Combining formal verification and conformance testing for validating reactive systems

Vlad Rusu * †
IRISA/INRIA, Campus de Beaulieu, F-35042 Rennes Cedex, France

SUMMARY
This paper presents a combination of verification and conformance testing techniques to support the formal validation of reactive systems. The idea is to use symbolic test selection techniques to extract subgraphs \textit{(components)} from a specification, and to perform the verification on the components rather than on the whole specification. Under reasonable sufficient conditions, this constitutes a sound compositional verification technique, in the sense that a property verified on the components also holds on the whole specification. This may considerably reduce the global verification effort. Moreover, once verified, a component forms the basis of an adequate test case, i.e. when executed on an implementation, it will not issue false positive or negative verdicts with respect to the verified properties. The approach has been implemented using the STG test selection tool and the PVS theorem prover. It is demonstrated here on a smart-card application: the Common Electronic Purse System. Copyright © 2003 John Wiley & Sons, Ltd.

KEY WORDS: formal verification; conformance testing; electronic purse

1. INTRODUCTION

Formal verification and conformance testing are two well-established approaches for validating software systems. In \textit{verification}, a formal specification of the system is proved correct with respect to some higher-level \textit{requirements}. In \textit{conformance testing} [1] the external, observable traces of a black-box implementation of the system are compared with those of its formal specification, according to a \textit{conformance relation}. For validating critical software systems, verification and conformance testing play complementary roles: the former ensures that the operational specification meets its requirements,
while the latter checks that the final implementation of the system conforms to its specification. Thus, through verification and testing, a formal connection between the system’s initial requirements and the final running system can be established.

There is a lot of interest in formal verification from researchers and, recently, formal verification has started to penetrate the industry. More recently, conformance testing (and other forms of testing) have become a topic of interest to the verification community, as a practical means for software validation. This interaction has resulted in new algorithms, and tools for testing based on verification technology (mainly model checking) have emerged [2,3]. However, a tight integration of verification and testing techniques that exploits their complementarity (along the lines discussed at the beginning of this section) is, in the author’s opinion, an open research direction, and one with practical relevance.

Indeed, verification without subsequent testing of the final system is hardly imaginable. Conversely, verification prior to conformance testing is also crucial, as shown by the following scenario. Assume that, for example, an independent third-party testing laboratory has to test the conformance of a black-box implementation $I$ of a system developed by a software company, with respect to a standard provided by an official normalisation body. The standard includes a large state machine $S$ and some requirements $R$ describing what $S$ is supposed to do. Now, assume that the standard is inconsistent, i.e. the specification $S$ violates $R$. (As standards are large and complex documents, this can easily happen.) If $I$ does conform to $S$, conformance testing will issue a ‘Pass’ verdict, even though $I$ may violate the initial requirements $R$. That is, an actually erroneous implementation may go undetected. If, however, testing reveals that $I$ does not conform to $S$, it will issue a ‘Fail’ verdict, even though $I$ might actually satisfy the initial requirements. But, in conformance testing, the specification is typically assumed to be correct, thus, the implementation (and its developers!) may be wrongly blamed for the error. These problems arise from the fact that $S$ violates $R$. Clearly, the specification must be formally verified before any testing based on it is performed.

Verifying a large (typically, an infinite-state) specification is difficult and is rarely done in practice; and test generation from such specifications is also a costly operation. This paper proposes an approach that integrates the verification and test generation efforts in one common task. In a first step, symbolic test generation techniques are employed to compute a set of components (subgraphs) of the specification $S$, and the requirements are verified on the components. Once verified, the components form the basis of adequate test cases: if a test case discovers a difference between $S$ and $I$ with respect to a verified requirement, it can only be the implementation’s fault; and the risk of not detecting a violated requirement is reduced. Moreover, under reasonable sufficient conditions, a requirement verified on the components also holds on the whole specification as well. This may considerably reduce the verification effort as components are typically much smaller than the whole specification.

The approach has been implemented using the STG test generation tool [4] for extracting components and the PVS theorem prover [5] for verifying them. This paper reports on experience with the approach on the CEPS (Common Electronic Purse System) specification [6]. The properties verified are invariants involving existential and universal quantifiers over unbounded domains, and the specification is an infinite-state, extended state machine with about 40 variables of complex record and parametric-size array types, and about 100 transitions operating on the variables. By contrast, the

---

§See also Section 7. Any set of references in this field is necessarily incomplete.
components extracted with STG have less than ten transitions each and affect a small subset of the variables. This is small enough to be dealt with efficiently using theorem proving. Because of the infinite-state nature of the system, an approach based exclusively on model checking [7–9] cannot solve this problem.

The rest of the paper is organised as follows. Section 2 presents the basics of conformance testing and symbolic test generation using a simple example. In Section 3 the verification of invariants with PVS is presented. Section 4 presents the results that allow reduction of the correctness of a specification to that of one or several of its components. Section 5 summarizes the validation methodology. Section 6 reports on an experiment with the CEPS case study, and Section 7 provides conclusions together with a discussion of related work. The CEPS specification, the components extracted with STG, their translation to PVS, and the PVS proofs are available at http://www.irisa.fr/vertecs/Equipe/Rusu/CEPS/.

2. SYMBOLIC TEST GENERATION

Symbolic test generation is a program-synthesis problem. Starting from the formal specification of a system under test and from a test purpose describing a set of behaviours to be tested, compute a reactive program (the test case) which (1) attempts to control a black-box implementation of the system towards satisfying the test purpose and (2) observes the external traces of the implementation for detecting non-conformances between implementation and specification.

The model used here for specifications, implementations, test purposes, and test cases is the Input–Output Symbolic Transition Systems (IOSTS) model, a variant of the Lynch and Tuttle I/O automata [10]. The model is first presented intuitively by means of a simple example; then, formal definitions are given.

2.1. The IOSTS model

Figure 1 depicts an IOSTS with four locations $l_0, l_1, l_2, l_3$, where $l_0$ is the initial location. The IOSTS has several transitions between locations, which are labelled with an action that can be an input, an output, or an internal action. By convention, the name of an input (respectively output) action ends with ? (respectively !).

For example, the IOSTS in Figure 1 has one input action inc?, two outputs ret!, exc!, and one internal action $\tau$. Input and output actions may carry messages, which, together with variables and parameters are the three distinct kinds of data the system manipulates. Intuitively, variables are data to compute with, parameters are symbolic constants, and messages are used to communicate with the environment. The data can be of any type, e.g. Boolean, integer, record, and array of fixed or parametric size. For example, the IOSTS in Figure 1 has two variables $x, y$, one parameter $p$, and one message $m$, all of integer type.

Each transition is also decorated with a guard and a set of parallel assignments that may involve any of the variables and parameters, but only those messages carried by that transition’s action. It is

---

Footnote:
The main difference with I/O automata concerns the input-completeness condition, which is not required in IOSTS. Another difference concerns the separation of symbolic data into variables, parameters, and messages, each playing a specific role.
assumed that all operations in guards and assignments are type-correct, e.g. all guards have the type Boolean, and only expressions of integer type may be assigned to integer variables. For example, the transition with origin $l_1$ and destination $l_2$ has guard $m \geq 0$, action $\text{inc}\(m\)$ carrying message $m$, and assignment $x := x + m$.

