Interface Generation and Compositional Verification in JavaPathfinder

Dimitra Giannakopoulou and Corina Pasareanu

NASA Ames Research Center,
Moffett Field, CA 94035, USA,
{dimitra.giannakopoulou, corina.s.pasareanu}@nasa.gov

Abstract. We present a novel algorithm for interface generation of software components. Given a component, our algorithm uses learning techniques to compute a permissive interface representing legal usage of the component. Unlike our previous work, this algorithm does not require knowledge about the component's environment. Furthermore, in contrast to other related approaches, our algorithm computes permissive interfaces even in the presence of non-determinism in the component. Our algorithm is implemented in the JavaPathfinder model checking framework for UML statechart components. We have also added support for automated assume-guarantee style compositional verification in JavaPathfinder, using component interfaces. We report on the application of the presented approach to the generation of interfaces for flight software components.

1 Introduction

Component interfaces are a central concept in component-based software engineering. Although in current practice, interfaces typically describe the services that a component provides and requires at a purely syntactic level, the need has been identified for interfaces that document richer aspects of component behavior. Such extended interfaces are usually not provided, which makes their automatic generation an area of active research [1, 9, 5].

This paper addresses the automatic generation of interfaces that describe legal sequences of component calls. Such interfaces can serve as a documentation aid to application programmers, but can also be used by verification tools in checking that components are invoked correctly within a system. In fact, component interfaces are key for modular program analysis. They reduce the task of verifying a system consisting of a component and a client, to the more tractable task of verifying that the client satisfies the component's interface.

In previous work [6, 14], we presented a framework based on learning, to perform automated assume-guarantee model checking of safety properties. To check that a system consisting of components M_1 and M_2 satisfies a safety property P, the framework automatically builds and refines *assumptions* A for one of the components, for example M_1, to satisfy P, which it then tries to discharge on the other component, M_2. Although assumptions A essentially constitute interfaces for component M_1, their generation relies on knowledge of component M_2. Moreover, the focus of the framework was to compute assumptions that would allow to prove or disprove the property in the system, rather than assumptions that precisely document the behavior of a component.

The algorithm presented here for interface generation is also based on learning. However, in contrast to our work discussed above, it concentrates on the creation of precise component interfaces, *irrespective* of the component clients. By precise, we mean *safe* and *permissive*, as defined in [9]. An interface is safe if it accepts no illegal sequence of calls to the component. An interface is permissive if it includes all the legal sequences of calls to the component. Moreover, in [8], we presented an algorithm for generating what we call *weakest* assumptions in the context of Labeled Transition Systems. Weakest assumptions essentially constitute precise component interfaces. The difference of the current algorithm is that it is iterative, meaning that it can return partial results. Moreover, the approach in [8] required an expensive determinization step that we avoid here by dealing with the non-determinism in the component *dynamically*, during component analysis, as guided by counter-examples. Furthermore, our past experience, as well as other independent work [5], has indicated that the learning-based approach is more efficient for components that have relatively small interfaces.

Henzinger et al. also target the generation of safe and permissive interfaces in [9]. Unlike our framework, their work based on abstraction techniques and it is only applicable to components that are *visibly deterministic*. The latter requires that the behavior of the component be deterministic with respect to the methods / actions in its communication interface (we will henceforth call the communication interface of a component its *alphabet* in order to avoid confusion with interface in this context). In the applications that we have been dealing with, this requirement proved too restrictive. For example, we often need to generate interfaces that focus on specific aspects of the component behavior, and that therefore include only a subset of the component's alphabet. Components that are visibly deterministic with respect to their full alphabet, typically lose this property when a subset of that alphabet is considered. Finally, Alur et al. [1] also use learning to synthesize interface specifications for abstracted Java components. However, their approach is heuristic-based, i.e., they do not always obtain precise interfaces.

We have implemented our algorithms in the JavaPathfinder (JPF) model checking framework for UML statechart components [10]. We have also added support for automated assume-guarantee style compositional verification in JPF, using component interfaces. JPF is an open source model checker for Java programs which, until now, provided no support for compositional verification.

The contributions of this work can be summarized as follows:

1. A novel algorithm for automated generation of precise component interfaces, also applicable to components that are not visibly deterministic
2. Implementation of our algorithm in the JPF open source model checker. In addition to interface generation, we have provided support for verification

of safety properties expressed as finite state automata as well as assume-guarantee reasoning in JPF, where assumptions and guarantees are both expressed as finite-state automata. The implementation is freely available as JPF's compositional verification (`cv`) extension.
3. Case studies in the context of NASA applications that demonstrate the use of our algorithm in practice.

