Trace Based Reachability Verification for Statecharts

Kumar Madhukar, Ravindra Metta, Ulka Shrotri, R. Venkatesh
TRDDC, Tata Consultancy Services,
Plot 54 B, Hadapsar Industrial Estate, Pune 411013. India
{kumar.madhukar|r.metta|ulka.s|r.venky}@tcs.com

***Abstract*—Statecharts are widely used to model the behavior of reactive systems. While this visual formalism makes modeling easier, the state of the art in verification of statechart specifications is far from satisfactory due to the state explosion problem. We present History Abstraction, a trace-based verification technique to address this problem. Given a set of traces in a statechart model, the model is abstracted to contain at most three states per statechart: current, history and future. A path to a desired state in the abstract model is a sketch of a potential path to that state in the original model. We follow an incremental concretization procedure to extend the sketch to a complete path in the original model. This paper presents our technique. Our experiments suggest that the technique scales to large industry models.**

I. Introduction

The use of formal specification notations such as Statemate statecharts [1] has been on the rise for the specification of embedded reactive systems. Such expressive visual formalisms make modeling easier and facilitate early verification and validation of requirements through formal analysis. Model checkers are widely used for the verification of standard properties such as state reachability and data racing of the formal models. Industry models typically have a large number of concurrent components causing the state-space to grow exponentially. Due to this state explosion problem, model checkers often do not scale to real world models [2]. In this paper, we present what we believe to be the first trace-based verification technique aimed at containing the state explosion problem for Statemate statecharts.

Given a Statemate statechart model M and a set of traces Tr, our basic motive is to try to extend the traces to reach the states that are not reached in any of them. Note that, one can easily obtain a seed Tr by running a bounded model checker with a small depth for state reachability or by using the traces produced by production test data, etc. From the set of traces, we select a trace that has an unreached state as an immediate successor of the last reached state in the trace. We call such a trace as a ***candidate trace***.

Given a candidate trace, we construct M_A, an abstract model of M that contains at most three states per statechart. For each statechart S in M, these three states are:

- C, the *current* state - the last reached state of S in the candidate trace. C is also the initial state of M_A.
- H, the *history* state - corresponding to the set, say S_r, of all states of S reached in Tr except C.
- F, the *future* state - corresponding to the set, say F_S, of immediate successor states of C unreached in Tr.

We set the initial system state of M_A to the last system state of the candidate trace. Since F represents an unreached set of states, we would like to verify the reachability of each F in M_A. For this, we add all those transitions of M to M_A that directly impact the reachability of a state represented by F (except in some cases, explained in Section 3.1). We do not add some transitions to the abstract model as they can never be taken starting from C without reaching a state represented by F. Further, for each impacting transition added to the abstract model:

- we retain only those parts of the transition's actions that impact F's reachability and discard the rest
- if the transition's source and target states are already reached, but neither is C, we discard the guards

Reactive systems tend to have long dependence chains due to their very nature. The intuition behind the above construction of M_A is to limit this dependency.

Next, we run a model checker with a bound k on M_A to obtain a trace to F. If no trace is found, it means that the candidate trace does not extend from C to any state in F_S in M, in the next k steps. If a trace does exist, the transitions that constitute the trace represent a *'set' of transitions* where each transition represents a potential path segment of a path in M from C to a state in F_S. Intuitively, the trace constitutes a set of 'likely' transitions in M that help reach a state in F_S. We then try to find a concrete path-segment to each transition in the set using an incremental concretization procedure, exploring these transitions in a depth-first manner. The composition of these path segments, prefixed with the candidate trace, forms a concrete path in M.

This technique is expected to scale because the size of M_A is likely to be significantly smaller than the size of M and, at each point of time during the concretization, we incrementally construct individual concrete path segments as opposed to complete concrete paths. However, the following factors are expected to hinder its scalability. Firstly, the concretization may require many iterations . Secondly, the performance of the technique is directly impacted by the quality of the seed set of traces. In particular, if a candidate trace could not be extended to a desired future state at a given depth, the trace may still be extensible to the state at a higher depth. And, another candidate trace may also be extensible to the desired state. Thus, we need to try all candidate traces of a given set of traces at the largest feasible depth before we give up.