**Definition 1. (IOSTS)** An IOSTS is a tuple $\langle D, \Theta, Q, q^0, \Sigma, T \rangle$ where

- $D$ is a nonempty, finite set of typed data, which is the disjoint union of a set $V$ of variables, a set $P$ of parameters and a set $M$ of messages. For each $d \in D$, $\text{type}(d)$ denotes the type of datum $d$;
- $\Theta$ is the initial condition, a Boolean expression on $V \cup P$;
- $Q$ is a nonempty, finite set of locations;
- $q^0 \in Q$ is the initial location;
- $\Sigma$ is a non-empty, finite alphabet, which is the disjoint union of a set $\Sigma^i$ of input actions, a set $\Sigma^o$ of output actions, and a set $\Sigma^{int}$ of internal actions. For each action $a \in \Sigma^i \cup \Sigma^o \cup \Sigma^{int}$, there is a (possibly empty) tuple of types $\text{sig}(a) = \langle \vartheta_1, \ldots, \vartheta_k \rangle$, called the signature of the action (the signature of an internal action $a \in \Sigma^{int}$ is the empty tuple);
- $T$ is a set of transitions. Each transition consists of:
  - a location $q \in Q$, called the origin of the transition,
  - an action $a \in \Sigma$ called the action of the transition,
  - a tuple of messages $\mu = \langle m_1, \ldots, m_k \rangle$ such that if $\text{sig}(a) = \langle \vartheta_1 \ldots \vartheta_k \rangle$ is the signature of action $a$, then, for all $i \in [1, k]$, $\text{type}(m_i) = \vartheta_i$,
  - a Boolean expression $G$ on $V \cup P \cup \mu$, called the guard of the transition,
  - a set of expressions $A$, called the assignments of the transition, such that for each variable $x \in V$ there is exactly one assignment in $A$, of the form $x := A^x$, where $A^x$ is an expression on $V \cup P \cup \mu$,
  - a location $q' \in Q$ called the destination of the transition.
2.1.1. Parallel composition

The parallel composition operation lets each system perform independently its internal actions and imposes synchronisation on the shared input/actions, during which messages are passed from output to input. There are no shared data between the two systems. Formally, for $j = 1, 2$, the two IOSTS $S_j = (D_j, \Theta_j, Q_j, q^0_j, \Sigma_j, T_j)$ with data $D_j$ and alphabet $\Sigma_j = \Sigma_j^1 \cup \Sigma_j^0 \cup \Sigma_j^{int}$ are compatible for composition if $D_1 \cap D_2 = \emptyset$, $\Sigma_1^1 = \Sigma_2^1$, $\Sigma_1^0 = \Sigma_2^0$, $\Sigma_1^{int} \cap \Sigma_2^{int} = \emptyset$, and each action $a \in \Sigma_1^1 \cup \Sigma_2^1$ (which also means $a \in \Sigma_1^0 \cup \Sigma_2^0$) has the same signature in both systems.

Definition 2. (Parallel composition) The parallel composition $S = S_1 \parallel S_2$ of compatible IOSTS $S_1, S_2$ is the IOSTS $(D, \Theta, Q, q^0, \Sigma, T)$ that consists of the following elements: $V = V_1 \cup V_2$, $P = P_1 \cup P_2$, $M = M_1 \cup M_2$, $\Theta = \Theta_1 \cup \Theta_2$, $Q = Q_1 \times Q_2$, $q^0 = (q^0_1, q^0_2)$, $\Sigma^1 = \emptyset$, $\Sigma^0 = \Sigma_1^0 \cup \Sigma_2^0$, $\Sigma^{int} = \Sigma_1^{int} \cup \Sigma_2^{int}$. The set $T$ of transitions of the composed system is defined as follows:

- for each transition $(q_1, a, \mu, G, A, q'_1) \in T_1$ with $a \in \Sigma_1^{int}$ and for each location of the form $(q_1, q_2)$ with $q_2 \in Q_2$, there is a transition of the form $(q_1, q_2), a, \mu, G, A, (q'_1, q'_2))$ in $S_1 \parallel S_2$ (and similarly for all $a \in \Sigma_2^{int}$);
- for each pair of transitions $(q_1, a, \mu_1, G_1, A_1, q'_1) \in T_1$, $(q_2, a, \mu_2, G_2, A_2, q'_2) \in T_2$ with $a \in \Sigma_1^1$, let $G_{1,2}$ (respectively $A_{1,2}$) denote the expression obtained by replacing in $G_2$ (respectively $A_2$) each message from $\mu_2$ by the corresponding, same-position message from $\mu_1$. Then, in $S_1 \parallel S_2$ there is a transition $(q_1, q_2), a, \mu_1, G_1 \land G_{1,2}, A_1 \cup A_{1,2}, (q'_1, q'_2))$ (and similarly when $a \in \Sigma_2^1$).

2.1.2. Informal semantics

A behaviour of an IOSTS starts in the initial location with values of the variables and parameters satisfying the initial condition, and proceeds by firing transitions, updating the variables according to the guards and assignments of the transitions fired\(^\dagger\), and exchanging messages with the environment through the corresponding input and output actions. The value of a variable stays the same between two assignments, and the value of a parameter is never modified. The value of a message is only relevant when firing a transition labelled by an action that carries that message: if the action is an input (respectively an output), the message carries the value from (respectively to) the environment. After the communication the value of the message is lost; thus, to be memorised it has to be assigned to a variable.

For example, a behaviour of the IOSTS represented in Figure 1 starts in location $l_0$ with some positive value for $p$ and the same value for $x$, fires the transition labelled by the $\tau$ internal action, assigns variable $y$ to 0, and reaches location $l_1$. Then, when an $inc$ input action carrying a message $m \geq 0$ occurs from the environment, the variable $x$ is increased by the value of $m$, and the control is now in location $l_2$. Next, the IOSTS performs the $ret!$ output action, which fires the transition with origin $l_2$ and destination $l_1$. The value of message $m$ is nondeterministically chosen to satisfy the guard of that transition, i.e. $m = x$. That is, the value of $x$ is sent to the environment by the $ret!$ output.

\(^\dagger\)Assignments without effect, e.g. $x := x$, are omitted from the graphical representations.
2.1.3. Formal semantics

More formally, a state is a pair \((l, v)\) where \(l\) is a location and \(v\) is a valuation for the variables and the parameters. An initial state is a state \((l_0, v_0)\) such that \(l_0\) is the initial location and \(v_0\) satisfies the initial condition. A valued action is a pair \(\langle a, w\rangle\) where \(a\) is an action and \(w\) is a valuation for the message(s) carried by the action. Note that the values of messages are not contained in the states, but in the valued actions. For convenience, internal actions are considered as valued actions carrying a tuple of values of length 0. For \(t\) a transition of the IOSTS, the transition relation \(\varrho_t\), is the set of triples \((s, \alpha, s')\), where \(s = (l, v), s' = (l', v')\) are states and \(\alpha = \langle a, w\rangle\) is a valued action such that \(l, l',\) and \(a\) are respectively the origin, the destination, and the action labelling \(t\); the values of variables, parameters and actions defined by \(v, w\) satisfy the guard of the transition \(t\); and the valuation \(v'\) is obtained from \(v, w\) by the assignments of transition \(t\). The global transition relation \(\varrho\) is \(\varrho = \bigcup_{t \in T} \varrho_t\), where \(T\) denotes the set of transitions of the IOSTS.