Related Work The work closest to ours was discussed above. Several other approaches to automatic generation of component interfaces have been proposed in the literature. For example, Whaley et al. [17] use a combination of static and dynamic analyses to generate interfaces for Java components. Tkachuk et. al [16] use static analysis to obtain component abstractions, used as environments during modular analysis. Some approaches are based on extracting interfaces from sample execution traces, e.g. [3]. All these techniques generate approximate interfaces, as opposed to our work that aims at producing precise interfaces that provide correctness guarantees. Interface generation is related to compositional verification. In particular, assume-guarantee reasoning is a compositional approach that uses assumptions when reasoning about components in isolation [11, 15, 2, 7]. Component interfaces can be used as assumptions in this context.

2 Background

We model software components using labeled finite state transition systems (LTSs), where transitions are labeled with component actions.

Let $\mathcal{A}ct$ be the universal set of observable actions and let τ denote a local action *unobservable* to a component's environment. Let π denote a special *error state*, which models safety violations in the associated transition system; π has no outgoing transitions.

LTSs An LTS M is a four-tuple $\langle Q, \alpha M, \delta, q_0 \rangle$ where: Q is a finite non-empty set of states; $\alpha M \subseteq \mathcal{A}ct$ is a set of observable actions called the *alphabet* of M; $\delta \subseteq Q \times (\alpha M \cup \{\tau\}) \times Q$ is a transition relation; and $q_0 \in Q$ is the initial state.

Let $M = \langle Q, \alpha M, \delta, q_0 \rangle$ and $M' = \langle Q', \alpha M', \delta', q_0' \rangle$. M *transits* into M' with action a, denoted $M \xrightarrow{a} M'$, if $(q_0, a, q_0') \in \delta$ and either $Q = Q'$, $\alpha M = \alpha M'$, and $\delta = \delta'$ for $q_0' \neq \pi$.

An LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$ is *non-deterministic* if it contains τ-transitions or if there exists $(q, a, q'), (q, a, q'') \in \delta$ such that $q' \neq q''$. Otherwise, M is *deterministic.*

Traces A *trace* t of an LTS M is a finite sequence of observable actions that label the transitions that M can perform starting at its initial state, ignoring the τ-transitions. For $\Sigma \subseteq \mathcal{A}ct$, we use $t \uparrow \Sigma$ to denote the trace obtained by removing from t all occurrences of actions $a \notin \Sigma$. For a set of traces T, $T \uparrow \Sigma = \{t | \exists t' \in T . t' \uparrow \Sigma = t\}$.

Parallel Composition Parallel composition "$\|$" is a commutative and associative operator such that: given LTSs $M_1 = \langle Q^1, \alpha M_1, \delta^1, q_0^1 \rangle$ and $M_2 = \langle Q^2, \alpha M_2, \delta^2, q_0^2 \rangle$, $M_1 \parallel M_2$ is an LTS $M = \langle Q, \alpha M, \delta, q_0 \rangle$, where $Q = Q^1 \times Q^2$, $q_0 = (q_0^1, q_0^2)$, $\alpha M = \alpha M_1 \cup \alpha M_2$, and δ is defined as follows (the symmetric version also applies): (1) $M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2$ if $M_1 \xrightarrow{a} M_1'$ and $a \notin \alpha M_2$, and (2) $M_1 \parallel M_2 \xrightarrow{a} M_1' \parallel M_2'$ if $M_1 \xrightarrow{a} M_1'$, $M_2 \xrightarrow{a} M_2'$, and $a \neq \tau$.

3 Interface Generation

In this section we define safe and permissive interfaces for software components and we describe our approach to synthesizing such interfaces automatically.

3.1 Safe and Permissive Interfaces

Let M be a software component. For simplicity of presentation, we will first assume that M includes an error state that expresses the undesired behavior of M (for example, some assertion violations). Later in this section we will discuss the more general case where the component property is given as a separate (safety) automaton.

Let $\Sigma \subseteq \alpha M$ denote the communication alphabet of component M, i.e., the set of actions through which M communicates with its environment. Our goal is to compute M's precise interface as a finite state automaton A over Σ. As mentioned, we need to make sure that A is both *safe* and *permissive*, as defined formally below.

Let us first define the legal and illegal languages of component M. A word $t \in \alpha M^*$ is *illegal* if it corresponds to *some* trace of M that leads to error state π; otherwise, the word is *legal*. Then $\mathcal{L}_{legal}(M)$ denotes the set of legal words of M and $\mathcal{L}_{illegal}(M)$ denotes the set of illegal words of M. Note that $\mathcal{L}_{legal}(M)$ and $\mathcal{L}_{illegal}(M)$ are complementary. Furthermore, note that, while illegal words correspond to actual traces in the component, legal words may also represent behavior that is never executed by the component (and hence could never lead to violations).

Definition 1. *A is a* safe *interface iff $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma = \emptyset$.*

In other words, an interface is safe if it accepts no illegal words of M.

Definition 2. *A is a* permissive *interface iff $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$.*

In other words, an interface is permissive if it accepts all legal words of M.

