CS4052 Coursework2: A Simple Model Checker

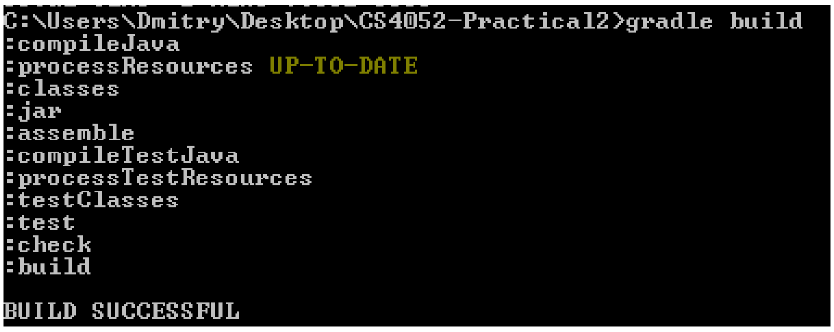
Report

Introduction

The goal of this practical work is to design and implement a model checker for an action and state-based logic *asCTL* interpreted over transition systems. Additionally, the model checker must support fairness in the sense that any constraints can be added to restrict the set of paths considered for verification. If a model checker computes that the logical formula doesn't hold for a certain model, the system must generate the trace (counterexample) where the formula indeed doesn't hold.

Instructions of how to build the project

The project can be built only using Java 1.8 (not lower). The project uses the build automation system called "Gradle". To build the project (this includes compiling, assembling and executing automated tests), run the command *gradle build* (Image 1):



```
C:\Users\Dmitry\Desktop\CS4052-Practical2>gradle build
:compileJava
:processResources UP-TO-DATE
:classes
:jar
:assemble
:compileTestJava
:processTestResources
:testClasses
:test
:check
:build

BUILD SUCCESSFUL
```

Image 1. Gradle build command.

Structure of the code

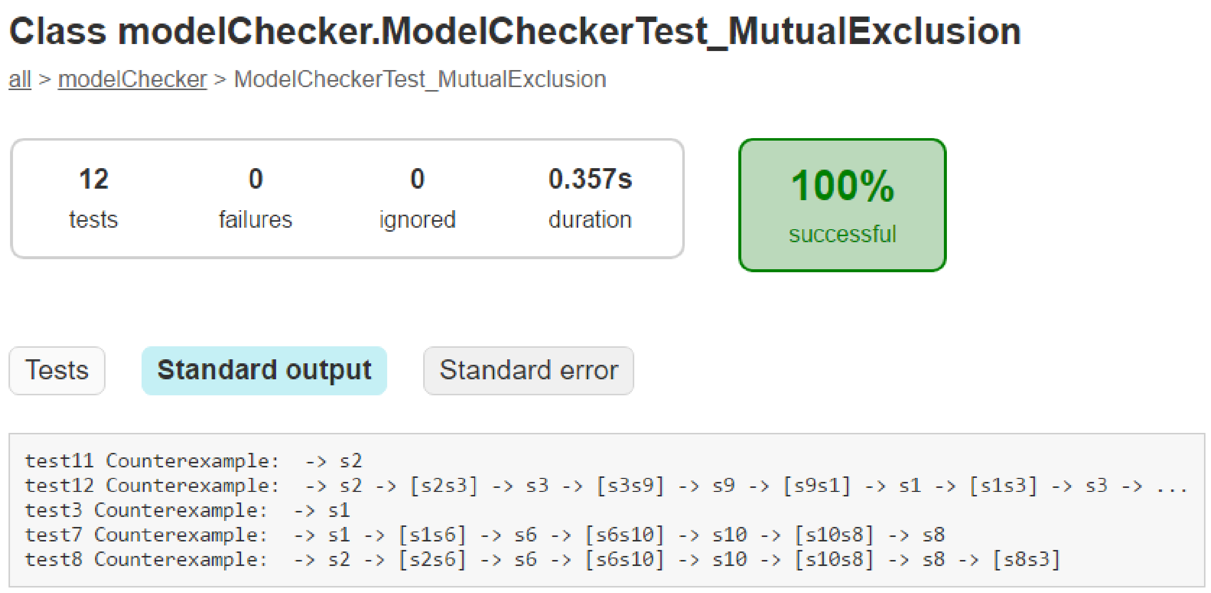
The following components were provided by the lecturer:

1) Formula parser: "formula", "formula.pathFormula", "formula.pathFormula" packages.
2) Model builder: "model" package

Model Checker itself is in "modelChecker" package. It consists of several components:

1) *ENFGenerator* class is responsible for formula conversion from the raw form to the ENF.
2) *SubsetCalculatorImpl* class is responsible for computing *Sat(Φ)* – the subset of states of the model where *Φ* holds.
3) *TraceGenerator* class is responsible for generating counterexamples in case if a model checker returns that a formula is not satisfied.