Definition 3. (Behaviour) A behaviour \(\beta : s_1 \alpha_1 s_2 \alpha_2 \cdots \alpha_{n-1} s_n\) is a sequence of alternating states and valued actions, such that \(s_1\) is an initial state and such that for \(i = 1, \ldots, n - 1\), the triple \(\langle s_i, \alpha_i, s_{i+1}\rangle\) is in the transition relation \(\varrho\).

A run is the subsequence of a behaviour obtained by removing all the actions.

Definition 4. (Run) Given a behaviour \(\beta : s_1 \alpha_1 s_2 \alpha_2 \cdots \alpha_{n-1} s_n\), the run of \(\beta\) is the subsequence \(\rho : s_1 s_2 \cdots s_n\) of \(\beta\) containing only the states of \(\beta\).

A trace is the subsequence of a behaviour containing only what is externally visible, that is, states and internal actions are removed from the sequence.

Definition 5. (Trace) The trace of a behaviour \(\beta\) is the subsequence \(\sigma : \alpha_1 \cdots \alpha_k\) of \(\beta\) containing only the valued inputs and valued outputs of \(\beta\).

2.1.4. Conformance relation

This is what is being tested for in conformance testing. There are several variants (see, for example, [11–13]) of this relation. Here such a relation between IOSTS is formalised.

First, for \(\beta\) a behaviour of an IOSTS \(I\) and \(\rho\) a run (respectively \(\sigma\) a trace) of \(\beta\), \(I\) after \(\beta\) denotes the last state of \(\beta\), and, for a trace \(\sigma\), \(I\) after \(\sigma\) denotes the set of states \(\{I\text{ after }\beta \mid \sigma\text{ is a trace of }\beta\}\). That is, \(I\) after \(\sigma\) is the set of states in which the IOSTS \(I\) may be after the observable trace \(\sigma\). Because of internal actions that are hidden in the trace, a black-box system \(I\) may be in any of those states, but the exact one is not known.

Next, for a state \(s = \langle l, v\rangle\), \(out(s)\) denotes the set of valued outputs \(\alpha = \langle a, w\rangle\) for which there exists a state \(s'\) such that \((s, \alpha, s')\) is in the transition relation \(\varrho\). For \(S\) a set of states let \(out(S) = \{out(s) \mid s \in S\}\). That is, \(out(S)\) is the set of valued outputs that the IOSTS can emit when it is in a state from the set \(S\). These notations allow the following definition to be introduced.

Definition 6. (Conformance) Given \(I, S\) two IOSTS, \(I\) is said to conform to \(S\) if, for each trace \(\sigma \in traces(S)\): \(out(I\text{ after }\sigma) \subseteq out(S\text{ after }\sigma)\).
That is, after every observable sequence of valued actions of the IOSTS $S$, which plays the role of the specification, the next possible valued outputs observed on the black-box implementation $I$ are among those allowed by the specification. This is similar to the $ioco$ relation [13] except that deadlocks are not observable.

**Example 1.** Consider the IOSTS $S$ depicted in Figure 1 and $I$ depicted in Figure 2 (the locations of the $I$ are not represented). Then, $I$ does not conform to $S$ because, after an $inc?$ input carrying a negative value $m$, $S$ cannot emit the $exc!$ output: the guard $y < 0$ does not allow it. If this guard is changed to $y \geq 0$ then $I$ conforms to the new specification.

### 2.2. Test generation

In practice, following the standard [1] test cases are generated using a specification (e.g. the IOSTS depicted in Figure 1) and a test purpose (e.g. the IOSTS depicted in Figure 3). A test purpose is another IOSTS that gives an abstract description of a part of the system that will be selected for testing.

For example, Figure 3 depicts a test purpose that targets (accepts) the behaviours of the specification in Figure 1 that end with the $exc!$ action, except for those during which an $inc?$ input carrying a positive value occurs. Note that it is not necessary to give all the details, e.g. behaviours of the specification after which the $exc!$ action occurs: these are automatically computed by STG. The resulting test case is depicted in Figure 4.

Intuitively, the test selection algorithm [14] selects the largest subgraph of the specification that (1) includes the subgraph of the test purpose that leads to $Accept$ and (2) excludes the subgraph of the test...
purpose that leads to Reject. Finally, on the resulting IOSTS, the locations labelled Accept are renamed Pass, and the test case is completed to include all inputs that are not predicted by the specification—which lead to a new location Fail. Pass and Fail locations correspond to verdicts on the conformance between the specification and an implementation on which the test case is executed. Reaching the Fail location means that an error (a non-conformance) was detected, while reaching Pass means that the test purpose was satisfied and no errors were detected**.

The test generation algorithm, including translation of the resulting IOSTS into executable code (currently, C++ and Java are supported) and execution on an implementation are implemented in the STG tool [4]. The theoretical properties underlying STG guarantee the verdict reached on the conformance of the implementation with its specification is always correct. (Of course, this says nothing about the correctness of the specification with respect to its requirements; this is the object of verification).

2.2.1. Interacting with STG

A typical interaction with STG consists of loading and compiling a file that contains a specification and one or several test purposes described as IOSTS. A command-line interface allows one to enter the commands for test generation (e.g. compute crossproduct of test purpose and specification, eliminate subgraph leading to Reject, add verdicts), either one by one or several in sequence. The generated test cases (as well as the specification and the test purposes) can be visualised using the DOTTY graph visualiser from AT&T.

**There is also a third verdict, Inconclusive. It represents the negation of both Pass and Fail, i.e. non-conformances were not detected, but the test purpose cannot be satisfied.
The generation of a test case is a trial-and-error process. Assume, for example, that one wants to test that a sequence of events occurs in a given order. For this, the user starts by giving an initial test purpose, which is a sequence of transitions labelled by the given sequence of events. Then, STG generates a test case and the user examines the result visually. Each test generation step is fast (it is essentially a traversal of the control graph of the crossproduct between test purpose and specification); thus, it can be repeated several times until the user is satisfied with the result. At each step, the user may refine the test purpose by adding more details from the specification and rejecting undesired parts by means of Reject locations. It is always possible to select a given connected subgraph of the specification by providing enough detail in the test purpose. In the author’s experience, this trial and error process converges rapidly for applications whose control graph is of the order of hundreds of transitions.

2.3. Running the test

The IOSTS TC in Figure 4 can be employed to test, for example, an implementation I whose behaviour is shown in Figure 2 for conformance to the specification S given in Figure 1. The execution model of a test case on an implementation is that of parallel composition (see Definition 2). The implementation being a black box, there are no shared variables between it and the test case. Note that the inputs of the implementation are outputs of the test case and reciprocally. Note also that the transition labelled y < 0 in the test case is never fireable, thus, the Pass location is not reachable. That is, the test purpose cannot be satisfied, but valuable information is still obtained if the Fail location is reached, i.e. a non-conformance is detected. By executing TC on I the Fail location is reached and a Fail verdict is given, meaning that I does not conform to S.