3.2 Learning Interface Specifications with L*

Our approach for learning interface specifications is illustrated in Figure 1. We use an off-the-shelf learning algorithm, L* [4], to iteratively compute interface specification A for M that is both *safe* and *permissive*. L* learns an unknown

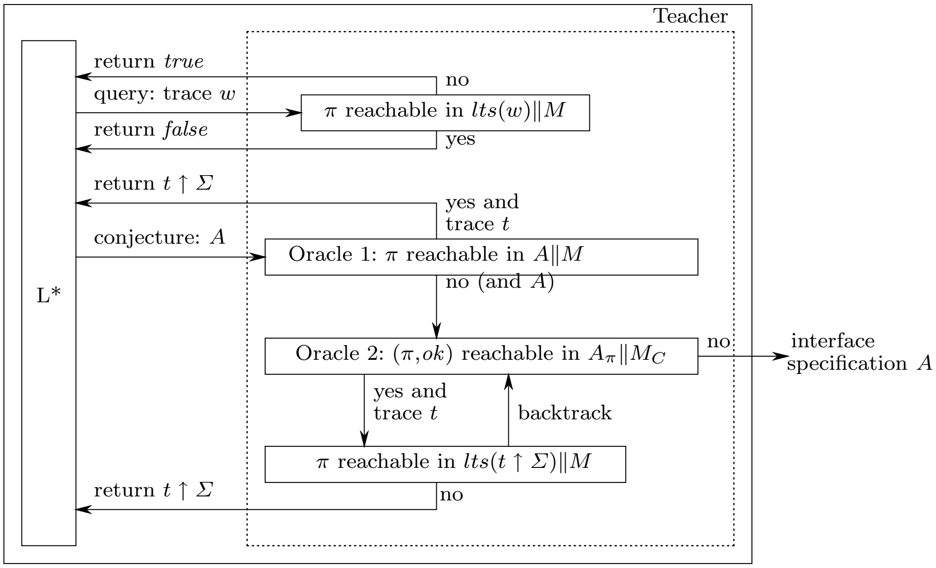

Teacher
L*
return *true*
query: trace w
return *false*
no
π reachable in $lts(w)\|M$
yes
return $t \uparrow \Sigma$
yes and
trace t
conjecture: A
Oracle 1: π reachable in $A\|M$
no (and A)
Oracle 2: (π,ok) reachable in $A_\pi\|M_C$
no
interface
specification A
yes and
trace t
backtrack
π reachable in $lts(t \uparrow \Sigma)\|M$
return $t \uparrow \Sigma$
no

Fig. 1. Learning interface specifications with L*

language (over a given alphabet) and produces a *minimal* deterministic finite state automaton that accepts it; the learning process is iterative and it uses a *teacher* that provides answers to queries and counterexamples to conjectures (for details on L* see [4]). In our framework, the problem of answering queries and counterexamples is reduced to reachability problems, solved by a model checker.

Queries L* is first used to repeatedly *query* M to check whether or not, in the context of strings w, M violates the property. This is equivalent with checking if an error state π is reachable in $lts(w)\|M$. Here $lts(w)$ denotes an LTS over Σ that accepts string w (and its prefixes). The results of the queries are used by L* to first make a "conjecture", i.e. it builds an automaton A that accepts all the strings for the positive queries (the case error unreachable), and does not accept the strings for the negative queries (the case error reachable).

The conjectured automaton A is then checked to make sure it is both safe and permissive. This is done with the help of a *teacher* that implements two oracles as described below.

Oracle 1 checks if A is *safe* by checking whether π is reachable in $A \parallel M$. If it is, then it means that A is un-safe. The resulting counterexample t, projected on the interface alphabet Σ, is returned to L* to refine its conjecture. If the error state is un-reachable, then it means A is safe and we proceed to Oracle 2.

Oracle 2 checks if safe interface A is also permissive, i.e. we want to check that $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$. This amounts to making sure that there are no

```
Oracle 2
input: safe interface $A$;
begin
  Model-check $A_\pi \| M_C$:
        if $(\pi, ok)$ is reachable by trace $t$ then
           if $\pi$ is not reachable in $lts(t \uparrow \Sigma) \| M$ then
              return $t \uparrow \Sigma$ to L*;
           else
              backtrack;
  output: safe and permissive interface $A$;
end.
```

Fig. 2. Oracle 2

words $w \in \Sigma^*$ such that $w \in \mathcal{L}_{legal}(M) \uparrow \Sigma \cap \mathcal{L}_{illegal}(A)$. This is equivalent to $w \in \mathcal{L}_{illegal}(A)$ and $\forall t \in \alpha M$ such that $w = t \uparrow \Sigma$, $t \in \mathcal{L}_{legal}(M)$.

We search for such words using a special reachability procedure performed on $A_\pi \parallel M_C$ (see pseudo-code in Figure 2). Here A_π denotes the *completion* of A with an error state, i.e. we complete each state with outgoing transitions to π, such that each state has outgoing transitions labeled with every action in Σ. Similarly, M_C denotes the *completion* of M with a special *s*ink state. We need these constructions to reason about traces in $\mathcal{L}_{illegal}(A)$ and $\mathcal{L}_{legal}(M)$, respectively. Note that $\mathcal{L}_{illegal}(A) = \mathcal{L}_{illegal}(A_\pi)$ and $\mathcal{L}_{legal}(M) = \mathcal{L}_{legal}(M_C)$. Note also that for Oracle 2, since both A_π and M_C contain error states, we need to distinguish between the two in $A_\pi \parallel M_C$ (this was not necessary for queries and Oracle 1).

