# Theory-Aided Model Checking of Concurrent Transition Systems Supplementary Material


Guy Katz
*Weizmann Institute of Science*
*guy.katz@weizmann.ac.il*

Clark Barrett
*New York University*
*barrett@cs.nyu.edu*

David Harel
*Weizmann Institute of Science*
*david.harel@weizmann.ac.il*


## A. Derivation Rules for the $\mathcal{TS}$ Solver

A *derivation tree* consists of nodes containing sets of assertions. The root node contains an initial set of assertions and each non-leaf node is labeled by a derivation rule used to derive the children of the node from the node itself. The derivation rules used by the $\mathcal{TS}$ solver give rise to a sequence of derivation trees (called a *derivation*). The initial tree in the derivation contains only a single node with the initial set of assertions. Each subsequent tree in the sequence is obtained from its predecessor by the application of a derivation rule to one of the predecessor's leaves. A branch terminating with a leaf consisting of the value $\bot$ is called a *closed* branch; if all branches are closed, we say that the derivation tree is closed. A derivation culminating with a closed derivation tree indicates that the initial set of assertions is unsatisfiable. A derivation that leads to a derivation tree containing a leaf node that is not $\bot$ and to which no derivation rule can be applied indicates that the initial set of assertions is satisfiable. When such a tree is produced, the derivation terminates.

We now describe the actual derivation rules used by the theory. The first rule, used to initiate the traversal of the state space, is the *Start* rule:

$$\textsc{Start}\ \frac{\Gamma[\neg \mathit{safe}(s)]}{\Gamma, \neg \mathit{safe_state}(s,q_1)\ \ \ldots\ \ \Gamma, \neg \mathit{safe_state}(s,q_n)}\ ,$$

where $\Gamma$ denotes the initial set of assertions (in particular it includes $\mathfrak{P}$ and $\Phi$). The derivation rule contains a nondeterministic split such that each branch adds a single assertion to $\Gamma$.[1] The *Start* rule is only applicable when the following guard conditions hold:

$$\Gamma\ \vDash_{TS}\ \forall q \in Q.\ (I(s,q) \iff (q=q_1) \vee (q=q_2) \vee \ldots \vee (q=q_n)), \text{where } s \text{ is of sort } S_Q$$

$$\neg \mathit{safe_state}(s,q_i) \notin \Gamma \text{ for } i \in \{1 \ldots n\}.$$

Intuitively, the *Start* rule translates the fact that the system is unsafe into an assertion that one of the initial states is unsafe. The first part of the guard condition ensures that all of the potential initial states are considered as possibilities, and the second part ensures that the rule is applied only once at the beginning of the derivation.

Next comes the *Decide* rule. This rule performs a similar function to that of *Start*, and applies when the traversal of the state space is already underway:

$$\textsc{Decide}\ \frac{\Gamma[\neg \mathit{safe_state}(s,q)]}{\Gamma, \neg \mathit{safe_state}(s,q_1)\ \ \ldots\ \ \Gamma, \neg \mathit{safe_state}(s,q_n)}\ .$$

The *Decide* rule is applicable when the following conditions hold:

$$\neg \mathit{deadlock}(s,q) \notin \Gamma \qquad (1)$$

$$\Gamma\ \vDash_{TS}\ \forall q' \in Q, e \in E.\ ((\mathit{Tr}(s,q,e,q') \wedge R(s,q,e) \wedge \neg B(s,q,e)) \iff (q'=q_1) \vee (q'=q_2) \vee \ldots \vee (q'=q_n)) \qquad (2)$$

where $s$ is of sort $S_Q$ and $n \geq 1$.

Intuitively, if a state $q$ has not already been recorded as a non-deadlock state (condition 1), the *Decide* rule lets the solver derive the unsafety of one of $q$'s successor states, as long as such successors exist (condition 2). Note in particular that the *Decide* rule will not apply if $q$ is a deadlock state because condition 2 will fail.

For cases where *Decide* is not applicable and no deadlock state has been found, we have the *Unsat* rule:

$$\textsc{Unsat}\ \frac{\Gamma[\neg \mathit{safe_state}(s,q)]}{\bot}$$

The *Unsat* rule is applicable when there are no more states to explore on this branch (and at least one state has been explored) and no deadlock has been discovered. Specifically, the guard condition is:

$$\text{whenever } \neg \mathit{safe_state}(s,q') \in \Gamma, \text{ we also have } \neg \mathit{deadlock}(s,q') \in \Gamma.$$

In addition to the derivation rules, we use the *theory lemma* feature of the DPLL($T$) framework to periodically generate a *deadlock lemma*:

$$\mathfrak{P} \wedge \Phi \implies \neg \mathit{deadlock}(s,q)$$

[1] Splits are implemented by utilizing the SAT solver through the DPLL($T$) splitting-on-demand framework.