All the test data and automated tests are in */src/test/* folder. The test data is in */resources/* folder. The models in Json format are in */models/* folder; formulas in Json format are in */formulas/* folder. Each test formula file name corresponds to the certain model file: for example, *formula1_3* means that this formula is a test formula number three for a model number one. Each test class corresponds to the model as well: *ModelCheckerTest_1* tests model 1, *ModelCheckerTest_2* tests model 2 and so on. If a model checker returns false, a counterexample is generated. All counterexamples can be found in */build/reports/test/junit/index.html*; go to the test class that interests you, counterexamples for this test suit are in the tab "Standard output" (Image 2).


Class modelChecker.ModelCheckerTest_MutualExclusion

all > modelChecker > ModelCheckerTest_MutualExclusion

12	0	0	0.357s	100%
tests	failures	ignored	duration	successful

Tests | Standard output | Standard error

```
test11 Counterexample:  -> s2
test12 Counterexample:  -> s2 -> [s2s3] -> s3 -> [s3s9] -> s9 -> [s9s1] -> s1 -> [s1s3] -> s3 -> ...
test3 Counterexample:  -> s1
test7 Counterexample:  -> s1 -> [s1s6] -> s6 -> [s6s10] -> s10 -> [s10s8] -> s8
test8 Counterexample:  -> s2 -> [s2s6] -> s6 -> [s6s10] -> s10 -> [s10s8] -> s8 -> [s8s3]
```

Image 2. Display of counterexamples

Logic *asCTL*: Grammar and Semantics

A model checker for an action and state-based logic, called *asCTL*, interpreted over transition systems was developed. This section details the grammar of *asCTL* as well as the semantics that were assumed for the development of the model checker.

Let Φ be a state formula, ϕ a path formula, $p \in AP$ an atomic proposition, $\alpha \in Act$ an action, and $A, B \subseteq Act$ subsets of actions, the required *asCTL* grammar is defined as follows.

A state formula is given by:

$$\Phi ::= true \mid p \mid \neg\, \Phi \mid \Phi \,\wedge\, \Phi \mid \exists\phi \mid \forall\phi$$

And a path formula is given by:

$$\phi ::= \Phi \;\; \mathcal{U}_B \, \Phi \mid \Phi_A \mathcal{U}_B \, \Phi \mid \Phi_A \mathcal{U} \;\; \Phi \mid \Phi \;\; \mathcal{U} \;\; \Phi$$

The semantics of the *Until* operator were implemented as stated in the specification, with the following special cases and assumptions:

- The cases where $A, B = \emptyset$ or $A, B = Act$ are handled in the same way as if the A or B where not specified: all the actions are taken as if they satisfy the logic formula (less restrictions are imposed).
- $\Phi \ \mathcal{U} \ \Phi$ is handled as **strong** until. The *weak until* operator is out of the scope of the implemented model checker.

Additionally, the following extra action-indexed path operators are defined:

Operator	**Grammar**	**Semantics**
Next	$\mathcal{X}_B\Phi$	holds for a state s_1 iff $s_{1+1} \vDash \Phi$ and $\alpha_1 \in B$
Eventually	${}_A\mathscr{F}_B\Phi$	holds for a path π iff there is a state $s_i \vDash \Phi$, $\alpha_i \in B$, and until then all the transitions α_k with $0 \leq k < i$, $\alpha_k \in A$. If one of the sets A or B is missing, then fewer restrictions are being imposed, as with the specified Until operator.
Always	$\mathcal{G}_B \ \Phi$	holds if all the states $s \ \ \vDash \Phi$, and all the actions $\alpha \ \ \in B$

asCTL ENF Existential Normal Form

The first step of the model checker algorithm is to obtain an equivalent ENF (Existential Normal Form) of the input *asCTL* formula. In order to do so, the equivalence rules for CTL are applied, with the following additional equivalences in order to comply with the *asCTL* grammar:

asCTL	**ENF**
$\forall\boldsymbol{\mathcal{X}_B\Phi}$	$\neg\exists\ \mathcal{X}_B\neg\Phi$
$\forall(\boldsymbol{\Phi_A\mathcal{U}_B\ \Psi})$	$\neg\exists\big(\neg\Psi_A\mathcal{U}_B(\neg\Phi\wedge\neg\Psi)\big)\wedge\neg\exists\ \mathcal{G}_B\ \neg\Phi$
$\forall\boldsymbol{\mathcal{G}_B\Phi}$	$\neg\exists\ (true\ \mathcal{U}_B\ \neg\Phi)$
$\forall_{\boldsymbol{A}}\boldsymbol{\mathcal{F}_B\Phi}$	$\neg\exists\mathcal{G}_B\neg\Phi$
$\exists_{\boldsymbol{A}}\boldsymbol{\mathcal{F}_B\Phi}$	$\exists\ (true_A\mathcal{U}_B\Phi)$