To check the practicability of the technique, we implemented it in an in-house statechart verification tool [3] and

978-1-4673-6292-4/13/$31.00 © 2013 IEEE

experimented on a large automotive model. In our experiments, the technique scaled verification to show reachability of hitherto unreached states in a large industry model, suggesting that our technique is useful in practice.

The rest of the paper is organized as follows. We give the relevant work in Section II and background information on Statemate statecharts and SAL in Section III. Sections IV-A and IV-B present the abstraction and concretization techniques respectively. Section V presents our experimental results. The paper concludes with some remarks and future work in Section VI.

II. Related Work

Zhang et. al [4] propose a 'dynamic abstraction' that applies the information gathered from an unsatisfiability proof to create an abstraction during successive steps of their model checking algorithm. In contrast we use a given set of unsatisfiability proofs (traces) to construct an abstract model, which is then used to extend the traces to desired future states. In an earlier work, Bischoff et al. [5] proposed another 'dynamic abstraction' technique that builds an under-approximate set of reachable states using a breadth-first traversal of a corresponding BDD, which is then used to perform a SAT-based verification. In contrast, we use a known set of reachable states and their corresponding traces, which we then use to perform a SAT-based verification. In contrast to both these 'dynamic' abstractions, our trace-based abstraction is 'dynamic' with respect to the set of traces.

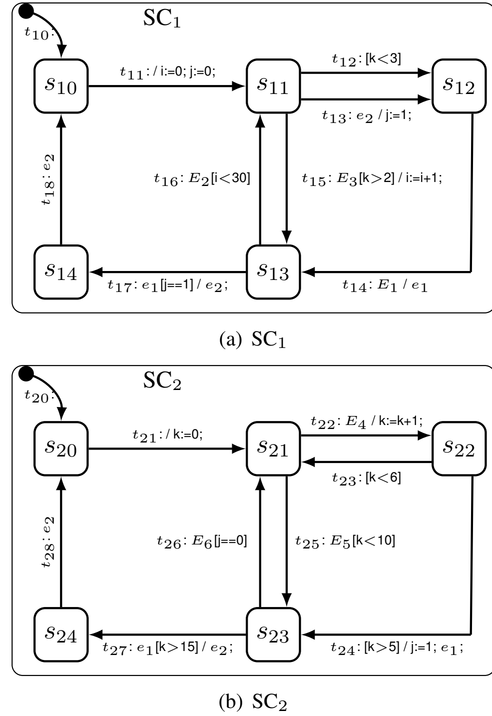
Concolic test generation for Simulink/Stateflow models [6] employs directed traversal from a point already reached and uses feedback from a current run to obtain the future part of the run. In comparison, our technique first constructs a smaller abstract model from the set of already reached points from a given set of runs and then extends the runs on the abstracted model.

Finally, the words 'history' and 'future' have been used earlier by Abadi et. al. in [7] in the context of mapping from a concrete model to a abstract model. In this work, new auxiliary variables called history and prophecy variables are added in an additional component to a state machine. History variables record past behavior and prophecy variables guess future behavior. In comparison, we use already recorded history (traces) to build a smaller abstract model, which we then use to guess a path to a future state and concretize the guess to a complete path on the original models.

III. Background

A. Statemate

Statecharts are extensions of conventional state transition systems. The main extensions are hierarchy and concurrency. A transition is an edge between two states, labeled with *e[c]/a*, where *e* is an event (also called a *trigger*), *c* is a condition (also called a *guard*) and *a* is an action. A transition *e[c]/a* is ***enabled*** if its source state is active, event *e* occurs, condition *c* holds true and no higher priority transition is enabled. In Statemate, transitions exiting higher level states in the hierarchy assume higher priority. Whenever a transition is taken, the corresponding action *a* is executed.



