Abstract

Mutation of model-checker specifications is used to generate test-cases automatically. In general, model-checker based techniques are efficient when using small, abstract models, but are limited in their applicability by the performance of the model-checker. Issues that contribute to potentially bad performance are equivalent mutants which result in no test-cases, and the fact that many different mutants lead to identical or subsumed test-cases. Another drawback of such approaches is that the size of resulting test-suites can be rather large, which might require test-suite reduction as an additional post-processing step. In this paper, we describe how the number of mutants can be minimized with special properties, which help to detect equivalent mutants more quickly and to avoid the creation of redundant test-cases. Consequently, the size of created test-suites is reduced.

1. Introduction

Specification mutation has been suggested as a method to automatically generate test-cases with model-checkers [3]. A model-checker can be used to automatically derive traces that illustrate the difference between original and mutated specification. Such traces can be used as test-cases. In general, model-checker based test-case generation is very flexible and fully automated, but the applicability is limited by the performance of the model-checker. Although a model-checker is very efficient for smaller models, the state explosion problem quickly leads to performance problems.

Abstraction methods and improved model-checking techniques are directions of research that will hopefully lead to improvements with regard to the performance of the test-case generation. Part of the problem, however, is caused by inefficient and redundant calls to the model-checker during the test-case generation process.

For example, even though it is possible to detect equivalent mutants using a model-checker, this still requires an explicit call to the model-checker for each mutant and consumes a similar amount of time as the creation of a test-case. Consequently, time is spent on model-checking of mutants that do not contribute to the test-suite. Furthermore, experience has shown that a large number of mutants result in identical or subsumed test-cases.

In our experience, test-suites resulting from mutation based approaches are usually far from minimal with regard to the mutation score (i.e., percentage of mutant models that can be distinguished from the original model). A similar observation has been made in coverage oriented test-case generation with model-checkers [9]. With increasing model size the number of mutants and therefore test-cases increases. Consequently, a necessary countermeasure might be test-suite reduction, where a subset of an existing test-suite is heuristically selected, such that a certain coverage criterion is fulfilled with as few test-cases as possible.

In this paper, we present a simple approach to reduce the number of calls to the model-checker. The proposed solution represents model mutants using characteristic properties. These are used to identify equivalent mutants and to detect already covered mutants during test-case generation, thus excluding them from the test-case generation. The model-checker NuSMV [5] is used to illustrate the presented ideas. Resulting test-suites are smaller than when using a traditional approach, while the mutation score with regard to those mutants used for test-case generation is not changed. As the size of a test-suite is reduced, the effects on the fault sensitivity are empirically analyzed.
2. Test-Case Generation with Model-Checkers and Mutation

A model-checker is a tool that takes as input an automaton-based model of a system and a property specified with temporal logics, and then effectively examines the complete state space of the model in order to determine whether the property holds. Detected violations are illustrated with counterexamples, which in current model-checkers are implemented as linear sequences of states that begin in an initial state of the model and lead to some state such that the property violation is demonstrated by the sequence or the final state of the sequence.

Testing with model-checkers is based on the idea to interpret counterexamples as test-cases. The state of a model is described by a set of variables. Therefore, each state in a counterexample can be interpreted as a valuation of all variables. The values of variables that represent the input the system receives from its environment are used as test data, and the values of output variables that are calculated by the model are used as test oracle.

The use of mutation for model-checker based test-case generation was initially proposed by Ammann et al. [3]. As a model-checker takes as input an automaton-based model and a property, the creation of counterexamples can be forced by applying mutation to either of the two inputs. The automaton is not mutated directly; instead, the textual representation used as input to the model-checker is modified. The resulting mutants are checked against a given specification, and counterexamples illustrate specification violations. A slightly different approach is taken by Ammann and Black [1], who apply mutation to properties that “reflect” the transition relation of the model. The mutants of the reflected properties are then checked against the model, and counterexamples are used as test-cases. A kind of equivalence check is performed in the state-machine duplication approach by Okun et al. [10], who combine a mutant with its original model and then use the model-checker to search for paths that show how mutants affect output variables. In a related approach, mutation based testing with regard to safety properties is described by Ammann et al. [2].