Given the above constructions, checking permissiveness reduces to checking reachability of states of the form: (π, ok), were π is an error state coming from A_π and ok denotes a non-error state in M_C. If such a combined state is found, then the trace t leading to it *may* indicate that A is not permissive, since $w = t \uparrow \Sigma$ leads to an error state in A_π but it is legal in M_C (and hence in M). However, due to non-determinism in M (and hence in M_C), it may be the case that on another path, t *does* lead to the error state. Even if this is not the case, there may exist other traces t' such that $w = t' \uparrow \Sigma$ and t' leads to an error in M_C on a different path (see Figure 3).

We check both these cases by performing a *query* on $t \uparrow \Sigma$. Note that *we do not stop the state space exploration*, but rather, we take trace t that is returned, and we check if, in the context of $t \uparrow \Sigma$, M violates its properties.

If the query returns true, then it means the interface is not permissive, and therefore $t \uparrow \Sigma$ is returned to L* for refinement, and the learning process continues with more queries and eventually with a new conjecture.

If the query returns false, then t does not correspond to a real counterexample. Model checking therefore ignores this state; it backtracks, and then continues its state space exploration. If no traces that satisfy the condition above exist, then indeed the conjectured automaton is also the most permissive interface, and therefore it is output to the user.

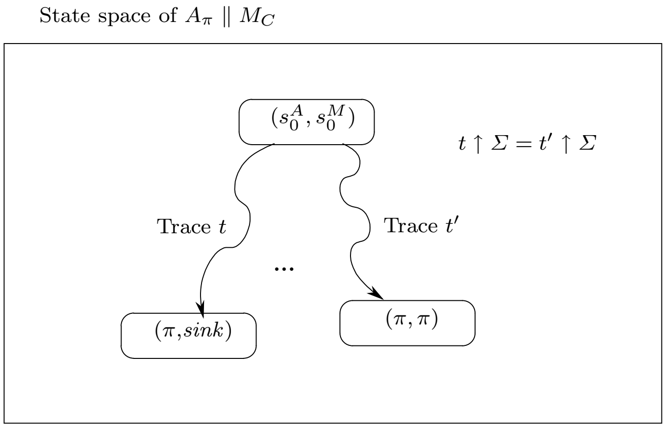

State space of $A_\pi \parallel M_C$
(s_0^A, s_0^M)
$t \uparrow \Sigma = t' \uparrow \Sigma$
Trace t
Trace t'
...
$(\pi, sink)$
(π, π)

Fig. 3. Example for Oracle 2: dealing with non-determinism

We note that every query is stored in the L* memoized table, so the result of the query on the same trace $t \uparrow \Sigma$ later (when A is the same) will be obtained directly (and faster) from the table.

Properties as safety automata Assume now that M does not have error states, and we want to generate an interface specification for ensuring a property P, given as a (deterministic) safety automaton, encoding all the desired behaviors of the component. Conversely, P_π encodes all the un-desired behaviors of the component. The procedure described above will be exactly applicable to this case as well, if we treat $M||P_\pi$ as M above.

3.3 Correctness and Termination

We argue here the correctness and termination of our approach. To argue correctness, we first show that Oracle 1 (and similarly the queries) guarantee a safe interface while Oracle 2 guarantees a permissive interface; therefore, the teacher implemented by our approach is correct.

Proposition 1. *Oracle 1 returns A iff* $\mathcal{L}_{legal}(A) \cap \mathcal{L}_{illegal}(M) \uparrow \Sigma = \emptyset$.

Proposition 2. *Oracle 2 returns A iff* $\mathcal{L}_{legal}(M) \uparrow \Sigma \subseteq \mathcal{L}_{legal}(A)$.

Due to lack of space we omit the proofs here; they proceed by contradiction and follow the arguments given informally in the previous section.

Theorem 1. *Given component finite state M (that may include error states), the algorithm implemented by our approach terminates and it returns a safe and permissive interface A.*

Proof. Correctness follows from the two propositions above. Termination follows from the correctness of L, which is guaranteed that, if it keeps receiving counterexamples, it will eventually terminate.*

Discussion As mentioned, in previous work we defined an algorithm for building safe and permissive interfaces for finite state components [8]. That algorithm involves the determinization of M (using the sub-set construction) that results in an exponential cost in computation time, regardless of the size of the interface specification. However, for components with small interfaces, the interface automaton is expected to be much smaller than the component itself. We address this problem by using L*, which builds incrementally automata with increasing size, finishing with the *minimal* deterministic automaton representing a safe and permissive interface.