(a) SC_1

(b) SC_2

Fig. 1. A Sample Statechart Model

Definition 1. *A* **transition** *is a tuple* $\langle src, tgt, ev, cond, act\rangle$ *where src is the source state of the transition, tgt is its target state, ev is an event, cond is a condition and act is an action. The elements* ev, $cond$ *and* act *are optional.*

The elements of the tuple $\langle src, tgt, ev, cond, act\rangle$ of a transition t are respectively referred to as src_t, tgt_t, ev_t, $cond_t$ and act_t.

A Statemate model consists of a set of statecharts, reacting in parallel. Fig 1 shows an example model consisting of charts SC_1 and SC_2 executing concurrently. A **step** denotes a reaction of the model where each statechart executes an enabled transition. The execution proceeds in a sequence of steps. An event generated during the ith step gets sensed at the beginning of the $i+1^{th}$ step and expires immediately after the $i+1^{th}$ step. The contents of a step are determined by the *system status* at the beginning of that step. The *system status* is a detailed snapshot of the system and is referred to as a *configuration*.

Definition 2. *A* **configuration** *is a tuple* $\langle S, V, E\rangle$ *where S is a set of currently active states of each statechart, V is a set of current values of data items and conditions and E is a set of events to be sensed in the next step.*

A Statemate model is said to be in a *stable configuration* if it can not react any further without an environmental stimulus. A Statemate model senses the environmental stimuli only when it is in a stable configuration. Based on this, events in

Statemate are categorized into *internal* and *external* events. Internal events are those generated by the model and external events are those generated by its environment. When in a stable configuration, a Statemate model senses environmental stimuli and begins a chain reaction (a sequence of steps) reacting only to the internal stimuli until it reaches a stable configuration again. Any environmental stimuli occurred during the chain reaction are stored and are sensed only upon reaching the next stable configuration. This sequence of steps from one stable configuration to another is called a **superstep**. Such an execution of steps may alternately be represented as the sequence of transitions executed by the steps. Such a transition sequence is called a *trace*.

Definition 3. *A* **trace** *is a sequence of transitions (equivalently, a sequence of configurations) denoted by* $\langle t_1, t_2, ..., t_n \rangle$.

The above descriptions constitute the essential Statemate terminology that we need for the rest of this paper. For the complete set of Statemate statechart constructs and their semantics, the reader may refer to [8].

B. SAL

Symbolic Analysis Laboratory (SAL) is a framework developed at SRI [9] for combining different tools for program analysis, abstraction, theorem proving, and model checking of transition systems. SAL has a language for describing transition systems. Following is a brief list of important features of the SAL language. For more details, the reader may refer to [9].

The basic building block of any SAL program is a `MODULE`. A `MODULE` has a set of variables, divided into input, output, global and local variables. The input variables are read-only variables. The rest are read-write variables. Initialization is carried out exactly once and that is when the system execution starts. All transitions of a `MODULE` are defined in its `TRANSITION` section.

SAL modules can be composed asynchronously using the operator `[]`. Properties expressed as LTL formulas can be specified in SAL as theorems, which can be verified using SAL's model checkers: sal-smc (a symbolic model checker), sal-bmc (a bounded model checker) and sal-inf-bmc (an infinite state bounded model checker). We used sal-bmc for our experimentation.

IV. History Abstraction

A. Abstraction

Our abstraction is somewhat motivated by path extensions where the idea is to allow an intermediate configuration (one which is known to be reachable along some path) of the system to become an initial configuration. We aim at reducing the state-space drastically by allowing initialization in an intermediate configuration. Given a (candidate) trace, we are interested in the reachability of those states that lie in its immediate future. A state s is in the immediate future of a trace tr if there is a transition t from s' to s, where s' is one of the states the system is in when tr completes execution. The abstraction maintains a maximum of three states per statechart. It further eliminates or abstracts most of the transitions in a way that the reachability to an immediate future state is not impacted, for a given selection of the candidate trace.