In this paper, we consider methods that are based on mutation of the model [2, 3, 10]. The model-checker has to encode the state space of each mutant, e.g., using ordered BDDs. The time this takes depends on the model, the chosen model-checker and the techniques it implements. For example, our experience is that the model encoding phase in NuSMV [5] consumes a large part of the overall model-checking time. Consequently, equivalent mutants consume time for encoding and model-checking, but result in no test-cases. Furthermore, mutants often result in identical test-cases, or test-cases that are subsumed by other, longer test-cases. Such mutants are redundantly checked, because resulting test-cases do not contribute to the overall fault sensitivity of a test-suite.

Even after removal of subsumed test-cases, the size of a resulting test-suite is often very large. This can be especially problematic for regression testing, and therefore test-suite reduction [7] is sometimes applied to minimize the test-suite size. Here, time is consumed to create test-cases in the first place, and then more time is consumed trying to heuristically find a proper subset of the resulting test-cases, that is minimal with regard to a given coverage criterion.

As a countermeasure to these problems, we propose to use dedicated properties to minimize the number of model-checker calls, and consequently the performance and test-suite size are improved significantly.

3. Characteristic Properties

We use the formal framework for transition systems described by Rayadurgam and Heimdahl [12] in order to discuss the ideas presented in this paper. This framework can be applied to any transition based formalism, including NuSMV models. A system state is uniquely determined by the values of $n$ variables \{$x_1, x_2, \ldots, x_n$\}. Each variable $x_i$ has a domain $D_i$, and consequently the reachable state space of a system is a subset of $D = D_1 \times D_2 \times \ldots \times D_n$. The set of initial values for the variables is defined by a logical expression $\rho$. The valid transitions between states are described by the transition relation, which is a subset of $D \times D$. The transition relation is defined separately for each variable using logical conditions. For variable $x_i$, the condition $\alpha_{i,j}$ defines the possible pre-states of the $j$-th transition, and $\beta_{i,j}$ is the post-state condition. A simple transition for a variable $x_i$ is a conjunction of $\alpha_{i,j}$, $\beta_{i,j}$ and a guard condition $\gamma_{i,j}$: $\delta_{i,j} = \alpha_{i,j} \land \beta_{i,j} \land \gamma_{i,j}$.

The disjunction of all simple transitions for a variable $x_i$ is a complete transition $\delta_i$. The transition relation $\Delta$ is the conjunction of the complete transitions of all the variables \{$x_1, \ldots, x_n$\}. Consequently, a basic transition system is defined as follows:

**Definition 1 (Transition system)** A transition system $M$ over variables \{$x_1, \ldots, x_n$\} is a tuple $M = (D, \Delta, \rho)$, with $D = D_1 \times D_2 \times \ldots \times D_n$, $\Delta = \bigwedge_i \delta_i$, and the initial state expression $\rho$. For each variable $x_i$ there is a transition relation $\delta_i$, that is the disjunction of several simple transitions $\delta_{i,j} = \alpha_{i,j} \land \beta_{i,j} \land \gamma_{i,j}$, where $\alpha_{i,j}$, $\beta_{i,j}$, and $\gamma_{i,j}$ are pre-state, post-state, and guard conditions of the $j$-th simple transition of variable $x_i$.

Mutation is applied to the textual representation of a model, for example in the input language of the model-checker NuSMV. Conceptually, we can think of mutation as changes in a simple transition $\delta_{i,j}$ of a variable $x_i$. We
only consider first order mutants, i.e., mutants that differ in a single change from the original model.

Definition 2 (Mutant) A mutant \( M' = (D, \Delta', \rho) \) of transition system \( M = (D, \Delta, \rho) \) differs from \( M \) in exactly one simple transition. That is, \( \exists i,j : \delta_{i,j} \neq \delta'_{i,j} \land \forall k \neq j : \delta_{i,k} = \delta'_{i,k} \land \forall l \neq i : \delta_l = \delta'_l \).

Besides a formalism for system modeling, model-checkers use different property specification languages. In this paper, we use the Linear Temporal Logic (LTL) [11]. An LTL formula consists of atomic propositions, Boolean operators and temporal operators. In this paper, we use the operators \( \bigcirc \) and \( \Box \): The operator "\( \bigcirc \)" refers to the next state. E.g., "\( \bigcirc a' \)" expresses that \( a \) has to be true in the next state. "\( \Box \)" is the always operator, stating that a condition has to hold in all states of a trace.