Of course, as a theory lemma must be valid in the theory, this lemma can only be generated if $q$ is not a deadlock state in $s$, or, formally,

$$\begin{aligned}&\mathfrak{P}, \Phi \ \models_{TS} \ \exists q' \in Q, e \in E.\\ &Tr(s, q, e, q') \wedge R(s, q, e) \wedge \neg B(s, q, e),\\ &\text{where } s \text{ has the sort } S_Q \text{ and } q \text{ has the sort } Q.\end{aligned}$$

Because $\Gamma$ includes $\mathfrak{P}$ and $\Phi$, the DPLL($T$) framework will ensure that whenever a deadlock lemma is added, the predicate $\neg deadlock(s, q)$ will be included in $\Gamma$ on all future branches. For our purposes, we can view the generation of a deadlock lemma as a derivation rule which modifies a given derivation tree by adding $\neg deadlock(s, q)$ to *every* node of the tree (except those labeled with $\bot$).

Clearly, not every strategy for applying derivation rules and deadlock lemmas will be complete. Thus, we enforce the following strategy. A deadlock lemma for $s$ and $q$ is generated immediately after an invocation of *Decide* triggered on $\neg safe_state(s, q)$; and moreover, this is the only time a deadlock lemma is generated.

This strategy guarantees that deadlock lemmas are valid and are generated only for states whose successors have been expanded, and that each state's successors are only expanded at most once in any derivation.

*Lemma 1:* For each $q$, the *Decide* rule is applied at most once with trigger $\neg safe_state(s, q)$ in any derivation starting with *Start* triggered on $\neg safe(s)$.

*Proof:* Observe a derivation tree $T$ in which the *Decide* rule has just been applied with trigger $\neg safe_state(s, q)$. The strategy used by the $\mathcal{TS}$ solver guarantees that in this case, the invocation of the *Decide* rule for $q$ will be immediately followed by the generation of a deadlock lemma for $q$ (recall that if *Decide* was applied to $q$, it cannot be a deadlock state). This transforms $T$ into a new tree, $T'$, in which the assertion $\neg deadlock(s, q)$ has been added to every node.

The $\mathcal{TS}$ solver now continues working on tree $T'$. In order for the *Decide* rule to be re-applied to $q$ in some node of $T'$, the assertions in that node must not contain $\neg deadlock(s, q)$. However, this is clearly not possible, as new nodes in $T'$ are derived from existing nodes by the addition of terms (rules *Start*, *Decide* and the lemma generation rule). Thus the only nodes in the tree where $\neg deadlock(s, q)$ is not present are nodes derived using the *Unsat* rule, in which case the branch is closed and no further rules may be applied. ■

We also observe that in every derivation that starts with $\mathfrak{P}, \Phi, \neg safe(s)$, the *Start* rule is applied precisely once.

*Lemma 2:* In every derivation starting with $\mathfrak{P}, \Phi, \neg safe(s)$, the *Start* rule is the first rule applied and it is applied precisely once.

*Proof:* This holds because (i) *Start* is the only rule applicable at the very first step of the derivation process; and (ii) as rules only add to the set of assertions along any branch, every leaf contains all of the assertions in its parent nodes; and (iii) because of (ii), once *Start* has been applied once, its guard rule prevents it from being applied again to any successor derivation tree. ■

The described derivation rules and proof strategy also guarantee partial correctness (i.e., soundness and completeness):

*Proposition 1:* Let $s$ be an input system for which the $\mathcal{TS}$ solver terminates when started with $\mathfrak{P}, \Phi, \neg safe(s)$. Then $s$ is safe iff the final derivation tree is closed.

*Proof:* Suppose that $s$ is unsafe; then let $q_0 \to q_1 \to \ldots \to q_n$ be a shortest path in $s$ such that $q_0$ is an initial state, $q_{i+1}$ is a successor of $q_i$ for all $0 \le i < n$, and $q_n$ is a deadlock state.

We now claim that for each derivation tree after the first, there is a value of $i$ with $0 \le i \le n$ such that the tree has a leaf node containing $\neg safe_state(s, q_i)$ and not containing $\neg safe_state(s, q_k)$ for $i < k \le n$ and also not containing $\neg deadlock(s, q_k)$ for $i \le k \le n$. We prove this by induction on the derivation. It is easy to see this is true for the second derivation tree as it is generated using the *Start* rule and so does not yet contain any assertions of the form $\neg deadlock(s, q)$. Thus, the largest $i$ such that $\neg safe_state(s, q_i)$ appears in the tree satisfies the claim.