The algorithm begins to build the abstract model using the final configuration of a candidate trace. The abstract model is made to initialize from a configuration which is consistent with this. The state in which a statechart is in, as per this configuration, is called the *current* state for that statechart. The unreached immediate future states of every current state are preserved as part of the abstraction, in the form a single state - called a *future* state. All transitions between a current state and its successors are included in the abstract model, with their target states changed to the *future* state. All other states reached through some trace, from a given set of traces, are merged into a single state - called a *history* state. Every transition exiting the current state to reach a state in history, and vice-versa, is added to the abstracted model with its source or target state changed appropriately. A transition between two states in history, if needed, is made to appear as a loop from the *history* state to itself. The triggers and guards of these transitions are omitted. Such a looping transition is needed only if its action modifies the trigger or guard of an existing transition in the abstract model. Additionally, to allow the system to break out of such loops, we add a dummy external (environment) event as their trigger.

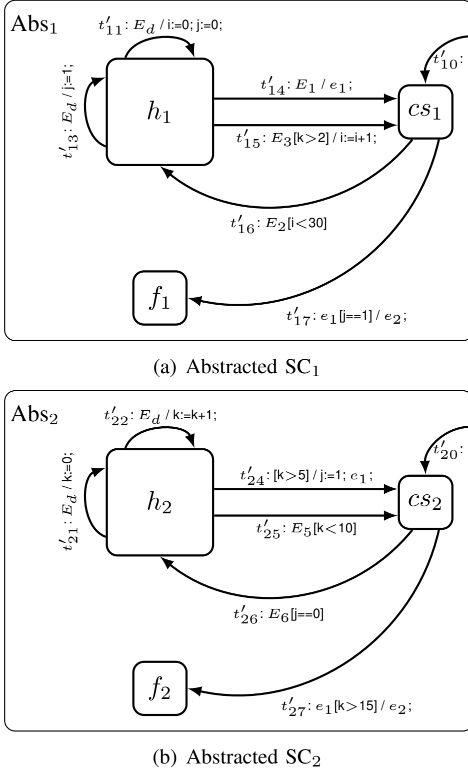
Formally, let M denote the model. Let Tr be a given set of traces, tr be our candidate trace in Tr and C_f denote the final configuration of tr. Let cs_i denote the state of statechart SC_i as per C_f. Let S_r be the set of all states reached, through one or more traces in Tr, excluding those in C_f. Let F_i denote the set of immediate unreached successors of cs_i. Let C denote the union of cs_i, over all i. Let S_i and T_i, respectively, denote the set of states and transitions in SC_i. Let E_d be a dummy external event. We construct an abstract model M_A from M, as follows:

For each statechart SC_i in M, M_A keeps an abstracted chart Abs_i. The states of this chart are cs_i, a future state f_i to represent all immediate unreached successors of cs_i, and a history state h_i. The transitions in Abs_i are populated by executing the following steps for each transition $t \in T_i$:

1) if tgt_t is cs_i and $src_t \in S_r$, add $t[src \mapsto h_i]$[1] in Abs_i
2) if src_t is cs_i
   a) if $tgt_t \in F_i$, add $t[tgt \mapsto f_i]$ in Abs_i
   b) if $tgt_t \in S_r$, add $t[tgt \mapsto h_i]$ in Abs_i
   c) if tgt_t is cs_i, add t in Abs_i
3) if $src_t \in S_r$ and $tgt_t \in S_r$ and act_t modifies a variable v such that v is used to evaluate the guard of some transition t_a in Abs_i, add $t[src \mapsto h_i, tgt \mapsto h_i, ev \mapsto E_d, cond \mapsto true]$ to Abs_i

After the first iteration, the abstraction procedure keeps iterating over the set T_i to execute step 3 until no more transitions are added to the abstract model.