In [12], the guard and the post-state condition can refer to the values of variables in the pre- and post-state. For simplicity, we assume that the guard \( \gamma_{i,j} \) refers only to the pre-state, and \( \beta_{i,j} \) only refers to the values of variables in the post-state. Without this assumption, \( \gamma_{i,j} \) or \( \beta_{i,j} \) can refer to the values of variables at two different states in time. A possible solution to this problem is to use "shadow" variables that track the values from the previous solution, as described by Ammann and Black [1]. A shadow variable for the previous state always has the previous value of the variable it shadows, therefore all variables that refer to the previous state in \( \beta_{i,j} \) can be replaced with their corresponding shadow variables. The same is possible the other way round for \( \gamma_{i,j} \). The disadvantage of such an approach is, that the state space is increased by the shadow variables. A different solution would be to access variables in the correct context specific to a \( \bigcirc \) operator. This, however, requires that the constraints \( \gamma_{i,j} \) and \( \beta_{i,j} \) can be split into two parts \( \gamma'_{i,j}, \gamma''_{i,j} \) and \( \beta'_{i,j}, \beta''_{i,j} \), respectively, such that: \( \beta_{i,j} = \beta'_{i,j} \land \beta''_{i,j} \), and \( \gamma_{i,j} = \gamma'_{i,j} \land \gamma''_{i,j} \), respectively.

From the definition of a transition system it follows that any \( M \) which has the transition \( \delta_{i,j} = \alpha_{i,j} \land \beta_{i,j} \land \gamma_{i,j} \) has to satisfy the following property, which expresses that always (\( \Box \)) when pre-state \( \alpha_{i,j} \) and guard condition \( \gamma_{i,j} \) are true, the post-state condition \( \beta_{i,j} \) has to be true in the next state (\( \bigcirc \)):

\[
\psi_1 := \Box ((\alpha_{i,j} \land \gamma_{i,j}) \rightarrow \bigcirc \beta_{i,j}) \tag{1}
\]

A mutation can have different effects. If the mutation has no effect on the transition system, then the mutant is referred to as equivalent. We define a mutant to be equivalent, if \( \Delta \leftrightarrow \Delta' \). If this is not the case, then the mutant is inequivalent. If a mutant is inequivalent, this means that at least one of the following cases applies: (1) The mutant model contains behavior that the original model does not allow. (2) The original model contains behavior that the mutant model does not allow. In the first case the mutant does not satisfy \( \psi_1 \). In the latter case the original model does not satisfy \( \psi_2 \).

Note, that this definition of mutant equivalence does not take into account that some behaviors of the mutant might not be observable. Theoretically, a model can contain not only input and output, but also hidden variables. A change might not be observable if only hidden variables are affected by the change, but the change does not propagate to an observable output.

We use the language of the model-checker NuSMV in this paper to describe transition systems. Listing 1 is an example of how a transition relation of a variable can be defined in NuSMV. Each entry of the \texttt{case} statement corresponds to a simple transition \( \delta_{i,j} \) for variable \( x_i \). The conditions \( \phi_j \) correspond to the conjunctions of \( \alpha_{i,j} \) and \( \gamma_{i,j} \), and \( \beta_{i,j} \) corresponds to the atomic proposition \( x'_i \), where \( x'_i \) denotes \( x_i \) in the post-state. There is an ordering on the conditions, therefore strictly speaking, the conditions are to be interpreted as \( \phi_1 \land \neg \phi_2 \land \phi_3 \), etc., and in the general case as: \( (\land_{1 \leq j < k} \neg \phi_j) \land \phi_k \).

assign

\texttt{next}(x_i) := \texttt{case}
\phi_1 : \xi_1; \\
\phi_2 : \xi_2; \\
... \\
\texttt{esac;}

Listing 1. \texttt{ASSIGN} section of an SMV file. The transition relation of a variable \( x_i \) is given as a set of conditions \( \phi_j \) and next values \( \xi_j \).