We also note here that the approach of Henzinger et al. [9] can only handle components that are visibly deterministic, and therefore could not handle the case illustrated in Figure 3. On the other hand, the approach of Alur et al. [1] handles non-deterministic components, but it does not guarantee that the interface is permissive, since it only uses heuristics to implement what it amounts to Oracle 2 (called "superset query" in [1]). That work argues that the superset query can not be implemented efficiently, since it involves the determinization of component M. In our work we avoid an explicit determinization step of M. Instead, our approach deals with the non-determinism in the component dynamically (during model checking of the component) and only *selectively* (as guided by counterexamples).

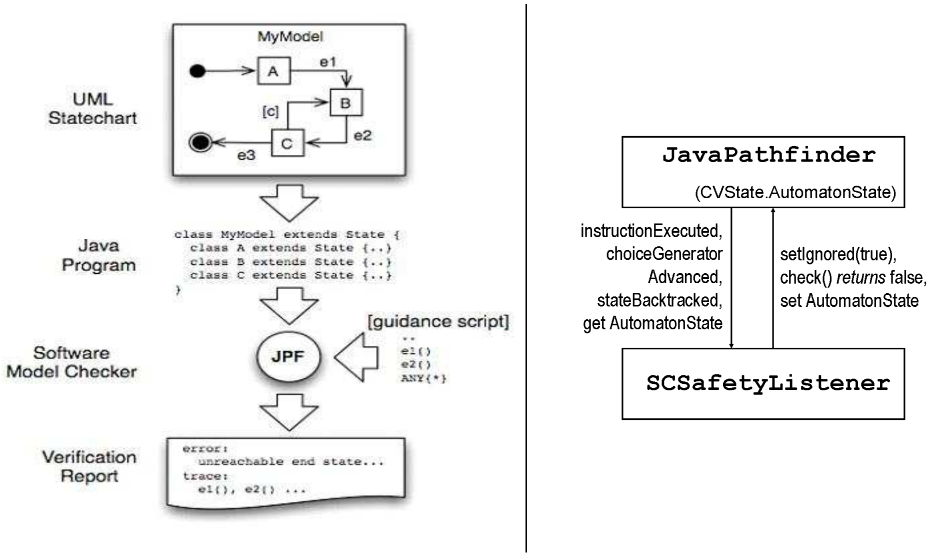
4 Compositional verification in JPF

4.1 Java PathFinder

Java PathFinder (JPF) [10] is an open-source verification framework developed by the RSE group at NASA Ames. It has been started as an explicit state model checker for Java byte-code. The focus of JPF is on finding bugs, such as concurrency related bugs (deadlocks, races, missed signals etc.), runtime related bugs (e.g. unhandled exceptions), etc. JPF can also check for violations of user-specified assertions that encode application specific requirements. JPF uses a variety of scalability enhancing mechanisms, such as user extensible state abstraction and matching, on-the-fly partial order reduction, configurable search strategies, and user definable heuristics (searches, choice generators).

4.2 JPF's UML Statechart Extension

JPF has recently been extended with a statechart modeling and analysis capability that allows Java modeling of UML state machines [13]. Many UML development systems can produce code from diagrams, but this code is usually aimed at production systems, and is not suitable for software model checkers. The approach taken in JPF (Figure 4(left)) is based on a specific translation scheme from UML state charts into Java code that (a) is highly readable, (b) shows close correspondence between diagram and program, (c) provides a 1:1 mapping between model and program states, and (d) imposes few restrictions about aspects and actions that can be modeled.



```
class MyModel extends State {
  class A extends State {..}
  class B extends State {..}
  class C extends State {..}
}
```

```
..
e1()
e2()
ANY{*}
```

```
error:
  unreachable end state...
trace:
  e1(), e2() ...
```

Fig. 4. Example illustrating JPF's UML extension (left) and JPF's listener (right)

The JPF Statechart extension is specialized to handle the obtained Java models more efficiently than random Java code. These Java models can be run in isolation, which corresponds to running them in the context of an external environment that may provide any input event at any stage (we will call this the universal environment). Alternatively, a guidance script may be provided by the user, which represents the input event sequences that can be provided by the external environment.

We have used the JPF statechart extension to implement our interface synthesis algorithms for components expressed in the JPF statechart framework. In the context of this work, we do not attempt to perform compositional reasoning for UML statecharts. The reason is that statechart composition semantics is obfuscated and setting up compositional reasoning for statecharts is a challenge even at a purely theoretical level. Rather, we use UML statecharts, as supported by JPF, to represent finite state components with Labeled Transition System semantics. Therefore composition of components comes down to LTS composition, as described in Section 2. The interfaces that we generate are expressed as LTSs in the FSP notation [12].

4.3 Assume-guarantee Reasoning in JPF

We have implemented assume-guarantee reasoning in JPF. As mentioned, components are given as UML statecharts (instances of class `CVState`). Both properties and assumptions are represented as finite state automata (instances of class `gov.nasa.jpf.cv.SCSafetAutomaton`).