[1] $x[y \mapsto z]$ denotes the change applied to x where the value of the component y in x is replaced by z.



(a) Abstracted SC_1

(b) Abstracted SC_2

Fig. 2. Abstracted Statechart Model

Fig. 2 shows the abstract model obtained from the statecharts in Fig. 1 for the trace $\langle t_{10}, t_{11}, t_{12}, t_{20}, t_{21}, t_{22}, t_{23}, t_{25}, t_{14} \rangle$. The configuration C_f at the end of this trace is $\langle cs_1 = s_{13}, cs_2 = s_{23}, i = 0, j = 0, k = 1, e_1 = true, e_2 = false \rangle$. The set of reached states, S_r, is $\{s_{10}, s_{11}, s_{12}, s_{20}, s_{21}, s_{22}\}$. The transition sets $\{t'_{14}, t'_{15}, t'_{24}, t'_{25}\}$, $\{t'_{17}, t'_{27}\}$, $\{t'_{16}, t'_{26}\}$ and $\{t'_{11}, t'_{13}, t'_{21}, t'_{22}\}$ are added in the steps 1, 2(a), 2(b) and 3, respectively.

Claim. *If the original model M, starting from the configuration C_f, has a path p to a state s in F_i, say, such that the intermediate states of p belong to $S_r \cup C$, then there exists a path p' in the abstracted model, M_A, which reaches the future state representing s.*

Proof. The following steps demonstrate a construction of p' by modifying p.

- replace the guard of each transition in p by E_d, unless the transition appears with the guard in M_A
- remove the action from each transition, unless either it was already a part of the abstract model, or it affects any of the remaining guards in p
- remove a transition if its guard has been replaced and its action has been removed

It is easy to see that p' is valid trace to the *future* state representing s in the abstracted model. The abstraction, therefore, does not limit the behavior of a model for a given set of history, future and current states. On the other hand, it allows for infeasible runs. We eliminate such infeasible runs by concretizing the paths incrementally.

B. Path Concretization

The reachability of a future state, in the abstracted model, is checked by translating the model to a SAL specification. The negation of the reachability property is encoded as an LTL formula and we check for its validity using a bounded model checker. The counterexample generated, in the event of the formula not being valid, gives us a sequence of transitions leading to some future state. The details of this procedure are out of scope of this work and may be found in [3].

The counterexample trace generated by SAL shows reachability of the desired state, in the abstracted model. This trace is essentially a sequence of transitions leading to a *future* state in M_A. This trace may not be feasible in the actual model. And even if it is, it may not be a complete trace with respect to the actual model. In other words, since we have removed all transitions having no direct effect on the reachability to any future state, the obtained trace may not be contiguous. The gaps in the trace need to be filled by the transitions that we may have omitted, during the abstraction step. Moreover, the order of transitions imposed by the trace may not conform to the actual model. The enabled transitions of two statecharts executing in parallel may fire in any order. The removal of guards and triggers from the self-looping transitions in the *history* state is another reason why we need to look at different possible reorderings of the sequence.

Let us assume, for simplicity, that the transitions in the trace are distinct. Since we are not interested in the order in which these appear in the trace, we treat it as a set of transitions. Starting from the initial configuration, we want to find out if there's a way to take the transitions in this set to reach the desired state. This process of concretization of a path essentially consists of the following four steps:

1) Starting from the current configuration, check if it is possible to enable any transition in the set (i.e., reach the source of a transition along some path such that its guard evaluates to true).
2) If so, select one of the transitions which can be enabled and pick a path p enabling it; extend the path obtained so far by appending p to it, update the current configuration, remove the selected transition from the set and continue.
3) If not, undo the last step, i.e., chop off the path segment appended last; store this information so as to avoid such a path in the following iterations.
4) Discard the set if you can't undo any further.