The property \( \psi_1 \) can be instantiated for a model \( M \) corresponding to Listing 1 and the \( k \)-th transition of variable \( x_i \) as follows:

\[
\psi_1^{x_i,k} := \Box ((\land_{1 \leq j < k} \neg \phi_j) \land \phi_k) \rightarrow \bigcirc x_i = \xi_k
\]

The property \( \psi_2 \) can be instantiated similarly. For example, assuming a mutant of Listing 1 that uses \( \phi_2' \) instead of \( \phi_2 \), \( \psi_2 \) is instantiated as follows:

\[
\psi_2^{x_i,2} := \Box ((\neg \phi_1 \land \phi_2') \rightarrow \bigcirc x_i = \xi_2)
\]
Ideally, we want a property \( \psi \) for \( M' \) which identifies an inequivalent mutant simply if \( M \) does not satisfy \( \psi \):

\[
M \not\models \psi \iff M \neq M'
\]  

(3)

If \( M' \) is equivalent to \( M \), then \( M \models \psi_2 \). If \( M \not\models \psi_2 \), then we know that \( M \) and \( M' \) are inequivalent. This corresponds to the left-to-right implication in equation 3. The right-to-left implication is not fulfilled by \( \psi_2 \), because a mutant can be inequivalent although \( M \models \psi_2 \). This is the case if the mutated simple transition simply restricts the set of possible transitions. Therefore, the right-to-left implication in equation 3 requires another property to cover all cases:

\[
\psi_3 := \Box((\alpha_{i,j} \land \gamma_{i,j} \land \neg(\alpha'_{i,j} \land \gamma'_{i,j})) \rightarrow \neg \beta_{i,j})
\]  

(4)

The property \( \psi_3 \) covers those cases where the mutant removes possible transitions from the model. For our example NuSMV mutant, this can be instantiated as follows:

\[
\psi_{3}^{x_1,2} := \Box((\neg \phi_1 \land \phi_2 \land \neg \phi_2) \rightarrow \neg x_1 \neq x_2)
\]

Definition 3 (Characteristic Property) The characteristic property \( \psi \) of mutant \( M' = (D, \Delta', \rho) \) of transition system \( M = (D, \Delta, \rho) \) is \( \psi := \psi_2 \land \psi_3 \), with \( \psi_2 \) and \( \psi_3 \) given in equations 2 and 4, respectively.

The characteristic property \( \psi \) of a mutant is the conjunction of \( \psi_2 \) and \( \psi_3 \). A mutant is inequivalent, iff \( M \not\models \psi \), which means that \( \psi \) fulfills equation 3. The idea of characteristic properties is similar to the reflected properties described by Ammann and Black [1]. However, not all equivalent or killed mutants can be represented or detected with reflected properties. The conditions of a case statement are unfolded with a process the authors call expoundment, similarly like in the NuSMV instantiation of \( \psi_1 \), \( \psi_2 \), and \( \psi_3 \). Therefore, reflection together with expoundment results in properties similar to \( \psi_2 \). As described above, this is not sufficient to identify all possible inequivalent mutants.

4. Using Characteristic Properties to Optimize Test-Case Generation

Eliminating Equivalent Mutants: Model-checkers can be used to perform equivalence checks in order to detect equivalent mutants. For example, the state-machine duplication method presented by Okun et al. [10] combines a model and a mutant, and then queries the model-checker whether there exists a path such that the outputs of the model and the mutant differ when using the same inputs.

The overall time consumed by this equivalence check can be reduced by using characteristic properties (Definition 3). For each mutant, a characteristic property is created. Then, the original model is checked against these properties. This improves the performance because (1) the model encoding phase only has to be done once and (2) the state space is smaller because the model is not modified. For example, using the method of Okun et al. [10], the state space is twice the size of that of the normal model. The performance can be further improved because it is sufficient to detect that a model does not satisfy the property, but counterexamples are not necessary. For example, when using the symbolic model-checker NuSMV, counterexample calculation, which takes considerable time, can be deactivated. If counterexamples are created, this process can be considered as an improvement of [1] that results in more test-cases.

As noted earlier, an inequivalent mutant might not be observably different. When using characteristic properties to detect equivalent mutants this is acceptable: Only characteristic properties of really equivalent mutants are satisfied by a model, while there is a small chance that an identified inequivalent mutant is in fact equivalent. The worst case is that not as many mutants are excluded from the test-case generation as might be possible.

Detecting Killed Mutants: Even if a mutant is not equivalent to the original model, it might still be unnecessary to use it for test-case generation. Often, different mutants result in identical test-cases. On the one hand, this can be because mutants cannot only be equivalent to the original model but also amongst themselves. On the other hand, different mutants can also lead to identical test-cases, and as our experience shows often do so. Another related performance drawback is caused by similar test-cases, where one is a prefix of another and can therefore be omitted.