Now, assume that the IOSTS S is the specification of a system that (among other requirements) should satisfy the property P: an exc! action never occurs. The specification S depicted in Figure 1 does satisfy this requirement. However, if the guard y < 0 on the transition from l3 to l0 is mistyped y ≥ 0, the resulting specification S’ does not. Then, the test case TC’ obtained from the specification S’ also contains the erroneous guard, and, as a consequence, the implementation I (which does not meet the requirement P) passes the test TC’.

Because of the error in the specification, an error in the implementation goes undetected. For large specifications it is not unreasonable to suppose that such errors may happen, and the above example (or the dual scenario presented in Section 1) demonstrates the need to verify specifications prior to test generation.

3. VERIFYING INVARIENTS OF AN IOSTS MODEL USING PVS

In this section the invariant-strengthening technique for proving invariants of an IOSTS model, and how to achieve this using the PVS theorem prover are described††.

3.1. Inductive and non-inductive invariants

Informally, a state predicate is an invariant if it holds at every state of every run. A state predicate \( \varphi \) is inductive if it holds initially and, for every state \( s \), by using only the information that \( \varphi \) holds at \( s \), it is possible to prove that \( \varphi \) also holds at all successors \( s' \) of \( s \) through the transition relation.

For example, in the IOSTS \( S \) represented in Figure 1, it is not hard to check that the predicate \( x \geq 0 \) is inductive. Indeed, it is true initially, and from any state satisfying \( x \geq 0 \), each transition leads to a state satisfying the predicate.

Any inductive predicate is also an invariant, but the converse is not true: for example, consider the predicate \( pc = l_3 \supset y \geq 0 \) (that is, whenever control is at location \( l_3 \), \( y \) is positive). It will be proved that this predicate is an invariant, but it is not inductive: by knowing only that it is true before transition labelled \( inc? \) from \( l_1 \) to \( l_3 \), it cannot be inferred that it is still true after the transition. To prove this, additional information is required; here, that \( x \geq 0 \) is an invariant.

3.2. Invariant strengthening

The additional information required to prove that a given predicate is an invariant can be obtained by invariant strengthening. To define this process formally one needs to recast some well-known definitions into the framework of IOSTS.

**Definition 7.** (Weakest pre-condition) For a transition \( t \) and a state predicate \( \varphi \), the predicate \( \widehat{\text{pre}}_t(\varphi) \) characterises all the states \( s \) from which, after taking transition \( t \), the predicate \( \varphi \) holds: \( \forall s'. \forall \alpha. \rho_t(s, \alpha, s') \supset \varphi(s') \).

The weakest pre-condition is also defined globally, i.e. for an IOSTS whose set of transitions is \( T \):
\[
\widehat{\text{pre}}(\varphi) = \bigwedge_{t \in T} \widehat{\text{pre}}_t(\varphi).
\]

**Definition 8.** (Inductive predicate) A predicate \( \varphi \) is inductive for an IOSTS \( S \) if \( \varphi \) holds in all initial states of \( S \), and the implication \( \varphi \supset \widehat{\text{pre}}(\varphi) \) is valid.

If \( \varphi \) is inductive then it is an invariant: from the fact that \( \varphi \) holds in a given state \( s \), and from \( \varphi \supset \widehat{\text{pre}}(\varphi) \), it is obtained that \( \widehat{\text{pre}}(\varphi) \) holds at \( s \), which amounts to saying that \( \varphi \) holds at all immediate successors \( s \) by the transition relation. By combining this observation with the fact that \( \varphi \) holds initially, a proof by induction is obtained that \( \varphi \) holds in all reachable states, i.e. it is an invariant.

If \( \varphi \) is not inductive, one can take \( \widehat{\text{pre}}(\varphi) \) (or a stronger predicate) and try to prove that it is inductive. The process continues until, for some natural number \( n \), \( \widehat{\text{pre}}^n(\varphi) \) (or some stronger predicate) has been proved inductive, i.e. for a predicate \( \psi \) and some natural number \( n \) such that \( \psi \supset \widehat{\text{pre}}^n(\varphi) \), it is the case that \( \psi \supset \widehat{\text{pre}}(\psi) \). Then, \( \psi \) is an invariant, and so is \( \widehat{\text{pre}}^n(\varphi) \) (and ultimately \( \varphi \)).

The process is not guaranteed to terminate if \( \widehat{\text{pre}}(\varphi) \) is systematically chosen to be the ‘next’ candidate for inductiveness; otherwise, this would constitute an automatic procedure for proving
invariants on infinite-state systems, which is an undecidable problem. The key point that allows convergence is the ability of the user to formulate assertions stronger than $\bar{\text{pre}}(\varphi)$. The $\bar{\text{pre}}(\varphi)$ assertions are computed by the theorem prover, and presented to the user in a way that makes it convenient to guess stronger assertions (by deleting hypotheses or conclusions from the PVS subgoal).

### 3.3. PVS

The PVS system consists of an input language, a typechecker, and an interactive prover. The input language is typed higher-order logic with a rich type system including simple types such as Booleans, enumerations, integers, and records, and more complex function types, subtypes, dependent types, and abstract datatypes. Having such an expressive language makes it easy to specify, for example, concurrent programs in a natural way, very close to a programming language.

The drawback is that typechecking the input language is undecidable. In fact, PVS transforms this apparent weakness into an actual strength, because whenever the typechecker cannot decide whether an expression is type-correct it generates a TCC (type-correctness condition). PVS declares unsound a theory where some TCCs are left unproved. Most can be discharged automatically, and those that cannot often point to subtle errors in the specification.

A PVS proof is a tree, the root of which is the theorem being proved. The leaves of the tree are called pending subgoals. A proof proceeds as a sequence of commands, each of which transforms the proof tree by either proving a pending subgoal or by replacing a pending subgoal by a new set of pending subgoals.

There are many proof commands, from propositional and first-order logic commands, to decision procedures and heuristic quantifier instantiation, all of which can be combined into high-level, user-defined proof strategies.

#### 3.3.1. Invariant strengthening in PVS

To perform invariant-strengthening in PVS, a specific proof strategy attempts to prove that a given predicate $\varphi$ is inductive. If this is the case the strategy succeeds and the proof is done. Otherwise, the strategy fails, and leaves one or several pending (unproved subgoals). Each pending subgoal is a PVS representation of a predicate of the form $\bar{\text{pre}}_t(\varphi)$, for $t$ a transition that does not preserve the validity of $\varphi$ (i.e. a transition $t$ such that $\varphi \supset \bar{\text{pre}}_t(\varphi)$ is not valid). By examining the subgoal, the user can formulate an auxiliary lemma $L$ stating that $\bar{\text{pre}}_t(\varphi)$, or a stronger predicate, is inductive, and prove this lemma by the same invariant-strengthening process. (Of course, this may lead to recursively proving a whole sequence of auxiliary lemmas, whose length depends on the user’s creativity.)

When the lemma $L$ is finally proved, using it allows PVS to settle the corresponding pending subgoal. Eventually, all pending subgoals are settled in this manner, which concludes the proof of invariance of $\varphi$.

For simple systems such as the one depicted in Figure 1 or for academic case studies such as [15,16] this global verification approach is good enough. For example, each of the two academic case studies amounts to proving a few dozen auxiliary lemmas. For larger case studies the effort of performing invariant strengthening may become prohibitive, because the user may be overwhelmed...
by the number of pending subgoals to prove and lemmas to formulate. This is why a compositional verification technique is introduced in the next section.