Formally, let $\langle t_1, t_2, .., t_n \rangle$ be the sequence of transitions returned by SAL. Let S denote the set of these transitions. Let C_k denote the configuration after the transition t_k, for $1 \leq k \leq n$. Let C_0 be the initial configuration. Let t_i be the first transition from one of the history states to itself. The transitions before t_i have not been abstracted and hence would have fired in order. Similarly, let t_j be the last transition from a history state to itself, and hence all transitions following t_j constitute a concrete path. Therefore, we start the

concretization procedure with the configuration C_{i-1}, the set of transitions being [t_i, t_{i+1},.., t_j]. Let M denote the original statechart model. The pseudocode in Algorithm 1 illustrates the *concretize* procedure.

Algorithm 1 Pseudocode of the concretize procedure

```
INPUT:

S_inp ← [t_1, t_2, .., t_n]
M

INITIALIZATION:

Compute i, j, C_0, C_1, .., C_{i-1} from the inputs
P_init ← ⟨t_1, t_2, .., t_{i-1}⟩; P_end ← ⟨t_{j+1}, t_{j+2}, .., t_n⟩
P ← P_init
S ← [t_i, t_{i+1}, .., t_j]
C ← C_{i-1}
D_in ← ∅    /* stores discarded intermediate paths */

C_term ← parent[C_0]    /* C_term is added for termination */
for each k ∈ [1..(i − 1)]
  C_{k-1} ← parent[C_k]
end for each;

PROCEDURE:

while |S| > 0

  if (C == C_term)
    discard S_inp; return
  end if;

  ⟨result, index, path⟩ ← findNext(C, S, D_in)

  if (result == true)
    P ← append(P, path)    /* extends P by appending path */
    S' ← S \ t_index
    C' ← updateConf(path)
    parent[S'] ← S; parent[C'] ← C
    S ← S'; C ← C'
  else
    S_prev ← parent[S]; C_prev ← parent[C]
    D_in ← D_in ∪ ⟨C_prev, S_prev, C⟩
    S ← S_prev; C ← C_prev
  end if;

end while;

P ← append(P, P_end)
if(validate(P, M)) then return P else return end if;
```

An intermediate path is disallowed by storing tuples of the form $\langle C, S, C'\rangle$, which essentially means that while starting from the configuration C, and the set of transitions to be taken being S, any path that leads to the configuration C' is uninteresting. This being so, because the configuration C' is known to be a non-progress configuration for the set S. The *findNext* procedure works with the SAL encoding of the model where the property to be satisfied has been expressed in LTL. The entire process of translation to SAL to get traces satisfying some desired property has been explained in one of our earlier works ([3]). Given a starting configuration C, a set of transitions S and a set of tuples, D_{in}, of the form $\langle C, S, C'\rangle$, *findNext* checks if there is a way to take some transition in S such that if C_f is the configuration at the end of that transition, then $\langle C, S, C_f\rangle \notin D_{in}$. While such a procedure seems powerful enough to answer the reachability of any state directly, in practice, bounded model checking fails to explore paths longer than a small finite length for large models. By interpreting the transitions constituting the trace in the abstract model as intermediate edges in the final path, our task reduces to connecting these edges, one at a time. The likelihood of these connecting paths being much smaller allows better scaling.

The procedure *updateConf* updates the current configuration to the system configuration at the end of the *path* and *validate* checks the feasibility of the path by running it on the actual model. The assumption that the transitions be distinct was only needed for the ease of argument. It is equally easy to maintain another data structure which allows us to store the multiplicity of each transition, unlike sets.

The set of transitions discarded by the *concretize* procedure, when it revisits the initial configuration, is used to *constrain* the abstraction so as to prevent SAL from returning the same set again, for a given future state. The procedure *constrain* encodes this as an additional constraint in SAL for every discarded set. Algorithm 2 outlines the entire process of reachability verification using *abstract-concretize-constrain*. The *abstract* procedure performs the history abstraction in a given model, as explained in the preceding section.