Test-suites also tend to get rather large, which might require test-suite reduction as a post-processing step. There are approaches of how to eliminate redundant test-cases for a complete test-suite (e.g., winnowing [1]), but ideally it should be avoided to call the model-checker on mutants unnecessarily, before creating redundant test-cases.

The characteristic properties introduced in the previous section can be used to solve these problem. A test-case kills a mutant, if it can distinguish between the original model and the mutant. According to our definition of inequivalent mutants, a test-case can distinguish between original model and mutant only if it covers a modified transition. Consequently, a test-case that kills a mutant is also inconsistent with the characteristic property of that mutant.

The time consumed by the test-case generation can be reduced by monitoring characteristic properties. Each time a test-case is created it is determined which characteristic properties are violated. A violated characteristic property shows that the mutant is already killed, and the mutant and its characteristic property can be removed. Note that this method relies on our definition of equivalence. If a mutant
is only internally inequivalent but the difference is not observable, then it might be excluded from test-case generation although there is no test-case that propagates the effect of the mutation to an observable output. We believe that in scenarios where this is not acceptable, no test-suite minimization at all can be applied.

In the worst case, \( m \) mutants result in \( m \) unique test-cases, and no test-case kills more than one mutant. Then, \( m - 1 \) properties have to be analyzed after the first test-case, \( m - 2 \) after the second, etc. Consequently, there is an upper bound of \( m \cdot (m + 1)/2 \) such analyses. In order to achieve a performance improvement, monitoring \( m \cdot (m + 1)/2 \) properties has to be faster than model-checking \( m \) mutants.

There are different techniques that can be used for the monitoring. For example, test-cases can be formulated as verifiable models and then model-checked against the characteristic properties. Such a process is used for coverage analysis with model-checkers. Test-cases are represented as models by adding a special state counter variable and by setting all other variables depending only on the value of this state counter [1]. A different approach avoiding the use of a model-checker is taken by rewriting techniques used in property monitoring in the field of runtime verification.

We propose to use LTL rewriting, which is an efficient method to monitor LTL formulas. For example, the method described by Havelund and Rosu [8] can be adapted to apply to characteristic properties. It defines a set of simple rewriting rules that are based on the states of an execution trace. An LTL formula is rewritten with every state of a trace, and if it results in a contradiction, then a property violation is detected. Such an approach is faster than model-checking; e.g., Havelund and Rosu claim their implementation is capable of 3 million rewritings per second. In [6] we used such a rewriting approach in the context of coverage based test-case generation with model-checkers.

**Optimized Test-Case Generation:** The normal approach to test-case generation with model-checkers and mutation is to create a set of mutants, and then call the model-checker on the mutants with either requirement or dedicated properties. Figure 1 extends this scheme by creating characteristic properties together with the mutant models. The characteristic properties can be used to eliminate equivalent mutants before starting the test-case generation, or before starting a new model-checker call on the mutating. Counterexamples are not only stored as test-cases but also applied to the characteristic properties with rewriting techniques. Any characteristic property that is not satisfied by a counterexample represents a killed mutant, therefore the corresponding mutant does not need to be included in the test-case generation.

As model-checking the mutants is generally slower than monitoring properties, this procedure will improve the performance for all but very small models. Less mutants are model-checked because redundant test-cases are avoided. At the same time, the size of resulting test-suites is reduced. The mutation score with regard to the mutants used for test-case generation is not changed.

The order in which mutants are selected for test-case generation has an influence on the number of necessary test-cases, as different test-cases kill different sets of mutants. As shown in [9], the minimal test-suite problem is NP-hard. Therefore, creation of an minimal test-suite is not feasible. In practice, random selection of mutants works very well. Heuristic selection might be considered in future research.

The performed minimization is related to the test-suite reduction problem [7], where a suitable subset of given test-suite is computed. Consequently, the drawbacks of test-suite reduction also apply to this approach: While maximal mutation score with regard to the mutants used for test-case generation is still achieved, every avoided test-case might reduce the overall fault sensitivity.

![Figure 1. Optimized test-case generation.](image)

**5. Empirical Results**

To evaluate the described ideas, a simple prototype implementation was applied to a manually created model of a windscreen wiper controller, provided by Magna Steyr. NuSMV reports a total of \( 2^{44 \cdot 8727} \) states, 93 BDD variables and 174762 BDD nodes after model encoding. All experiments were conducted on a PC with Intel Core Duo T2400 processor and 1GB RAM.