Model checking using assumptions and properties has been implemented using JPF listeners (Figure 4(right)). A listener is essentially configured client code that is notified when certain events occur while JPF performs its search. The notified listener code can interact with JPF, e.g. a JPF "property" listener informs JPF if the property holds via the return value of its `check()` method.

Checking for both assumptions and properties is implemented with the `gov.nasa.jpf.cv.SCSafetyListener` class. On creation, a `SCSafetyListener` is associated with a finite state automaton P, which expresses the property or assumption to be used during model checking. Note that the state of a listener is not included in the state that JPF explores / stores during model checking. However, the state of the automaton P needs to be part of the state space for correct state-space exploration and backtracking. We perform this by adding a static integer field of class `CVState` for the `cv` extension, which is set from within the listener.

An `SCSafetyListener` listens for and reacts to the following events:

- `instructionExecuted`: Signals to the listener that an instruction was executed by JPF. The listener reacts by invoking method `advance(...)` on the automaton P. Advancing the automaton corresponds to making a state transition, if the instruction that was executed corresponds to an action in the alphabet of the automaton. If a transition on an alphabet action is undefined from the current state, this is an illegal transition (corresponds to a transition to the *error* state π). For properties, this means that an error has occurred, so the result returned by the listener's `check()` method is false.
- `choiceGeneratorAdvanced`: Signals that the next statechart action is selected for execution. The reaction of the listener is to check whether this action would make P transition to the error state if it were to be executed (this does not change the state of P since the transition is not really executed yet). Reaching an error state in an assumptions means that the current path explored is not a valid path under this assumption and must therefore be ignored. The listener forces JPF to backtrack (by executing `vm.getSystemState().setIgnored(true)`).
- `stateBacktracked`: When the model checker backtracks, then the automaton must backtrack accordingly.

For example, in order to check some property described as an automaton provided in some file `Foo`, we need to include the following arguments when running JPF's main class `gov.nasa.jpf.JPF`:

```
+jpf.listener=.cv.SCSafetyListener
+safetyListener1.property= Foo
```

The first argument informs JPF that an `SCSafetyListener` will need to be notified of specific events, and the second one provides details for the listener, i.e., its unique id is "1", it is of type property (as opposed to assumption), and the automaton associated with it is provided in file `Foo` (this may also include the full path to `Foo`).

```
public boolean query(Vector sequence) throws SETException {

  Boolean recalled = memoized_.getResult(sequence);
  if (recalled != null) {
      return (!recalled.booleanValue());
  } else {
      // play the query as an assumption
      System.out.println("\n New query: " + sequence);
      SCSafetyListener assumption = new SCSafetyListener(
          new SCSafetyAutomaton
                  (true, sequence, alphabet_, "Query", module1_));

      JPF jpf = createJPFInstance(assumption, property, module1_);
      jpf.run();
      boolean violating = jpf.foundErrors();
      memoized_.setResult(sequence, violating);
      return (!violating);
  }
}
```

Fig. 5. Answering queries in `SCModularTeacher`

4.4 Interface Generation and Discharge

The interface generation in JPF is implemented in the main class `gov.nasa.jpf.tools.cv.ScRunCV`. The user can customize the generation via the following arguments:

`+assumption.alphabet=<actions>` defines the interface alphabet;

`+assumption.outputFile=<file name>` defines a file in which the generated interface is output.

This allows for a generated interface to be used for subsequent reasoning, either as an assumption, or as a property. The format currently used for expressing the interface is the FSP language.

The main method of `gov.nasa.jpf.tools.cv.ScRunCV` creates an instance of class `gov.nasa.jpf.tools.cv.SETLearner` to carry out the learning of the interface; an associated instance of `gov.nasa.jpf.tools.cv.SCModularTeacher` serves as the teacher. Our learning algorithm implementation uses JPF to perform the model checking steps described in Section 3. JPF model checks individual components in the context of the universal environment. Listeners are added as necessary to reflect the work of the Teacher, which consists of answering Queries, and implementing Oracle 1 and Oracle 2 in order to answer conjectures, as described in more detail below.

Queries and Oracle 1. Queries and Oracle 1 are performed in a similar fashion because they are concerned with checking whether error states are reachable in the component, in the context of a particular sequence (for queries) or finite state automaton (for Oracle1). As illustrated in Figure 5, to respond to a query, a listener instance `assumption` is created with an associated automaton that reflects the particular sequence that is being queried. JPF is then invoked, together with the `assumption` listener. If JPF returns errors, the answer to the query is `false`, otherwise the answer is `true`. Oracle 1 works in a similar fashion, with the difference that it also returns a counterexample.

Oracle 2. Oracle 2 checks for permissiveness of a computed interface. It needs to work on the completed component, as described in Section 3. This is a manual step that we intend to automate in the future. It similarly invokes JPF, but performs the search in the context of a specialized type of listener, the `gov.nasa.jpf.cv.SCConformanceListener`. Its aim is to detect the reachability of a (π, ok) combination of states in the interface and component where the interface is in an error state, while the component is in an non-error state.