4. VERIFICATION BY COMPONENTS

This section shows how to reduce the verification of an IOSTS specification to the verification of a number of its components, which are particular subgraphs of the specification. Under reasonable sufficient conditions, a safety property verified on the components also holds on the whole specification.

The verification is performed using PVS and invariant strengthening (see Section 3). The components are selected using the STG tool (see Section 2) from the control graph of the specification, using adequately chosen test purposes.

Definition 9. (Control graph) Given an IOSTS $S$, let $L$ be the set of locations of $S$ and $T$ the set of its transitions. The control graph $G(S)$ is the labelled graph $(L, T)$, where every edge in $T$ is labelled by the guard, action, and assignments of the corresponding transition of $S$.

For convenience, the nodes of $G(S)$ are called locations and the edges are called transitions. A path in a graph is a sequence of contiguous locations and transitions. A location of a subgraph $G'$ of a graph $G$, which is the destination of a transition whose origin is not in $G'$, is called an entry point of $G'$.

Definition 10. (Component) Given an IOSTS $S$ and $l$ a location of $S$, a component of $S$ with root $l$ is a subgraph $C$ of the graph $G(S)$ such that:

- $l_0$ is the only entry point of $C$; and
- there exists a path in $C$ from $l_0$ to every node of $C$.

Components (in the sense of the above definition) allow one to model the control graphs of functions or procedures that are inlined inside the graph of a main (calling) function. Another feature that is naturally described by this notion of component is that of (macro) transitions in the SDL language [17], which are graphs of (micro) transitions in the IOSTS sense.

A component can also be seen as part of a larger system that performs some specific function; for example, in the CEPS system (see Section 6) there is a component for inserting new currencies in the electronic purse, another one to query the existing currencies, and a third component to log all operations. It is reasonable to suppose that the control flow of such a function always starts in the same control node and is connected as imposed by Definition 10.

Example 2. Figure 5 depicts two components of the IOSTS $S$ from Figure 1.

Definition 11. (Defined and used variables) Let $S$ be an IOSTS. A transition $t$ of $S$ uses a variable or parameter $v$ if $v$ is present in the guard of $t$, or in the right-hand side of an assignment of $t$, or in the left-hand side of an assignment of $t$ within the index of an array. Transition $t$ defines a variable $v$ if $v$ appears in $t$ in the left-hand side of an assignment except in an array index.

Example 3. A transition with guard $x > 3$ and assignments $x := x + 1$ and $A[i] := 3$ defines the variables $A$ and $x$, and uses the variables $i$ and $x$.  

Copyright © 2003 John Wiley & Sons, Ltd.  
Definition 12. (Variables, parameters of a component) Given \( C \) a component of an IOSTS \( S \), the variables and parameters of \( S \) that are defined (used) by the transitions of \( C \) are said to be defined (respectively used) by \( C \). The variables that are defined only by transitions of \( C \) are said to be exclusively defined by \( C \).

Example 4. The component \( C_1 \) (Figure 5) uses variable \( x \) and defines variable \( y \), but \( y \) is not exclusively defined by \( C_1 \) as it is also defined outside \( C_1 \) (see Figure 1). In contrast, \( x \) is exclusively defined by the component \( C_2 \).

A component is not an IOSTS by itself, but it can be transformed into an IOSTS by giving it an initial condition and taking its root as the initial location.

Definition 13. (IOSTS obtained from component and initial condition) Given a component \( C \) of an IOSTS \( S \) and a predicate \( Q \) on the variables and parameters of \( C \), \( \text{iosts}(C, Q) \) denotes the IOSTS whose graph is the graph of \( C \), whose initial location is the root of \( C \), and whose initial condition is \( Q \).

Example 5. Let \( Q_1 : x \geq 0 \). The component \( C_2 \) of Example 2 is transformed into the IOSTS \( \text{iosts}(C_2, Q_1) \) with initial location \( l_1 \) and initial condition \( Q_1 \).

In the rest of this section two propositions are stated and proved, which provide sufficient conditions under which an invariant proved on a component also holds on the whole specification. Proposition 1 deals with the case of properties that may only involve data which are not modified outside the component (i.e. syntactically, the formula expressing the property contains only parameters and variables exclusively defined in the component in the sense of Definition 12).

This is not enough, in general, and Proposition 2 provides another set of sufficient conditions to cover the case of properties that may also involve variables defined outside the component, provided a global invariant on the variables is known to hold. The notation \( S \models Q \) is used for ‘The predicate \( Q \) is an invariant of the system \( S \)’ and \( S \models Q \) is used for ‘\( Q \) holds in the initial states of \( S \)’.

Proposition 1. Let \( C \) be a component of an IOSTS \( S \), let \( V' \) be the set of variables that are exclusively defined by \( C \), and \( P' \) be the set of parameters used by \( C \). Let \( Q \) be a property involving only variables and parameters in \( V' \cup P' \), such that \( S \models Q \) and \( \text{iosts}(C, Q) \models Q \). Then, \( S \models Q \) also holds.

For proving Proposition 1 the following result is needed.
Claim. If \( t \) is a transition of an IOSTS, and \( Q \) is a predicate involving only variables that are not defined by \( t \), and parameters, then \( Q \supset \widetilde{p\mathcal{E}}_t(Q) \) holds.

The claim can be interpreted as follows: for transition \( t \) to modify the truth value of a predicate \( Q \), \( t \) must define (i.e. modify) the variables of \( Q \). That is, the guard of \( t \) alone cannot have the effect of changing the truth value of \( Q \).

Proof of the Claim. Let \( X \) be the set of variables that are defined by \( t \), \( Y \) the set of (variables and parameters) that are used, but not defined by \( t \), and \( \mu \) the messages that are carried by the action of \( t \). Let \( G(X, Y, \mu) \) denote the guard of \( t \). Since \( t \) does not define \( Y \), the assignments of \( t \) are of the form \( X := A(X, Y, \mu) \), \( Y := Y \). Then, using Definition 7, \( \widetilde{p\mathcal{E}}_t(Q) \) can be written as

\[
\widetilde{p\mathcal{E}}_t(Q) := \forall X' \forall Y' \forall \mu. [(G(X, Y, \mu) \land X' = A(X, Y, \mu) \land Y' = Y) \supset Q(Y')]
\]

The universal quantification over \( Y' \) can be removed (i.e. \( Y' = Y \)), yielding:

\[
\widetilde{p\mathcal{E}}_t(Q) := \forall X'. \forall \mu. [(G(X, Y, \mu) \land X' = A(X, Y, \mu)) \supset Q(Y)]
\]

The implication \( Q \supset \widetilde{p\mathcal{E}}_t(Q) \) can be written—using the fact that \( Q \) is a predicate involving only variables and parameters in \( Y \), and that \( Y \) is free in \( \widetilde{p\mathcal{E}}_t(Q) \):

\[
Q \supset \widetilde{p\mathcal{E}}_t(Q) \supset \forall X'. \forall \mu. [(Q(Y) \supset ((G(X, Y, \mu) \land X' = A(X, Y, \mu)) \supset Q(Y)))]
\]

The expression inside \([\ldots]\) has the form \( Q \supset (E \supset Q) \), which is trivially true.