As an example test-case generation method we used the state-machine duplication approach described by Okun et al. [10] described earlier, as it does not depend on requirement properties. The following mutation operators were used (see [4] for details): STA, SNO, MCO, LRO, RRO.

Table 1 lists the results with regard to test-suite size and creation time for regular test-case generation (Normal) and when applying the optimizations described in the previous section (Optimized). The test-suite size is given as the number of unique test-cases. In average, 62% of all test-cases created with the normal approach are redundant, but only 9% in the case of the optimized version.

Table 2 shows the effects on the test-suite quality. The mutation score is measured using a set of 4038 model mutants (including equivalent mutants and also different muta-
The number of redundant test-cases would be further decreased. Bly longer test-case. The consequence would be, that the mutant. It is likely that the test-case can be extended with a short sequence, rather than requiring a new, possibly longer test-case. The consequence would be, that the number of redundant test-cases would be further decreased. Traditionally, it is done using heuristic approaches as a post-processing step.

A further optimization of the test-suite size is conceivable by fully exploiting the information produced by the rewriting. Currently, rewriting is only used to detect when a property is not satisfied. If a formula is changed but not shown to be false, this suggests that the considered test-case affects the property and thus the mutant, but does not yet kill the mutant. It is likely that the test-case can be extended with a short sequence, rather than requiring a new, possibly longer test-case. The consequence would be, that the number of redundant test-cases would be further decreased. A further optimization of the test-suite size is conceivable by fully exploiting the information produced by the rewriting. Currently, rewriting is only used to detect when a property is not satisfied. If a formula is changed but not shown to be false, this suggests that the considered test-case affects the property and thus the mutant, but does not yet kill the mutant. It is likely that the test-case can be extended with a short sequence, rather than requiring a new, possibly longer test-case. The consequence would be, that the number of redundant test-cases would be further decreased.

### 6. Conclusions

In this paper, we have considered the sources of bad performance in test-case generation using model-checkers and mutation, and proposed a solution that avoids some of the identified problems. The idea is to represent model mutants as characteristic properties, which can then be used to identify equivalent mutants more quickly, and to avoid creation of redundant test-cases. In addition to the improvement of the test-case generation performance, the resulting test-suites are minimized while still achieving the highest possible model-based mutation scores. This reduction in the test-suite size is often important when test-case execution is costly. Traditionally, it is done using heuristic approaches as a post-processing step.

A further optimization of the test-suite size is conceivable by fully exploiting the information produced by the rewriting. Currently, rewriting is only used to detect when a property is not satisfied. If a formula is changed but not shown to be false, this suggests that the considered test-case affects the property and thus the mutant, but does not yet kill the mutant. It is likely that the test-case can be extended with a short sequence, rather than requiring a new, possibly longer test-case. The consequence would be, that the number of redundant test-cases would be further decreased.

### Table 1. Results of the test-case generation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Mutants</th>
<th>Normal</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Size</td>
<td>Time</td>
<td>Size</td>
</tr>
<tr>
<td>STA</td>
<td>768</td>
<td>343</td>
<td>29m48s</td>
</tr>
<tr>
<td>SNO</td>
<td>630</td>
<td>331</td>
<td>20m3s</td>
</tr>
<tr>
<td>MCO</td>
<td>274</td>
<td>202</td>
<td>10m26s</td>
</tr>
<tr>
<td>LRO</td>
<td>584</td>
<td>305</td>
<td>43m40s</td>
</tr>
<tr>
<td>RRO</td>
<td>510</td>
<td>178</td>
<td>14m16s</td>
</tr>
</tbody>
</table>

### Table 2. Coverage and mutation scores.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Normal</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Transition</td>
<td>Score</td>
</tr>
<tr>
<td>STA</td>
<td>100%</td>
<td>80.26%</td>
</tr>
<tr>
<td>SNO</td>
<td>100%</td>
<td>79.47%</td>
</tr>
<tr>
<td>MCO</td>
<td>100%</td>
<td>87.64%</td>
</tr>
<tr>
<td>LRO</td>
<td>96.34%</td>
<td>87.64%</td>
</tr>
<tr>
<td>RRO</td>
<td>96.34%</td>
<td>72.76%</td>
</tr>
</tbody>
</table>

### References