Now, assume the claim holds for some tree $T$ in the derivation (whose position is second or later) with value $i$. Let $T'$ be its successor in the derivation. We consider each of the possible ways in which $T'$ could have been obtained from $T$:

- $T'$ cannot be derived using *Start* as (by Lemma 2) this rule can only be applied once in the sequence.
- Suppose $T'$ is derived from $T$ using the *Unsat* rule. It cannot be the case that the *Unsat* rule was applied to the leaf containing $\neg safe_state(s, q_i)$ as that would require $\neg deadlock(s, q_i)$ to also be present in the leaf node and our inductive hypothesis states it is not. Thus the leaf satisfying the claim in $T$ is still present and satisfies the claim in $T'$.
- Suppose $T'$ is derived using *Decide* followed by an application of the deadlock lemma (we lump these two rules together for simplicity and wlog). If the *Decide* rule uses $\neg safe_state(s, q_i)$ as its trigger, then $T'$ contains a new leaf which differs from the previous leaf by the addition of $\neg safe_state(s, q_{i+1})$ and $\neg deadlock(s, q_i)$. By minimality of the path, there are no new leaves with $\neg safe_state(s, q_k)$ with $k > i + 1$ and $q_{i+1} \ne q_i$. Thus, it is clear that the value $i + 1$ satisfies the claim in $T'$. If the *Decide* rule uses $\neg safe_state(s, q)$ as its trigger with $q \ne q_i$, there are two possibilities. The first possibility is that $q$ contains a successor $q_k$ with $k > i$. In this case it is easy to see that the value $k$ satisfies the claim in $T'$. If $q$ contains no such successor, then it is clear that the value $i$ continues to satisfy the claim in $T'$.

This shows the claim is true for every tree in the derivation, in particular the final tree which implies that the final tree is not closed.

For the other direction, suppose towards contradiction that $s$ is safe but that the final derivation tree is not closed. Then, there is a leaf node in the tree which is not $\bot$ and to which rules *Start*, *Decide*, and *Unsat* may not be applied. We denote this node by $\alpha$.

Node $\alpha$ has at least one assertion of the form $\neg \mathit{safe_state}(s, q)$ (generated by the *Start* rule at the beginning of the derivation). Any other terms of this form were generated by the *Decide* rule. From this, and by the guard condition for *Decide*, it follows that for every state $q$ such that $\neg \mathit{safe_state}(s, q)$ is a term in $\alpha$, $q$ is reachable in $s$.

We now distinguish between two cases. If there exists some $q$ such that $\neg \mathit{safe_state}(s, q)$ is in $\alpha$ but $\neg \mathit{deadlock}(s, q)$ is not, then the *Decide* rule may be applied — because we know, by our assumption that $s$ is safe, that $q$ is not a deadlock state. If there is no such $q$, then *Unsat* may be applied. Either case contradicts our assumption that no rule may be applied in $\alpha$, as needed. ■

Proposition 1 tells us that when the $\mathcal{TS}$ solver terminates, it gives a correct result. However, it may not always terminate. The following lemma characterizes one case in which termination is guaranteed:

*Proposition 2:* For an input system $s$ with a finite set of states, the $\mathcal{TS}$ solver terminates.

*Proof:* Lemma 2 ensures that in every derivation, the *Start* rule is applied only once. Further, as Lemma 1 shows, the *Decide* rule may only be applied once per state in the entire derivation. Similarly, according to the proof strategy used by the solver, the lemma generation rule may only be applied once per state, as it is only invoked after an invocation of *Decide*. Finally, the last derivation rule, *Unsat*, always reduces the number of open branches in the tree, and thus may only be applied a finite number of times once no other rules are available. From all of the above, we deduce that every derivation is of finite length. ■

### B. Backward Reachability

The derivation rules given so far — *Start*, *Decide* and *Unsat* — effectively perform a reachability analysis over the state space, looking for bad states. In some cases, as in the case of periodic programs (Section V in the paper), it may be useful to also perform a backward reachability search, starting at bad states and checking if they are reachable. This is performed by extending the $\mathcal{TS}$ theory with the predicate $\mathit{reachable}_Q : S_Q \times Q$, where $\mathit{reachable}(s, q)$ signifies that state $q$ is reachable in $s$. The semantics are extended to include (for each $Q \in \bar{Q}^+$):

$$\forall s : S_Q, q : Q.\ \mathit{reachable}(s, q) \implies I(s, q) \vee \exists q' : S_Q, e : E.\ (\mathit{Tr}(s, q', e, q) \wedge R(s, q', e) \wedge \neg B(s, q', e) \wedge \mathit{reachable}(s, q')).$$

Intuitively, a state is reachable if it is either initial or if it has a predecessor state that is reachable. This last condition is what makes the search "backward": we will start at bad states, and attempt to construct a legal path backwards, towards an initial state.