The `gov.nasa.jpf.cv.SCConformanceListener` listens for and reacts to the following events:

- `executeInstruction`: When the instruction about to be executed by JPF is an assertion violation, then it means that the component has entered an error state. Since such states are not targeted by the listener, it performs
  `ti.skipInstruction();`
  `vm.getSystemState().setIgnored(true);`.
  The first command ensures that the exception is not processed by JPF, for efficiency. The second asks JPF to backtrack since this path cannot lead to the targeted combination of states.
- `instructionExecuted`: Similar to `gov.nasa.jpf.cv.SCSafetyListener`. When the automaton associated with the listener moves to an error state, the result returned by the `check()` method of the listener is set to false, since the component is in a legal state (illegal states are never reached since the listener advises JPF to backtrack when it reacts to `executeInstruction` events), while the interface is in an error state.
- `stateBacktracked`: Similar to `gov.nasa.jpf.cv.SCSafetyListener`.

As described in Section 3, when an (π, ok) state is detected by the model checker, the counterexample leading to this state is queried, and if it is not a real counterexample, the model checker will backtrack. Since a query involved calling the model checker, this would involve nested model checker calls. To avoid such nesting, our implementation exploits a memoized table that is used by the learner to store results of previous queries. Oracle 2 checks for the reachability of (π, ok) states in a *loop*. Whenever a counterexample is obtained by the model checker, then OraclE2 invokes a query on it. Each query stores its result in the memoized table.

Whenever a real counterexample is obtained, Oracle 2 exits the loop and reports the result to the learner. When a counterexample is spurious, then another iteration of the loop is entered. In this iteration, we wish to ensure that the model checker will not report the same spurious counterexample. We achieve this as follows. When a `gov.nasa.jpf.cv.SCSafetyAutomaton` is asked to advance in the context of a `gov.nasa.jpf.cv.SCConformanceListener`, if the automaton reaches an error state, it will get the path to this state from JPF. It will then

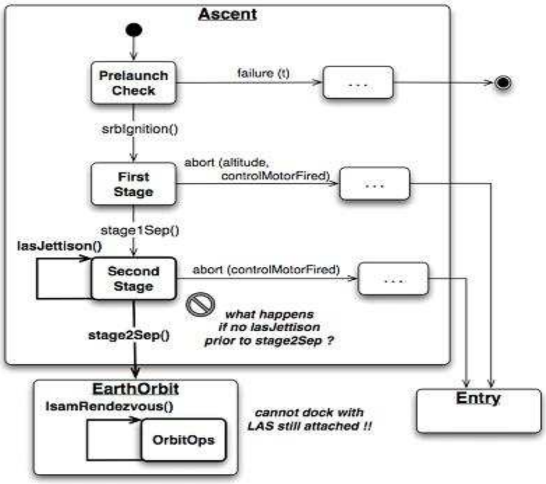


Fig. 6. Model of the Ascent and Earth Orbit flight phases of a spacecraft

check the memoized table to see if there is a result for the corresponding sequence stored there. If there is, and the result is true, then it means that this is a spurious counterexample, and it notifies JPF to backtrack. Therefore, we have implemented the nested model checking calls by consecutive calls to the model checker, where the information of spurious counterexamples is shared through the memoized table.

Interface discharge. For compositional reasoning, one needs to also discharge the generated interface on the component environment. This can be performed by model checking the environment component in the presence of a `gov.nasa.jpf.cv.SCSafetyListener` using the interface as a property.

5 Experience

In order to evaluate our implementation, we used a statechart model of the *Ascent* and *EarthOrbit* flight phases of a space-craft (see Figure 6). The JAVA model is available with the JPF distribution under `examples/jpfESAS`. The UML statechart diagrams for the model are included in `examples/jpfESAS.doc`.

The model was created and used to demonstrate the features of the JPF UML statechart extension to our NASA mission customers. Several properties were analyzed on the model, and JPF returned violations for some of these properties. When the counterexamples obtained were analyzed, it was clear that some of the violations were spurious. The violations were related to the following properties:

- An event *lsamRendezvous*, which represents a docking maneuver with another spacecraft, fails if the LAS (launch abort system) is still attached to the spacecraft.

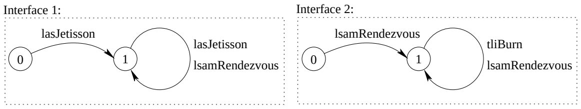


Fig. 7. Generated interface specifications

- Event *tliBurn* (trans-lunar interface burn takes spacecraft out of the earth orbit and gets it into transition to the moon) can only be invoked if EDS (Earth Departure Stage) rocket is available.