With that, the set of *asCTL* formulae that are in existential normal form is given by:

$$\Phi ::= true \mid p \mid \neg\ \Phi \mid \Phi\wedge\Phi \mid \exists\mathcal{X}_B\Phi \mid \exists\mathcal{X}\ \ \Phi \mid$$

$$\exists(\ \Phi\ \ \mathcal{U}_B\ \Phi) \mid \exists(\Phi_A\mathcal{U}_B\ \Phi) \mid \exists(\Phi_A\mathcal{U}\ \ \ \Phi) \mid \exists(\Phi\ \ \mathcal{U}\ \ \ \Phi) \mid \exists(\mathcal{G}\ \ \Phi) \mid \exists(\mathcal{G}_B\Phi)$$

Model checking algorithm

Before introducing the model checking algorithm, several notations must be explained. Let $TS = (S, Act, ->, I, AP, L)$ be a transition system representing out model. Then:

1) $Sat(\Phi)$ is the subset of S, where Φ holds.
2) $Sat(_A\Phi)$ is the subset of $Sat(\Phi)$, such that for each $s \in Sat(_A\Phi)$ there exists an incoming transition $t \mid t \in A, A \in Act$
3) $Sat(\Phi_B)$ is the subset of $Sat(\Phi)$, such that for each $s \in Sat(\Phi_B)$ there exists an outgoing transition $t \mid t \in B, B \in Act$.

Our model checking algorithm is the modification of the classical CTL model checking approach [1].

It consists of several steps:

1) The model is parsed and validated. If the model has terminal states, the system artificially adds the loop transitions to those states.
2) The formula that is fed into our model checker must be in ENF. If it is not, it is then converted to ENF using CTL conversion rules.
3) The formula is parsed into a *formula parse tree*.
4) Given a transition system *TS* and formulaΦ, the set $Sat(\Phi)$ is recursively computed (starting from the leaf nodes of formula parse tree), and Φ is satisfied for *TS* if and only if $I \subseteq Sat(\Phi)$, where I is a set of initial states.

How do we compute $Sat(\Phi)$? For all *asCTL* formulas Φ, Ψ over *AP* it holds that:

1. $Sat(true) = S$
2. $Sat(a) = \{s \in S \mid a \in L(s)\}$, *for any* $a \in AP$
3. $Sat(\Phi /\backslash\ \Psi) = Sat()\Phi \cap Sat(\Psi)$
4. $Sat(!\Phi) = S \backslash Sat\Phi$
5. $Sat(\exists X_A\ \Phi) = \{s \in S \mid Post(s) \cap Sat(_A\Phi) \neq \varnothing\}$
6. $Sat(E(\Phi\ _AU_B\ \Psi))$ *is the smallest subset T of S, such that:*
   1) $Sat(_B\Psi) \subseteq T$
   2) $s \in Sat(\Phi_A)$
   3) $Post_{rightactions}(s) \subseteq Sat(_B\Psi)$, *such that* $\{p_{right} \in Post_{rightactions}(s) \mid \exists\ t \in S \mid$ *transition* $t\text{->}p_{right}$ *is labeled with B*$\}$
   4) $Post_{leftactions}(s) \subseteq Sat(\Phi_A)$, *such that* $\{p_{left} \in Post_{leftactions}(s) \mid \exists\ t \in S \mid$ *transition* $t\text{->}p_{left}$ *is labeled with A*$\}$
   5) $Post_{actions}(s) = Post_{rightactions}(s) \cup Post_{leftactions}(s)$
   6) $Post_{actions}(s) \cap T \neq \varnothing$ *implies* $s \in T$
7. $Sat(\exists\Box_A\Phi)$ *is the largest subset T of S, such that:*
   1) $T \subseteq Sat(_A\Phi)$

2) $s \in T$ *implies* $Post(s) \cap T \neq \emptyset$

Fairness Constraint

An additional *asCTL* statement is taken as a fairness constraint. If a fairness constraint is provided, then the algorithm will only take into consideration the paths that satisfy that constraint for the verification of the *asCTL* formula in the transition system. Let η be the fairness constraint, and Φ the *asCTL* formula to be verified, $Sat(\Phi)$ the set of all states that satisfy Φ and $Sat(\eta)$ the set of all states that satisfy η, it follows that $TS \vDash_{\eta} \Phi$ if and only if $I \subseteq Sat(\Phi) \wedge I \subseteq Sat(\eta)$.