Proof of Proposition 1. Assume \( S \not\models \Box Q \). This means there exists a run \( \rho \) of \( S \) and a state \( s' \) on \( \rho \) such that \( s' \) violates \( Q \). Assume that \( s' \) is the first state on \( \rho \) where \( Q \) is violated, and let \( s \) be its immediate predecessor on the sequence \( \rho \) (\( s \) exists because \( Q \) holds at least in the initial states of \( S \)). Thus, there exists a transition \( t \) of \( S \) that is taken by \( \rho \) for going from \( s \) to \( s' \). Since the truth value of \( Q \) changes when this transition is taken, this means some variable involved in \( Q \) is modified by \( t \) (see the above claim; remember also that parameters cannot be modified). Since the only variables involved in \( Q \) are among the variables \( V' \) exclusively defined by \( C \), \( t \) must be a transition of \( C \). Hence, the run \( \rho \) can be split into two subsequences, \( \rho = \rho_1 \cdot \rho_2 \):

- \( \rho_1 \) is the prefix of \( \rho \) from the initial location of \( S \) to the last time it enters \( C \);
- \( \rho_2 \) is the suffix of \( \rho \) from the last time it has entered \( C \), to the faulty state \( s' \).

(Note that \( \rho_1 \) may be empty.) By definition of a component, the only way \( \rho \) can (re-)enter \( C \) is through the root of \( C \) (see Definition 10). A proof is now given of (1) that the first state \( s_0 \) of \( \rho_2 \) satisfies the initial condition \( Q \) of the IOSTS \( iossts(C, Q) \).

It has been shown that the transition \( t \) leading from \( s \) to \( s' \) is a transition of \( C \) (here, \( s' \) is the first state on \( \rho \) where \( Q \) is violated). Thus, every state before \( s' \) on this run, in particular, the first state \( s_0 \) of \( \rho_2 \), satisfies \( Q \), and (1) is proved. Now, by construction, after the run \( \rho_1 \), it is possible to execute the sequence \( \rho_2 \) and to reach the faulty state \( s' \). Hence, because of (1), the sequence \( \rho_2 \) is a run in \( iossts(C, Q) \). This means that \( s' \), the last state of \( \rho_2 \), is reachable in \( iossts(C, Q) \), and since \( Q \) is an invariant of \( iossts(C, Q) \), the predicate \( Q \) holds in \( s' \). A contradiction has been reached: the proof is done.

Copyright © 2003 John Wiley & Sons, Ltd.  
Below is an example of how Proposition 1 can be used to prove an invariant.

**Example 6.** Consider the component $C_2$ depicted in Figure 5. This component exclusively defines variable $x$. The predicate $Q_1 : x \geq 0$ is inductive on $iosts(C_2, Q_1)$, thus, it is an invariant of $iosts(C_2, Q_1)$. Thus, Proposition 1 can be applied to show that $Q_1$ is an invariant of the whole IOSTS $S$ of Figure 1.

A different technique is required for properties that involve variables defined outside a given component. For $l_0$ a location of an IOSTS, let $pc = l_0$ denote the predicate that characterizes all states $s = (l, v)$ such that $l = l_0$ and $v$ is an arbitrary valuation of the variables and parameters of the IOSTS.

**Definition 14.** (Predicate local to a component) A predicate $Q$ is local to component $C$ if $Q$ is of the form $pc = l \supset Q'$, where $l$ is a location of $C$, and $Q'$ is a predicate involving only variables and parameters defined or used by $C$.

**Example 7.** Let $C_1$ denote the component previously defined in Example 2. Then, the properties $pc = l_3 \supset y \geq 0$ and $pc = l_1 \supset y \geq 0$ are local to $C_1$. The predicate $pc = l_0 \supset x \geq 0$ is not local to $C_1$ because $l_0$ is not a location of $C_1$.

**Proposition 2.** Let $C$ be a component of $S$, and $Q'$ an invariant of $S$. Then, for any local property $Q$, if $iosts(C, Q') \models \Box Q$ holds, then $S \models \Box Q$ also holds.

**Proof.** The following observation will be used: a predicate local to a component is trivially true outside that component, i.e. if $l'$ is not a location of $C$, and $Q : pc = l \supset Q'$ is a predicate local to $Q$, then $l \neq l'$, thus $(pc = l' \supset Q) \equiv \text{true}$.††

Assume $S \not\models \Box Q$. This means there exists a run $\rho$ of $S$ and a state $s = (l', v)$ on $\rho$ such that $s$ violates $Q$. Since $Q$ is a local property of $C$, this means that the location $l'$ of $s$ is a location of $C$ (otherwise, by the above observation, one would have $s \models Q$). Then, just as in the proof of Proposition 1, the run $\rho$ can be split into two subsequences, $\rho = \rho_1 \cdot \rho_2$ (where $\rho_1$ may be empty):

- $\rho_1$ is the prefix of $\rho$ from the initial location of $S$ to the last time it enters $C$;
- $\rho_2$ is the suffix of $\rho$ from the last time it has entered $C$, to the faulty state $s$.

A proof is now given of (1) that the first state $s_0$ of $\rho_2$ satisfies the initial condition $Q'$ of $iosts(C, Q')$. By hypothesis, $S \models \Box Q'$ holds, and $s_0$ is a reachable state in $S$; thus this state satisfies $Q'$ and (1) is proved. Hence, from the state $s_0$ it is possible to execute the sequence $\rho_2$ as a run of $iosts(C, Q')$ and to reach $s$.

This means that $s$ is reachable in $iosts(C, Q')$, and, by $iosts(C, Q') \models \Box Q$, $s$ must satisfy $Q$, and a contradiction has been reached: the proof is done. □

**Example 8.** Below is a simple example of how Propositions 1 and 2 can be used together to prove invariants of the IOSTS $S$ represented in Figure 1, by reducing them to invariants of the components $C_1$ and $C_2$ of $S$ (see Figure 5).

††Proof. $pc = l' \supset Q$ is $pc = l' \supset (pc = l \supset Q')$, which can be rewritten (1) $(pc \neq l \lor pc \neq l' \lor Q)$. Now, either $pc$ equals $l$, in which case, using $l \neq l'$, the second disjunct of (1) is true; or $pc$ is not equal to $l$, in which case the first disjunct of (1) is true. In either case (1) is true.
Suppose one wants to prove that in $S$ the output $exc!$ never occurs. This amounts to proving that the guard $y < 0$ of the transition labelled $exc!$ is always false in location $l_3$, that is, to proving $S \models \square (pc = l_3 \supset y \geq 0)$. Let $Q_2$ denote the predicate $pc = l_3 \supset y \geq 0$. Then, $Q_2$ is a predicate local to component $C_1$ (see Definition 14), and Proposition 2 can be used to prove $S \models \square Q_2$. For this, a predicate $Q_1$ is required, such that (1) $S \models \square Q_1$ and (2) $iosts(C_1, Q_1) \models \square Q_2$. Now, in Example 6, by taking $Q_1 : x \geq 0$ and by using Proposition 1, $S \models \square Q_1$ has been proved. Then, it is not hard to show that $iosts(C_1, Q_1) \models \square Q_2$ holds as well ($Q_2$ is inductive over $iosts(C_1, Q_1)$). Thus, requirements (1) and (2) hold, and the proof of $S \models \square (pc = l_3 \supset y \geq 0)$ is done.