These violations were due to the fact that the universal environment was too general. The models had been created under the assumption that the use of the model respects some implicit flight rules. We decided to use our interface generation techniques to formalize the flight rules. More specifically, for each property, we generated a safe and permissive interface to eliminate its corresponding violations. To do this, we added a listener that eliminated all assertion violations that were not related to the targeted property, through the following arguments:

```
+jpf.listener=.tools.ChoiceTracker:.cv.AssertionFilteringListener
+assertionFilter.include=<method_name>
```

These arguments specify that all assertion violations that occur outside the particular `<method_name>` will be ignored.

The generated interface specifications are illustrated in Figure 7. The first one expresses the fact that the *lsamRendezvous* maneuvers cannot start before the *las* module of the spacecraft has jettisoned. According to the second one, it does not make sense to perform *tliBurn* prior to performing *lsamRendezvous*. These interfaces were inspected by the developer of the model that confirmed that they encode actual flight rules. Interface generation can therefore be used by developers to help them in the expression of the assumptions that their models encode. We note that other examples, including the input-output example from [6], are available with the JPF distribution.

6 Conclusions

We have proposed an algorithm for automatically synthesizing behavioral interface specifications for finite state software components. Our algorithm is the first iterative approach that is guaranteed to compute interfaces that are both safe and permissive, even in the presence of non-determinism in the visible behavior of a component. We have implemented our approach in the JavaPathfinder model checking framework for UML statechart components, and have obtained promising results from its application to several systems. The source code for the implementation and the examples is available through JPF's distribution.

In the future, we plan to investigate interface generation for methods with parameters. We have made some initial experiments using JPF's symbolic execution extension to generate values for parameters with infinite domains, and used these values to define finite interface alphabets related to their corresponding methods. We wish to pursue this direction further, and also plan to extend our results to generic Java components. For components that may be infinite-state, we will combine our approach with techniques such as predicate abstraction (similar to [1]). Finally, we plan to perform extensive evaluations of our approach.

References

1. R. Alur, P. Cerny, P. Madhusudan, and W. Nam. "Synthesis of interface specifications for Java classes". In *Proceedings of POPL'05*, pages 98–109, 2005.
2. R. Alur, T. Henzinger, F. Mang, S. Qadeer, S. Rajamani, and S. Tasiran. "MOCHA: Modularity in Model Checking". In *Proceedings of CAV'98*, volume 1427 of *LNCS*, pages 521–525, 1998.
3. Glenn Ammons, Rastislav Bodk, and James R. Larus. Mining specifications. In *Proceedings of ACM POPL'02*, pages 4–16, 2002.
4. D. Angluin. "Learning regular sets from queries and counterexamples". *Information and Computation*, 75(2):87–106, November 1987.
5. D. Beyer, T. A. Henzinger, and V. Singh. "Algorithms for Interface Synthesis". In *Proceedings of CAV'07*, volume 4590 of *LNCS*, pages 4–19, 2007.
6. J. M. Cobleigh, D. Giannakopoulou, and C. S. Pasareanu. "Learning Assumptions for Compositional Verification". In *Proceedings of TACAS'03*, volume 2619 of *LNCS*, pages 331–346, 2003.
7. C. Flanagan, S. N. Freund, and S. Qadeer. "Thread-Modular Verification for Shared-Memory Programs". In *Proceedings of ESOP'02*, pages 262–277, 2002.
8. D. Giannakopoulou, C. S. Pasareanu, and H. Barringer. "Assumption Generation for Software Component Verification". In *Proceedings of ASE'02*, pages 3–12. IEEE Computer Society, 2002.
9. T. A. Henzinger, R. Jhala, and R. Majumdar. "Permissive Interfaces". In *Proceedings of ESEC/SIGSOFT FSE'05*, pages 31–40, 2005.
10. Java PathFinder. `http://javapathfinder.sourceforge.net`.
11. C. B. Jones. "Specification and Design of (Parallel) Programs". In *Information Processing 83: Proceedings of the IFIP 9th World Congress*, pages 321–332. IFIP: North Holland, 1983.
12. Jeff Magee and Jeff Kramer. *Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.
13. Peter Mehlitz. "Trust Your Model - Verifying Aerospace System Models with Java Pathfinder". In *IEEE/Aero*, 2008.
14. C. S. Pasareanu, D. Giannakopoulou, M. Gheorghiu Bobaru, J. M. Cobleigh, and H. Barringer. "Learning to Divide-and-Conquer: Applying the L* Algorithm to Automate Assume-Guarantee Reasoning". *FMSD*, January 2008.
15. A. Pnueli. "In Transition from Global to Modular Temporal Reasoning about Programs". In *Logic and Models of Concurrent Systems*, volume 13, pages 123–144, 1984.
16. Oksana Tkachuk and Matthew B. Dwyer. " Adapting side effects analysis for modular program model checking". In *Proceedings of ESEC/SIGSOFT FSE 2003*, pages 188–197, 2003.
17. John Whaley, Michael C. Martin, and Monica S. Lam. " Automatic extraction of object-oriented component interfaces". In *Proceedings of ISSTA 2002*, pages 218–228, 2002.