We will also use an alternative (but equivalent) semantics for the system safety predicate. For each $Q \in \bar{Q}^+$:

$$\forall s : S_Q.\ \neg \mathit{safe}(s) \iff \exists q : Q.\ (\mathit{deadlock}(s, q) \wedge \mathit{reachable}(s, q)) \ ,$$

and so a system is unsafe if it has an unsafe initial state or (equivalently) a reachable deadlock state.

The derivation rules for this case are extended to include "backward" versions of the three original rules. Here, the negation of the initial state predicate plays the role that was previously played by the $\mathit{deadlock}$ predicate, namely that of marking those states that have been visited by generating a lemma. The first derivation rule is the *BStart* rule:

$$\textsc{BStart}\ \frac{\Gamma[\neg \mathit{safe}(s)]}{\Gamma, \mathit{reachable}(s, q_1)\ \ \ldots\ \ \Gamma, \mathit{reachable}(s, q_n)}$$

The *BStart* rule is only applicable when the following guard conditions hold:

$$\Gamma\ \vDash_{TS}\ \forall q \in Q.\ (\mathit{deadlock}(s, q) \iff (q = q_1) \vee \ldots \vee (q = q_n)), \text{where } s \text{ is of sort } S_Q$$
$$\mathit{reachable}(s, q_i) \notin \Gamma \text{ for } i \in \{1 \ldots n\}.$$

Intuitively, states $q_1, \ldots, q_n$ are the deadlock (bad) states of the system — the states on which we want to perform the backward reachability analysis. Next comes the *BDecide* rule:

$$\textsc{BDecide}\ \frac{\Gamma[\mathit{reachable}(s, q)]}{\Gamma, \mathit{reachable}(s, q_1)\ \ \ldots\ \ \Gamma, \mathit{reachable}(s, q_n)}\ .$$

The *BDecide* rule is applicable when the following conditions hold:

$$\Gamma \vDash_{TS} \neg I(s, q)$$
$$\neg I(s, q) \notin \Gamma$$
$$\Gamma\ \vDash_{TS}\ \forall q' \in Q, e \in E.\ ((\mathit{Tr}(s, q', e, q) \wedge R(s, q', e) \wedge \neg B(s, q', e)) \iff (q' = q_1) \vee \ldots \vee (q' = q_n))$$
$$\text{where } s \text{ is of sort } S_Q \text{ and } n \geq 1.$$

We also have a special *BDecide2* rule which handles the special case when a non-initial state has no predecessors:

$$\textsc{BDecide2}\ \frac{\Gamma[\mathit{reachable}(s, q)]}{\Gamma}\ .$$

The *BDecide2* rule is applicable when the following conditions hold:

$$\Gamma \models_{TS} \neg I(s,q)$$
$$\neg I(s,q) \notin \Gamma$$
$$\Gamma \ \models_{TS} \ \forall q' \in Q, e \in E.$$
$$\neg(\mathit{Tr}(s,q',e,q) \wedge R(s,q',e) \wedge \neg B(s,q',e))$$
$$\text{where } s \text{ is of sort } S_Q.$$

Finally, we have the *BUnsat* rule:

$$\text{BUNSAT} \ \frac{\Gamma[\mathit{reachable}(s,q)]}{\bot}$$

The *BUnsat* rule has the following guard condition:

whenever $\mathit{reachable}(s,q') \in \Gamma$, we also have $\neg I(s,q') \in \Gamma$.

In addition to the derivation rules, we again use the *theory lemma* feature of the DPLL($T$) framework to periodically generate an *initial state lemma*:

$$\mathfrak{P} \wedge \Phi \implies \neg I(s,q)$$

As before, only valid theory lemmas are allowed, so this lemma can only be generated if we know that $q$ is not an initial state.

Similarly to the forward reachability case, the $\mathcal{TS}$ solver's strategy dictates that the initial state lemma generation rule always be applied with trigger $(s,q)$ immediately after an invocation of *BDecide* or *BDecide2* with trigger $\mathit{reachable}(s,q)$. As before, this guarantees that one of these rules is applied at most once for each state $q$.