4.1. Discussion

A natural question that arises is whether Propositions 1 and 2 can be applied for the verification of parallel systems as well. The parallel composition of IOSTS is synchronisation on input/outputs with message-passing (see Definition 2 from Section 2), which only applies to IOSTS models that do not share variables. However, Propositions 1 and 2 are adapted for components that do share variables: the former refers to variables that are modified only in a given component, but may be accessed in reading by others, while the latter refers to variables that may be modified in more than one component. Clearly, these propositions are not adapted for reasoning on synchronously composed systems.

Alternatively, when applied to parallel composition with shared variables, the mechanisms defined in this section do not work either. For example, Proposition 2 does not hold because even if one proves an invariant on one component, some other component running in parallel with it may modify the value of a shared variable, thus, preventing the whole system from satisfying the invariant. The conclusion is that the mechanisms defined in this section are best suited for proving properties of components that are subgraphs of a specification, not for systems that consist of parallel components.

In Section 6 these mechanisms are applied for proving properties of an electronic purse system. As a result, the components on which interactive proofs actually have to be performed are about ten times smaller than the specification of the whole system. But first, the validation methodology is briefly recapped.

5. VALIDATION METHODOLOGY

5.1. Selecting components using STG

Components are selected using STG as described in Section 2. The user provides in a test purpose (a subgraph of) the graph of the component that he/she wishes to select. STG performs the selection by a forward exploration of the crossproduct of the test purpose with the specification, starting from a location of the latter (designated by the user) that will constitute the root of the selected component. Thus, by construction the two requirements for being a component (in the sense of Definition 10) are guaranteed: i.e. that for each selected node, there is a path in the component from the root to the node in question, and that the root is the only entry point in the component (the latter because STG never exits the selected component).

When the selection is done STG displays the result. If the user is not satisfied with what was selected (i.e. with respect to the applicability of the conditions for compositional verification (Propositions 1
and 2) he/she can refine the test purpose to include more detail. Test purposes typically employ the Reject location to specify, if necessary, what needs to be excluded from the component. When a component has been properly selected, the user checks if the conditions for compositional verification (Propositions 1 and 2) hold‡‡.

5.2. Verification using PVS

The components are then translated to PVS, and the verification is performed as explained in Section 3. The results are propagated to the whole specification using either Proposition 1 or Proposition 2.

5.3. Test execution using STG

Finally, after the verification is done, the user may ask STG to generate executable test cases in the form of Java or C++ programs, and execute the test cases on an implementation to obtain a verdict about the conformance of the latter with the (now verified) specification. Then:

- if testing gives a Fail verdict, and the difference between implementation and specification concerns a verified requirement, then it is clearly the implementation’s fault (that is, there are no false negative verdicts);
- if testing gives a Pass verdict, then the implementation did not violate any verified requirement during the testing phase (no false positives).

The above statements hold only for requirements that are invariance properties; and, in order to be detected by conformance testing, the requirements have to concern observable features of the system, such as absence of a given output.

6. CASE STUDY: AN ELECTRONIC PURSE SYSTEM

The CEPS [6] is a standard for creating multi-currency smart-card electronic purse systems. An electronic purse has an indeterminate number of slots, each of which corresponds to a currency and its respective balance. Thus, the slot type is a record, and the slots are contained in an array of slots of parametric size. The CEPS specifies, among others, functions that create, modify, or query the slots; a function for setting a reference currency; and another one for building a log of all operations on the card. In previous work [18] the author with others have built a detailed model of a significant portion of the CEPS as an IOSTS with 40 variables of complex record and array types and 92 transitions, and have generated and executed test cases on an implementation of the CEPS.

The specification itself was not verified, however; thus, there was no guarantee that the verdicts obtained were meaningful with respect to the specification’s requirements. Here, this doubt is eliminated: using STG together with PVS and the results from Section 4, the CEPS is verified in a compositional manner.

‡‡This checking can be automated; currently, it is not implemented in STG.
The verification of the Create/Update function is only outlined; more details are given for the substantially more involved verification of the Query function.

6.1. Create/update slots

This function consists of either creating a new slot for a currency, if the currency is not already in the array of slots; or of updating an existing slot, if the currency is already present in the array. The requirements for this function [6] are that all used slots contain different currencies. Essentially, this is achieved by not creating a new slot for an already existing currency. The requirement is expressed as a PVS invariant, and, instead of attempting to prove it on the whole specification, it is first proved on the component of the CEPS dedicated to creating and updating slots, and then, by Proposition 1, the requirement holds on the whole specification. Proposition 1 can be applied because the Create/Update component exclusively defines (Definition 12) the array of slots.

To select the Create/Update component, STG is used with a test purpose that describes a subset of the sequence of inputs and outputs required for creating/updating a slot. The test purpose has five transitions, and the selected component, whose internal details have been automatically selected by STG from the specification, has nine transitions, i.e. it is ten times smaller than the whole specification.

The component is then translated to PVS. The translation is automatic, except for the initial condition that has to be added to transform the component into an IOSTS (Definition 13) suitable for use with Proposition 1. The adequate initial condition is the property that all used slots contain different currencies. The PVS invariant expressing this same requirement is not inductive, but the subgoals left unproved by PVS suggest four relevant auxiliary invariants. The verification of the Create/Update function took about three days. By Proposition 1, the said requirement also holds on the whole specification.

6.2. Query

The requirement for this function [6] is that the card responds to the Query command with a sequence of slots; the slots can be returned in any order provided that each used slot is reported exactly once. In other words, the IOSTS specification of the CEPS must regard any implementation that chooses any particular order for reporting the used slots as being conformant, provided the order is a permutation of the one defined by the slot array.

To extract the query component the test purpose depicted in Figure 6 is used. Intuitively, the terminal requires a first slot and, while there are still slots in the card, the terminal may issue a sequence requiring a next slot. To each request, the card replies, but the details of the reply are not specified in this abstract view. The extracted component, simplified for better understanding, is shown in Figure 7. The requirements are expressed by the PVS invariants in Figure 8.

The allslots1 invariant in Figure 8 specifies that all cells in the report array correspond to different slots. The second invariant allslots2 specifies that each slot in the vSlots array, i.e. the actual slots on the card, is present in the report array. Together, they encode the requirements for the Query function.

To transform the component into an IOSTS, an initial condition saying that initially all used slots contain different currencies is added to it. The invariants are first proved on the PVS translation of the IOSTS and then, using Proposition 2, the invariants in Figure 8 hold on the whole specification as well.
FORMAL VERIFICATION AND CONFORMANCE TESTING 175

Figure 6. Test purpose: abstract description of the Query function.

Figure 7. Selected component: the Query function.
%%% at the end, all reported slots are different
allslots1: THEOREM
invariant(LAMBDA (s: State):
   s 'pc = Init AND s 'vSlotsReported = pSlotCount IMPLIES
   (FORALL (i, j: below(s 'reportSize)): i /= j IMPLIES
    s 'report(i) 'currency /= s 'report(j) 'currency))

%%% at the end, all used slots have been reported
allslots2: THEOREM
invariant(LAMBDA (s: State):
   s 'pc = Init AND s 'vSlotsReported = pSlotCount IMPLIES
   (FORALL (i: below(pSlotCount)): vSlots(i) 'InUse IMPLIES
    (EXISTS (j: below(s 'reportSize)):
     s 'report(j) 'currency = vSlots(i) 'currency)))

Figure 8. PVS invariants for properties of the Query function.