Algorithm 2 Outline of the entire procedure

```
M_A ← abstract(M, Tr, tr)
p_a ← (M_A, f)    /* p_a is a path to f in M_A */
while(p_a)
  p_o ← concretize(p_a, M) /* p_o is a path to f in M */
  if(p_o) then return tr.p_o    /* f is reachable if concretize returns a path */
  else
    M_A ← constrain(M_A, p_a, f); p_a ← (M_A, f)
  end if;
end while;
```

Correctness. The correctness of the algorithm follows since we validate each path by running it on the actual model. Such a validation is required since we disregard the priority of transitions while constructing the path incrementally. While checking if a transition can be enabled, we only check if it is possible to reach the source state of the transition such that its guard evaluates to true. We do not add the additional constraint, in order to simplify the search, that if there's a

TABLE I
EXPERIMENTAL RESULTS

State	Abstraction	Concretization (semi-manual)	#(Iterations)
1	< 20 sec	< 1 min	1
2	< 20 sec	< 1 min	1
3	< 20 sec	< 15 min	4
4	< 20 sec	< 10 min	-

higher priority transition exiting that state then its guard must evaluate to false.

Termination. In each iteration, *concretize* either reduces the size of the set of transitions by one, or marks a configuration pair, for a given set, as undesirable. Since the set of transitions has only finitely many elements and the undesirable configurations are never revisited, the procedure terminates in finite time under the assumption that the number of distinct configurations is bounded. Moreover, since discarding a set of transitions allows us to discard a set of configurations, the procedure *concretize* also terminates if the number of distinct configurations is bounded.

To illustrate this with our example in Fig. 2, let us assume that the initial trace returned by SAL is $\langle t'_{16}, t'_{13}, t'_{14}, t'_{17}\rangle$. Since the transitions $\{t'_{16}\}$ and $\{t'_{14}, t'_{17}\}$ at the beginning and the end of the trace, respectively, are concrete transitions, *concretize* starts with the set S as $\{t'_{13}\}$. The configuration, C, to begin with, is the updated configuration after executing t'_{16}, starting from $\langle cs_1 = s_{13}, cs_2 = s_{23}, i = 0, j = 0, k = 1, e_1 = true, e_2 = false\rangle$ (C_f, in the history abstraction section). The new configuration is very easily computed to be $\langle cs_1 = s_{11}, cs_2 = s_{23}, i = 0, j = 0, k = 1, e_1 = false, e_2 = false\rangle$. The call to *findNext*$(C, S, \emptyset)$ fails to return a path since such a path is infeasible. The SAL encoding is therefore modified with the additional constraint that disallows the set $\{t'_{16}, t'_{13}, t'_{14}, t'_{17}\}$ in future. The path $\langle t'_{26}, t'_{22}, t'_{22}, t'_{22}, t'_{22}, t'_{22}, t'_{22}, t'_{17}\rangle$, returned in a subsequent iteration of our procedure, is concretized to a valid path.

V. EXPERIMENTATION

We have implemented our abstraction technique as a part of our Statemate analysis tool [3], [2]. For our experiments, we used a Statemate model of an ECU that controls the back door functionality of a car. The model has 108 concurrent statecharts with a total of 407 basic states. The techniques in [3], [2] showed reachability of 367 basic states by producing traces (counterexamples) but failed to scale to the remaining 40 states. For our experimentation, we randomly chose 4 states out of these 40 and used the traces produced for the 367 states to construct the abstract models. Table I presents the time taken by the abstraction and concretization procedures, and the number of iterations needed for concretization for each of the 4 states.

Our technique found paths to all the four states in the abstract model. We were able to concretize the path to three of these four states with minimal manual efforts (we are yet to fully automate the concretization procedure). For the first two states, the paths that we obtained turned out to be an extension of the corresponding candidate traces. This could not be obtained by [3], [2] as they could not scale to these states due to the number of states in the model. For states 3 and 4, the paths produced were spurious. By constraining the model (refer Algorithm 2), we were able to find a valid path to state 3 in four iterations. For state 4, it turned out that our algorithm could never return a feasible path. This is because reachability to this state was dependent on a state that is unreached so far. Note that this exposes dependencies across states, where one should attempt to reach a state only when its dependencies are satisfied.