The backward reachability derivation rules are very similar to the forward reachability ones; it is straightforward to show that they do not change the solver's soundness and completeness, and that the solver still terminates for transition systems with finite state sets and finite event sets.

## C. A Simple Schedulable Periodic Program

Consider a periodic program with 3 tasks, $T_1, T_2, T_3$, each with execution time $C_1 = C_2 = C_3 = 1$. The tasks' periods are 2, 3 and 6, respectively. As depicted in Fig. 1, every 2 consecutive time slots include a scheduling of $T_1$; every 3 consecutive slots include a scheduling of $T_2$; and every 6 consecutive slots include a scheduling of $T_3$. The schedule repeats after every 6 steps (the hyper-period). Hence, this program is schedulable.

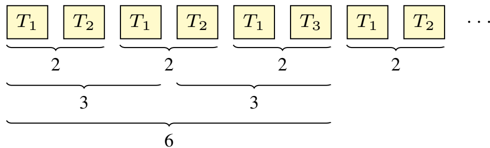


Figure 1: A scheduling for a periodic program with 3 tasks, $T_1, T_2, T_3$, with execution time $C_1 = C_2 = C_3 = 1$ and periods $P_1 = 2$, $P_2 = 3$ and $P_3 = 6$.

## D. Example: Applying the Shared Array Pattern to Tic-Tac-Toe

We demonstrate the shared array pattern on a behavioral application for playing Tic-Tac-Toe from [2]. Tic-Tac-Toe is a game played between "X" and "O" players on a 3x3 board. Each in their turn, the players mark an empty square on the board with their respective sign. A player wins by completing a row, a column or a diagonal. If neither player errs, the game is guaranteed to end in a draw.

In [2], the authors construct a behavioral application that plays "O", where a human player plays "X". Suppose that we wish to prove that the game application upholds the property that "X never wins by taking the upper row"; verifying this property can be useful, e.g., during incremental development [1].

The theory-aided verification of this system is as follows. In the implantation of [2], the game board is in fact a ternary array — with values $\mathit{empty}$, $O$ and $X$. By analyzing the individual threads, the $\mathcal{TS}$ solver recognizes this array and determines that a violation can occur only when the three upper row cells (say, cells 0, 1 and 2 in the array) are set to $X$. The $\mathcal{TS}$ solver then creates a 9-cell ternary array variable $\mathit{arr}$, and asserts that $\mathit{target} = \mathit{write}(\mathit{write}(\mathit{write}(\mathit{arr},0,X),1,X),2,X)$. Next, the solver writes fresh constants $c_0,\ldots,c_8$ to $\mathit{arr}$'s cells, and equates the result to $\mathit{target}$:

$$\mathfrak{P} \ \wedge \ \Phi \implies$$
$$\mathit{target} = \mathit{write}(\ldots \mathit{write}(\mathit{write}(\mathit{arr},0,c_0),1,c_2)\ldots,8,c_8)$$

The Tic-Tac-Toe application has, for every board square, a thread that waits for write events and then blocks any additional writes. Thus, every triggered write event fixes an array entry, causing the $\mathcal{TS}$ solver to generate a lemma that equates the corresponding $c_i$ constant to its final value. For instance, suppose the game so far has included moves $X(0,1), O(1,1), X(2,2)$, and that it is now O's turn to play. We denote this current state by $q_1$. Now, suppose the $\mathcal{TS}$ solver explores a new state, $q_2$, reachable from $q_1$ when $O$ marks square $O(0,2)$. Thread analysis shows that starting in state $q_2$ the blocker thread will forever block any additional write events to square $(0,2)$, and so the $\mathcal{TS}$ solver generates the lemma: $\mathfrak{P} \wedge \Phi \wedge \neg \mathit{safe_state}(s,q_2) \implies (c_2 = O)$. This causes the array theory solver to raise a conflict, which in turn causes the underlying SAT solver to deduce that sate $q_2$ cannot be unsafe. Thus, backtracking is performed, and another successor of state $q_1$ is checked (effectively choosing another move, instead of $O(0,2)$). Consequently, the successor states of $q_2$ need not to be explored.

**A Note.** $\mathcal{RWB}$ programs, and in particular those with shared arrays, tend to have a large number of threads. The resulting performance overhead may be mitigated via lightweight threading; see, e.g., the Erlang implementation of BP at http://www.b-prog.org.

# References

[1] D. Harel, R. Lampert, A. Marron, and G. Weiss. Model-Checking Behavioral Programs. *EMSOFT*, 2011.

[2] D. Harel, A. Marron, and G. Weiss. Behavioral Programming. *C. ACM*, 2012.