s) = \bigwedge_{s' \in \text{super}^*(s)} \text{ass}(s').\]

Let \(tr\) be an assignment transition, \(s_o = \text{from-state}(tr), s_1 = \text{to-state}(tr)\). The verification condition \(\text{vc}(tr)\) of \(tr\) is given by considering two cases:

1. The history-flag of \(tr\) is false:

\[\text{vc}(tr) = \text{wlp(action(tr), ass}(\text{init}^*(s_o))).\]

2. The history-flag of \(tr\) is true and \(s_o = s_i: \text{ass}^*(s') \land \text{predicate}(tr)\]

\[\text{vc}(tr) = \bigwedge_{s' \in \text{pds}(s_o)} \text{wlp(action(tr), ass}^*(s'))\]

For a brief explanation, we note that, for the transition \(tr\) of case 1, there is a set of transitions, each of which originates from a primitive state of \(s_o\).
and terminates at the same initial primitive substate of \( s_1 \). Therefore, the correctness of the transition \( tr \) is determined by the whole set of transitions. Since their destination states are identical, we need only to produce one weakest liberal precondition, \( wlp(action(tr), ass^+(init^+(s_1))) \), and formulate the verification condition by using a disjunction of the assertion associated with each of the primitive states covered by \( s_1 \) as our hypothesis. For case 2, since the transition \( tr \) actually means a set of transitions, each of which originates from a primitive substate (except return-states) to the same primitive substate, and since the assertion associated with each primitive substate may be different, \( vc(tr) \) is formulated as a conjunction of verification conditions, each corresponding to a transition on a primitive substate.

4 Knowledge-Base Management

The formula \( vc(tr) \) deduced by VCG for a transition \( tr \) is usually not a theorem. However, if \( tr \) is correct with respect to the associated assertions, \( vc(tr) \) should be a valid logical expression in the underlying domain-theory \( T \) of the language. Thus, what we expect is that, if \( tr \) is correct, then the theorem prover of RVF can prove or help prove that \( T \Rightarrow vc(tr) \) is a theorem.

The existence of a large set of axioms from the domain theory and the axiom schemata induced by the induction principles for recursively constructed data types is a distinguishing feature of the mechanical theorem proving associated with program verification. How to handle these domain axioms is the key issue in the design of a mechanical verifier. In this regard, a number of important approaches have been made in past years. Our approach follows those taken by [4, 7, 10, 14]. We have built some axioms from ring theory into the inference rules and the unification algorithms of the prover. We have also designed a knowledge-base (KB) for storing the remaining axioms and lemmas (currently the KB contains about 300 rules).

Our objective is to help the theorem prover avoid the search space explosion that can result from the existence of such a large KB. One strategy that we use is the transformation of as many axioms and lemmas as possible into term-rewriting rules. However, there are still some that are not well suited to be used as term-rewriting rules in our system. For example, there is a rule regarding a subset relation,

\[
\forall(x, s_1, s_2)(x \in s_1 \Rightarrow (s_1 \subset s_2 \Rightarrow x \in s_2)).
\]

Without a suitable control, the rule can be fired by any clause that has a literal of form \( a \in s_1, c \subset d \), or \( \neg(a \in s_1) \). Unfortunately, it is very difficult to define a fixed constraint to these sorts of rules, so that it is suitable to all of their possible applications.

With regard to this problem, we conjecture that a feasible approach is to allow the user to freely choose the subset of rules and define special constraints that he/she believes to be useful for the prover in solving a non-simple problem. Our design of the knowledge-base (KB) manager is motivated by this consideration. We have defined a number of distinct rule types. Associated with each rule type is a set of restrictions and control strategies. The user can control the use of each axiom or lemma by assigning it a rule type and/or by defining specific constraints. In the following, we will first give a general description of these rule types, then present the KB manager.