(It is not hard to check that the hypotheses of Proposition 2 are met: the initial condition has to hold on the whole specification, and this is exactly what has been proved using the Create/Update function.)

To prove the allslots1 invariant the invariant-strengthening scheme described in Section 3 was used. Four auxiliary invariants had to be proved, two of which are small syntactical modifications of the original property, while the two others are trivial inductive invariants. This verification took three days.

However, this simple approach does not work for proving the allslots2 invariant. This is because the allslots2 property is of the form (pc = CepInit AND vSlotsReported = pSlotCount) \(\supset\) Q, and Q, which states that every used slot is reported, holds only at the end, i.e. when all slots have been reported. Moreover, no small syntactical modification of it holds ‘during’ the reporting process to make a convenient auxiliary invariant. The reason for this difficulty is precisely the fact that the slots have been reported in an arbitrary order.

Thus, the allslots2 invariant had to be reformulated. A convenient way of doing this is to use finite sets, i.e. write the allslots2 aux property stating that when the reporting process is done, the used slots are a subset of the reported slots. The main steps of the proof of allslots2 aux are:

1. prove that the opposite inclusion holds during the reporting process, i.e. that the reported slots are a subset of the used slots;
2. at the end of the reporting process, the cardinalities of the two sets are equal.

These properties had to be strengthened to become inductive. Twelve auxiliary predicates were proved, which took one week for a moderately experienced PVS user, who became more experienced in the process.

Overall, the verification took about two weeks. Note that all the invariants (the main requirements as well as the auxiliary predicates required to make them inductive) were proved on components that are roughly ten times smaller than the whole specification; a direct verification of the whole specification...
would have taken considerably more time and effort. Hence, it was essential that STG allowed the selection of components with a small number of transitions.

7. CONCLUSIONS AND RELATED WORK

Safety-critical systems have to satisfy strict, formal requirements. Verification consists of proving that a formal specification of the system satisfies the requirements, while conformance testing consists of comparing a running implementation of the system with respect to its formal specification. Both approaches are required for ensuring that the final, running implementation operates correctly.

This paper proposes an approach that merges the verification and test generation efforts into one common task. Test selection is employed to extract from the specification the components that are relevant to the requirements. The requirements are verified on the components and, if some reasonable sufficient conditions hold, the whole specification also satisfies the requirements. After the verification, the extracted components form the basis of adequate test cases, in the sense that, when augmented with the verdict states (Pass and Fail) and executed on the implementation, the verdicts have the following properties.

- A Fail verdict generated by a difference between implementation and specification with respect to a verified requirement means that the implementation does violate that requirement (there are no false negative verdicts).
- A Pass verdict means that the implementation did not violate any verified requirement while the test case was executed (no false positives).

These statements hold for requirements that are invariance properties about observable features of the system (such as absence of a given output). The STG tool is used for extracting components, and the PVS theorem prover is used to verify them. The approach is demonstrated on an electronic purse system.

7.1. Limitations

The approach presented has its own limitations. First, it depends on the user’s ability to select adequate components using STG. For this, the user provides STG with adequate ‘test purposes’, which are abstract descriptions of the subgraph of the specification that he/she intends to select (see Section 2). This is a trial and error process that often converges rapidly for applications whose control graph is of the order of hundreds of transitions.

For the compositional verification method to work, the selected components also have to satisfy some sufficient conditions, which express the fact that some variables are accessed in writing only within a subset of the system’s control graph. (The conditions are syntactical and automatically checkable by STG.) This works best for systems whose control graph is the union of control graphs of well-structured procedures, whose variables are mainly local variables of those procedures. Another feature that is naturally described by this notion of component is that of (macro) transitions in the SDL language [17], which are graphs of (micro) transitions in the IOSTS sense.

Finally, the verification depends on the user’s ability to work with PVS (albeit in a guided and systematic manner, see Section 3).
7.2. Comparison with related work

7.2.1. Testing

Several conformance testing tools based on model-checking technology have been developed, and some have been transferred into commercial tools.

The TorX tool [2] combines test generation and execution, by performing, in parallel, an on-the-fly exploration of the implementation and of the specification. In TGV [3] and Autolink [19], the test generation and execution phases are distinct. TGV and Autolink have been incorporated into commercial tools (respectively, ObjectGeode and Tau from Telelogic [20]). In these tools, both verification and test generation are performed using observers (similar to the test purposes of this paper). An observer attempts to reach some interesting point in the program (e.g. a specific control point, or a specific condition between the data variables). The sequences leading to that point can then be employed for test generation. Both verification and test generation are based on model checking.

In other works from the area of structural testing, model checking is employed to generate counter examples, which are then used in the testing phase [21,22]. In [23], a theorem prover is used to certify test cases for parameterised real-time systems, which are computed by reachability analysis using a symbolic model checker.

Alternative testing approaches that handle data in a symbolic manner employ rewriting [24], abstract data types [25], or constraint solving [26]. In these works, the focus is mainly on structural testing. The symbolic state-space exploration (if it terminates) can also be used for verifying invariants. Combining verification and testing is an active area of research; see the Formal Approaches to Testing of Software (FATES’01,’02,’03) and the Workshop on Automated Program Analysis, Testing and Verification (WAPATV’00, ’01) series of events.

7.2.2. Compositional verification

Compositional verification has a long history [27]. Perhaps best known in this field are the assume-guarantee approaches. A specification is decomposed into its parallel components; each component is verified individually, assuming that the other ones satisfy their requirements; and the correctness of the whole system is inferred from the correctness of its parts. For synchronous parallel composition (e.g. the parallel composition of IOSTS) this kind of reasoning is called assumption-commitment, while for parallel composition with shared variables it is called rely-guarantee [28]. Other well-known verification methods for shared-variable concurrency include [29–31].

As suggested by the discussion at the end of Section 4, the compositional reasoning mechanisms presented in this paper are orthogonal to assumption commitment. However, they cannot be applied to shared-variable concurrency. They can be seen as assume-guarantee reasoning adapted to systems whose components are structural subgraphs, not parallel components.

7.2.3. Slicing

Slicing is a syntactical operation that consists of selecting a part of a program that has potential relevance to a given slicing criterion. The criterion is typically the values of (some, or all) program variables at a given control node, or set of control nodes [32,33]. Slicing with respect to a temporal...
logic formula has also been defined [34]. Once an adequate slicing criterion for a property has been found, computing the relevant ‘slice’ of the program is automatic, and the verification is automatic, too. This leads to significant savings (see, for example, the Bandera toolset [34] that uses slicing prior to model checking).

However, slicing is a conservative over-approximation of the relevant subset of the program that needs to be explored, and it is sometimes too conservative. For example, if a given variable $x$ occurs in a property, slicing will automatically select all the transitions (i.e. instructions) of a program that access $x$ in reading or in writing. Using the selection mechanisms described in this paper (i.e. selection using test purposes) it was possible to select components in which only a subset of these instructions were present; and this was essential for applying the results presented in Section 4, which allow for compositional verification.

REFERENCES