Our experiments suggest three key benefits of our technique: a) valid extensions of candidate traces are inherent to our technique, b) a spurious trace returned by our technique gets discarded quickly by the *concretize* procedure and c) the (reachability) dependency across states is highlighted.

VI. CONCLUDING REMARKS AND FUTURE WORK

We presented History Abstraction, a trace based abstraction technique that attempts to address state explosion. To the best of our knowledge, this is the first such technique for the verification of statecharts. As the abstraction reduces the state-space and the concretization deals with small path segments, this improves the scalability to large industry models. Moreover, as the technique does not depend on any Statemate specific semantics of statecharts, it is applicable to all state transition systems.

On large models, bounded model checkers do not scale because of long paths needed to reach some states. Our experiments show that this is the case in practice and that our technique scales to such states. However, our concretization may take many iterations if there are many spurious paths in the abstract model. For instance, this would be the case if there exist other future states that have to be reached before reaching a desired future state. This necessitates a precise dependency analysis for reactive systems which is hard because of the recursive dependency imposed by reactive loops. We plan to investigate this in future, to strengthen our technique.

REFERENCES

[1] D. Harel and A. Naamad, "The statemate semantics of statecharts," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 4, pp. 293–333, Oct. 1996. [Online]. Available: http://doi.acm.org/10.1145/235321.235322

[2] U. Shrotri, R. Venkatesh, and R. Metta, "Proving unreachability using bounded model checking," in *Proceedings of the 3rd India software engineering conference*, ser. ISEC '10. New York, NY, USA: ACM, 2010, pp. 73–82. [Online]. Available: http://doi.acm.org/10.1145/1730874.1730891

[3] A. Kulkarni, R. Metta, U. Shrotri, and R. Venkatesh, "Scaling up model-checking," in *Next Generation Design and Verification Methodologies for Distributed Embedded Control Systems*, S. Ramesh and P. Sampath, Eds. Springer Netherlands, 2007, pp. 275–283. [Online]. Available: http://dx.doi.org/10.1007/978-1-4020-6254-4_21

[4] L. Zhang, M. R. Prasad, M. S. Hsiao, and T. Sidle, "Dynamic abstraction using sat-based bmc," in *Proceedings of the 42nd annual Design Automation Conference*, ser. DAC '05. New York, NY, USA: ACM, 2005, pp. 754–757. [Online]. Available: http://doi.acm.org/10.1145/1065579.1065776

[5] G. P. Bischoff, K. S. Brace, G. Cabodi, S. Nocco, and S. Quer, "Exploiting target enlargement and dynamic abstraction within mixed bdd and sat invariant checking," *Electron. Notes Theor. Comput. Sci.*, vol. 119, no. 2, pp. 33–49, Mar. 2005. [Online]. Available: http://dx.doi.org/10.1016/j.entcs.2004.06.061

[6] M. Satpathy, A. Yeolekar, and S. Ramesh, "Randomized directed testing (redirect) for simulink/stateflow models," in *Proceedings of the 8th ACM international conference on Embedded software*, ser. EMSOFT '08. New York, NY, USA: ACM, 2008, pp. 217–226. [Online]. Available: http://doi.acm.org/10.1145/1450058.1450088

[7] M. Abadi and L. Lamport, "The existence of refinement mappings," *Theor. Comput. Sci.*, vol. 82, no. 2, pp. 253–284, May 1991. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(91)90224-P

[8] D. Harel and M. Politi, *Modeling Reactive Systems with Statecharts: The Statemate Approach*, 1st ed. New York, NY, USA: McGraw-Hill, Inc., 1998.

[9] L. de Moura, S. Owre, H. Rueß, J. M. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," in *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, ser. Lecture Notes in Computer Science, vol. 3114. Springer, 2004, pp. 496–500.