Knowledge-Base Rules. We have defined five rule types: reduction, elimination, forward-implication, backward-implication, and any-rule. Roughly speaking, a reduction rule is used for term-rewriting. An elimination rule is also a sort of term-rewriting rule, but it is used to remove accessors. So it plays a role similar to the elimination rule of Boyer-Moore's prover [4]. Forward-implication rules and backward-implication rules are used by the forward-inference and backward-inference procedures, respectively. A rule whose rule type is any-rule is one with the weakest restriction: it can be used both by the forward-inference procedure and the backward-inference procedure. The following is an any-rule type KB rule.

\[
\text{kbr \ elm-vs-subset-relation: any-rule \{domain-theory\}}
\]

\[
\forall(x: \alpha, s_1: \text{set}(\alpha), s_2: \text{set}(\alpha))
\]

\[
(x \in s_1 \Rightarrow (s_1 \subset s_2 \Rightarrow x \in s_2)).
\]

Knowledge-Base Management. RVF maintains three distinct KBs: a persistent knowledge-base (PKB), a global knowledge-base (GBK), and a working knowledge-base (WKB). The PKB is used to store axioms from the domain theory and those extensions of the theory which have been approved as reusable. The WKB is the knowledge base used by the prover in proving a particular conjecture. While the PKB is relatively stable, the WKB is usually modified dramatically during a proof process.

GBK is the initial state of WKB, which contains some rules loaded from PKB and those given by the user. However, the content of GKB is not fixed: it is determined by RVF or by the user according to the given theorem proving task. For instance, in proving a theorem which requires a deep reasoning, we may limit the GKB to the set of term-rewriting rules of the PKB plus some user-defined rules. A user-defined rule is not required be already proved, but it can be attached with special constraint. For example, one may add a rule similar to \( \text{elm-vs-subset-relation} \) above into GKB, but
define backward-implication as its rule-type. Then the rule can only be used by a backward-inference procedure, and in addition, its literal $x \in s_2$ can be resolved upon in the deduction.

Of course, all rules chosen by the user for GKB must eventually be verified to be valid. However, presumably, they are much easier to prove, because they are assumed to be equivalent (maybe, module the set of term-rewriting rules) to some rules of PKB or some immediate consequences of them. Thus to verify the validity of these rules, we can direct the theorem prover to do a sort of variant-checking, instead of normal theorem-proving. Since the variant-checking can only perform shallow reasoning, we can allow the prover to use the full set of rules of the PKB without the risk of an explosive search. Of course, there may be some rules, which can not be verified by variant-checking, either because that they are not true with respect to the PKB, or that they are not a variant of any PKB rule. In the later case, we can prove the rule separately by using the proof management facility. Once the rule is proved, it can be added to the PKB for reuse.

5 Proof Management

The proof manager of RVF is designed for handling verification failure and for supporting incremental development of proofs. More precisely, it is designed to do the following:

1. Decompose and simplify each VC into a set of conjectures. Create a proof-object for each conjecture with proving-strategies and control parameters provided by default or by the user, and then call the main prover to prove it.
2. Provide a working environment for the user to help RVF prove those conjectures that can not be proved by RVF in its initial proof attempts.
3. Provide a working environment for proving additional rules and lemmas, and maintain persistent records for those that are allowed to be reused.
4. Keep track of the entire verification process and maintain the validity and consistency of the verification results when changes occur in the specifications or in the knowledge base.

The proof manager consists of a natural deduction controller and a proof-object manager. As indicated earlier, the verification condition $vc(tr)$ for a transition $tr$ is usually a fairly large and complex formula. The natural deduction controller is used to decompose the formula into a set of conjectures, and name each conjecture in such a way, that the same name will be assigned to the corresponding conjecture when RVF is invoked again to verify the same program. The name is useful for the proof-object manager to identify the proof-file which was created by previously running the RVF for the same component VC, and which may have been edited by the user. The decomposition is based on the IMPLY algorithm of Bledsoe's UT Interactive Prover [3]. But our procedure is simpler: it applies decomposition only to closed formulas.

A proof-object is a named object with attributes: proof-status, proof-origin, proof-exports, proof-operations, proof-rules, proof-conjectures, local-bindings, Skolem-symbols, etc. For each proof-object created by RVF, the first element of proof-conjectures will be the component VC to be proved. Proof-operations and proof-rules contain definitions and rules given by the user for helping the prover. The attribute proof-exports contains names of those proof-conjectures that are allowed to be reused after the validity of the entire object is established. Proof-origin is the transition or state name for which the proof-object is created. This name will be useful for identifying the binding and data type for each program variable contained in the object when it is loaded from an existing proof-file. The attributes local-bindings and Skolem-symbols are used to introduce new constant or function symbols, and meaningful Skolem function symbols for skolemizing formulas contained in the object, respectively.

A proof-conjecture is a named object with attributes: status, predicate, proving-strategies, supporting-set, etc. Predicate is the theorem to be proved. The value of proving-strategies is a list of strategies that the prover is suggested to use for proving the predicate. Supporting-set is used to record the set of those rules and lemmas that are necessary for the proof. Each conjecture may optionally contain special values for some control parameters. Among them, forward-depth is the maximum number of forward-inference steps; backward-depth is the same as local-depth-limit of [15]; fns-nest-limit is the maximum number of repetition of a function symbol in its substructure; and the value of maximum-induced-vars is the maximum number of variables that can be contained in a goal clause, etc. Fig. 2. contains an example of a user created proof-object.

6 The Theorem Prover

The theorem prover consists of a simplifier, a forward-inference procedure (FIP) and a backward-inference procedure (BIP). Given a conjecture to be proved, it will be simplified immediately by the simplifier. If the resulting formula is true, then a proof has been found. Otherwise the formula is negated and transformed into a set $S$ of clauses (or a set of sets of clauses, if the prover is directed to use a case-analysis strategy). Then the
prover initializes the WKB with GKB, and calls the FIP with \( S \). The FIP will add \( S \) to the WKB, and deduce additional consequences from \( S \) and rules contained in the WKB, and add some or all of these consequences to the WKB. The prover will terminate with a proof if a refutation is found. Otherwise the BIP is invoked to deduce a contradiction with the WKB and with goal clauses chosen from \( S \). Some details of these procedures are given below.

The simplifier is designed to replace a term or a subterm by a simpler, but semantically equivalent term or subterm. The simplification is carried out by three procedures, namely, canonicalization, term-rewriting, and partial evaluation. Canonicalization is done by using a set of building-in term-rewriting rules. For example, the procedure will replace \( \neg(\neg p) \) by \( p \), \( a = a \) by \( \text{true} \), \( \text{true} \land p \) by \( p \), etc. Partial evaluation is used to deduce canonical forms for those terms or subterms, that are computable. For example, this procedure will reduce \((x + 2 + 3)\) to \( x + 5 \), and reduce \( 2 \times 3 \times 4 = 100 \) to \( \text{false} \). Term-rewriting is the main component of the simplifier, whose behavior is determined by the set of term-rewriting rules contained in the WKB.

The FIP accepts a set of clauses as the set-of-support [17], and deduces a set of clauses by unit resolution and paramodulation [17], using rules stored in the WKB. The FIP is used in order to achieve the following goals: 1. proving the conjecture (i.e. deducing a contradiction) if possible under the given constraints; 2. deducing additional rules which will be used by the BIP if the FIP terminates without a proof; 3. deducing additional term-rewriting rules for the simplifier.

The FIP is not allowed to use a backward-implication rule. Moreover, for a non-unit forward-implication rule, only literals contained in its antecedent can be resolved or paramodulated upon. If the entire antecedent of such a rule has been reduced to \( \text{true} \), then the (instantiated) consequent of the rule will become a backward-implication rule. Since a backward-implication rule thus produced usually contains fewer literals and fewer variables than its parent rule, this stipulation can be used to direct the BIP to make restricted use of certain rules.

The BIP is the most powerful proof procedure of RVF, which is based on the goal-oriented proving procedure, hierarchical deduction [15]. The BIP proves a theorem by traversing a tree of nodes. Each node contains a different set of rule clauses. All candidate goal clauses are contained in a goal-list. Each literal of a goal is indexed by a node name, through which a set of nodes can be located to obtain rule clauses for the resolution and paramodulation upon that literal. The root node of this tree is the WKB (but the BIP is not permitted to use a non-unit forward-implication rule). At the beginning of a deduction, the tree contains only the root node, while the goal-list contains only goal clauses chosen from \( S \). All literals of these goal clauses are indexed by the name of the root node.

The general proof process is as follows:

1. A goal clause \( G \) is taken from the goal-list, along with the index of the first literal of \( G \). A set of rule clauses is obtained by retrieving the node indicated by this index and all parent nodes of this node. This index will be the parent name of the new node being produced.

2. All "legal" resolvents are produced by applying resolution and paramodulation to \( G \) upon its first literal against each rule that is retrieved. For each of the resolvents produced, the indices and the order of the literals inherited from the goal clauses are retained. But the indices of the literals inherited from the rule clause are then replaced by a larger integer which will be the name of a node just produced.

3. If a contradiction is obtained, then the procedure reports the proof (refutation) path that has been found, and then returns \( \text{true} \). Otherwise the resolvents will be stored in the new node, and inserted into the goal-list according to their priorities computed with some heuristic evaluation functions. Then the above process is repeated.

In comparison with other goal-oriented proving procedures, the BIP admits proofs usually with fewer selections of goals. The "legal" resolvents of BIP are produced under a set of constraints or narrowing strategies, such as local subsumption, constraints on common tails, proper reduction, global subsumption, subgoal reordering, partial set of support, and semantic guidance, etc. The reader interested in a discussion of these constraints and strategies may refer to [15]. A special control (not given here) has been made to the paramodulation and to the use of the simplifier for the BIP.

7 Applications of RVF

RVF can be used both for batch-style verification and for incremental development of specifications and proofs. However, considering the high cost of debugging a program by formal verification, our approach encourages the user to apply formal verification to those programs which have been well designed, carefully annotated, and validated by prototyping to be correct with respect to the given sets of test data. The verifier is used mainly to make sure that the programs are correct with respect to all possible inputs, and detect the subtle problems which have not been detected by the validation. The example program mentioned earlier is a program of this kind. To describe the use of RVF
and illustrate the user interface, we now use RVF to verify this program.

First, we run RVF in batch-mode, and use it as an automatic verifier. Thus, we choose the suitable options from the prove-option-menu, and invoke RVF with the root state (i.e. define-key) of the program. Note that, normally, we do not expect the verifier to succeed immediately. The purpose of such a batch-style verification is to find out the set of problematic transitions and extract the unproven component VCs. Thus, we may initialize GKB with the full PKB, but choose strong constraints to force the theorem prover to terminate quickly.

In an earlier experiment with this program, RVF processed 12 transitions. Since one of them was verified by a non-interference test (i.e. it was determined that the action of the transition did not affect the assertions), a total of 11 VCs were deduced. Among them, 5 were simplified to true immediately. The remaining 6 VCs were decomposed by the natural deduction controller into 10 conjectures. Two of them were reduced to true by the simulator; One was proved by the FIP; and another four were proved by the BIP. The entire process took about 2 minutes; most of the time was spent in displaying the intermediate steps and results. RVF terminates with the following summary:

<table>
<thead>
<tr>
<th>FAILED IN VERIFYING define-key STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALLOCATED 12 TRANSITIONS</td>
</tr>
<tr>
<td>DEDUCED 10 CONJECTURES:</td>
</tr>
<tr>
<td>PROVED 7 CONJECTURES:</td>
</tr>
<tr>
<td>2 by SIMPLIFIER, 1 by FIP, 4 by BIP</td>
</tr>
<tr>
<td>UNVERIFIED TRANSITIONS:</td>
</tr>
<tr>
<td>define-kdata-2-transition-1</td>
</tr>
<tr>
<td>define-kdata-3-transition-1</td>
</tr>
<tr>
<td>define-kdata-2-transition-1-vc-1</td>
</tr>
<tr>
<td>define-kdata-3-transition-1-vc-1-1-2</td>
</tr>
<tr>
<td>define-kdata-3-transition-1-vc-1-3</td>
</tr>
</tbody>
</table>

It has been determined that the conjecture valid-concat-kdata (see Fig. 2) is useful for proving the unproven conjectures produced earlier for verifying the define-key state. We now illustrate how to prove this conjecture and make it reusable. With the proof management facility, we created a proof-object as given below.

```
proof valid-concat-kdata
proof-origin {define-key}
proof-export {valid-concat-kdata}
proof-rules
kb (str-char-rel-1) any-rule {kb-axiom}
\forall (p:st) \forall (ch) (ch \in p \Rightarrow \text{char}(ch))
kb concat-def1 any-rule {kb-axiom}
\forall (x:char,p1:st,p2:st) ((x \in p1 \land x \in p2) \Rightarrow x \in \text{concat}(p1, p2))
kbr concat-def2 any-rule {kb-axiom}
\forall (x:char,p1:st,p2:st) ((x \in \text{concat}(p1, p2) \Rightarrow (x \in p1 \lor x \in p2))
kbr concat-prop1 any-rule {kb-axiom}
\forall (p1:st,p2:st) \forall (p1 = "", p2 = ") \Rightarrow \text{concat}(p1, p2) = ""
kbr concat-prop2 any-rule {kb-axiom}
\forall (p1:st,p2:st) \forall (p1 = "," \land p2 = ") \Rightarrow \text{concat}(p1, p2) = ""
kbr empty-st-prop1 any-rule {kb-axiom}
\forall (ch) ch \notin ""
proof-conjectures
conjecture valid-concat-kdata: backward-implication
{valid-kdata-dfl}, {valid-kdata-dfl}
predicate \forall (x,y) \forall (valid-kdata(x) \land valid-kdata(y) \Rightarrow valid-kdata(concat(x,y)))
conjecture valid-kdata-dfl: forward-implication
{valid-kdata}
predicate \forall (st) \forall (valid-kdata(st) \Rightarrow
(st(st) \land st \neq"
\forall (ch) (ch \in st \Rightarrow \text{char-greater}(ch, \#2))
\forall (ch) (ch \in st \Rightarrow \text{char-less}(ch, \#7)))
proving-strategies (variant-checking)
conjecture valid-kdata-dfl: backward-implication
{valid-kdata}
predicate \forall (st) \forall (string(st) \land st \neq"
\forall (ch) (ch \in st \Rightarrow \text{char-greater}(ch, \#2))
\forall (ch) (ch \in st \Rightarrow \text{char-less}(ch, \#7))
\Rightarrow valid-kdata(st))
proving-strategies (variant-checking)
```

Fig. 2. A user created proof-object

With this proof-object and by initializing the WKB with only the set of term-rewriting rules of the PKB, the prover proved the conjecture (valid-concat-kdata) very efficiently. We note the following features.

1. In proving a conjecture or rule, the prover does not use a definition or a conjecture as a hypothesis unless its name is included in the using-set of the conjecture or rule (designated by '{ }'). For example, since the using-set of the conjecture valid-concat-kdata does not contain valid-kdata, then the definition valid-kdata (see Section 2) is not used as a hypothesis in order to prove valid-concat-kdata. Instead, it is used as a hypothesis in order to prove the conjectures valid-kdata-dfl and valid-kdata-dfl2.

2. For a backward-implication rule, only those literals that are contained in its consequent can be resolved or paramodulated upon in the deduction. For example, when valid-kdata-dfl2 is used in provingvalid-concat-kdata, only the literal valid-kdata(st) is allowed.
to be resolved upon. Those literals contained in the antecedent of the rule are "locked". This stipulation is related to the partial set-of-support strategy [15] of the hierarchical deduction.

3. After the conjecture is proved, RVF will record the set of rules necessary to the proof into the supporting-set of the conjecture. This set can be used for two purposes. One is to detect the possible invalidity of the proof-object when a change occurs in the environment. The other one is to efficiently replay a proof of the conjecture.

4. As shown in Fig. 2, valid-concat-kdata has been included in the set of proof-exports. Then after the proof-object has been verified to be valid, RVF will add valid-concat-kdata into GKB when the proof-object is loaded, so it will be usable globally. With the addition of this lemma, RVF succeeded in verifying the entire define-key state in a later batch-mode running.

We point out that our sample problem is too simple to illustrate the general use of RVF. Usually, besides proving reusable lemmas, one may need to prove some unproven conjectures separately by editing each of their corresponding proof-files. These proof-files will then be used when RVF is invoked later for verifying the entire program.

8 Remarks

We have described the main components of the mechanical verifier RVF, and illustrated its usage by verifying a piece of Reacto program. Our approach emphasizes the incremental development of proofs, the reuse of lemmas, the flexible use of the knowledge base rules, and the interaction with the prover. We have tried to make RVF more powerful and efficient in order to release the user from proving simple theorems. Most of these ideas have been investigated by previous research in formal verification [4, 8, 11, 14]. Our work is mainly to develop these ideas for supporting the design of reliable reactive systems.

There is a number of important issues not addressed in this report. They include inductive proofs, knowledge-base development, soundness of verification results, special inference problems associated with polymorphic type constructors and set-theoretic data-types, etc.

The VCG and the proof manager are written in Repfine [1]. The theorem prover is written in Common-lisp. Terms maintained by the prover are implemented with a term-integration technique [16]. This technique helps improve greatly the efficiency of the KB manager and the dependency maintenance procedure.

References