Given that definition of fairness in *asCTL*, the model checking algorithm is called with the new formula $\Phi \wedge \eta$.

asCTL Counterexamples

If the *asCTL* formula does not hold in the model of the transition system for which it is being tested, a counterexample path will be printed to Java's standard output with the following formats:

Counterexample	**Interpretation**
-> s1	The initial state "s1" does not satisfy Φ
-> s1 -> [] -> s2	The path starting from "s1" and going to "s2" without an action in the transition does not satisfy Φ
-> s1 -> [a] -> s2	The path starting from "s1" and going to "s2" with the action "a" in the transition does not satisfy Φ
-> s1 -> [a] -> s2 -> [b] -> [s3]	The path starting from "s1" and going to "s2" with the action "a", and to "s3" with the action "b" in the transition does not satisfy Φ
-> s1 -> [a] -> s2 -> [b] -> s3 -> [c] s2 -> …	The "…" denotes a loop from the last state to its previous appearance in the path. The specified path, with a loop s2, s3, s2, s3, s2… does not satisfy Φ

An important characteristic of the counterexamples that are being generated is that they are generated according to the "top level" state formula operator of the parse tree of the *asCTL* formula in ENF. For example, for the query $(\exists(\mathcal{X}_B \Phi)) \wedge \neg\Phi$, the $\wedge$ is the top level operator of the parse tree, and an initial state where this AND operator does not hold is going to be returned. A counterexample for either the left part or the right part of the AND operator is not generated.

The following table summarizes the counterexamples generated for each of the possible top level nodes of the parse tree:

Top Level node	**Counterexample returned**
$\boldsymbol{p}$	A state s where $s \in I, and\ p \notin L(s)$
$\neg\,\boldsymbol{\Phi}$	A state s where $s \in I$, and $s \vDash \Phi$
$\boldsymbol{\Phi} \wedge \boldsymbol{\Phi}$	A state s where $s \in I$, and $s \vDash \neg(\Phi \wedge \Phi)$
$\exists \boldsymbol{\mathcal{X}_B \Phi}$	A path π from an initial state s to $s_1$where $s_1 \vDash \neg\,\Phi$, or the transition from s to s_1, α, $\alpha \notin B$. Note that a single counterexample path is returned. If B is not given, then the restriction is dropped and the transition action is not checked.
$\exists(\boldsymbol{\Phi_A \mathcal{U}_B \Psi})$	A path π from an initial state s to s_n is returned where one of the following statements is valid: 1. $\pi \vDash \mathcal{G}\ (\Phi \wedge \neg\,\Psi)$ 2. $\pi \vDash (\Phi \wedge \neg\,\Psi)\mathcal{U}(\neg\Phi \wedge \neg\,\Psi)$ 3. $\pi \vDash (\Phi\ \mathcal{U}\ \Psi)$, but the transition action $\alpha_k, for\ k = n-1$ satisfies that $\alpha_k \notin B$ 4. $\pi \vDash (\Phi\ \mathcal{U}\ \Psi)$, but a transition $\alpha_i, for\ 1 \le i < n-1$, satisfies that $\alpha_i \notin A$ Note that if B is not present in the formula, then the condition 3 is dropped, and if A is not present, then the condition 4 is dropped. A single counterexample is returned.
$\forall \boldsymbol{\mathcal{X}_B \Phi}$	An initial path π is returned where one of the following statements is valid: 1. $\pi \vDash \mathcal{X}_B \neg\Phi$ 2. $\pi \vDash \mathcal{X}\ \Phi$, but $\alpha \notin B$ If B is not present, then the statement 2 is dropped.
$\forall(\boldsymbol{\Phi_A \mathcal{U}_B \Psi})$	An initial path path π is returned where one of the following statement is valid, according with the rules defined for the counterexample of $\exists(\Phi_A \mathcal{U}_B\ \Psi)$: 1. $\pi \vDash \neg(\Phi_A \mathcal{U}_B\ \Psi)$

For the rest of the possible *asCTL* top level operators, a counterexample is returned for their corresponding ENF formula, as defined in the following table:

Top Level node	**ENF Formula**	**Counterexample returned**
$\forall \mathcal{G}_B \Phi$	$\neg\exists\ (true\ \mathcal{U}_B\ \neg\Phi)$	An initial state s where $s \vDash\ true\ \mathcal{U}_B\ \neg\Phi$
$\forall_A \mathcal{F}_B \Phi$	$\neg\exists\mathcal{G}_B\neg\Phi$	An initial state s where $s \vDash \mathcal{G}_B\neg\Phi$
$\exists_A \mathcal{F}_B \Phi$	$\exists\ (true_A \mathcal{U}_B \Phi)$	An initial state s where $s \vDash \neg(true_A \mathcal{U}_B \Phi)$

Not that for those operators, a single initial state is returned, rather than a path that invalidates the "nested" path formula.

Testing

The model checker was tested on six models of varying complexity and topologies. The set of test formulas is representative enough to cover most of the possible combinations of logic operators. Overall, 44 JUnit tests were implemented and the model checker passed all of them. Here we'll describe the testing of the model representing the mutual exclusion algorithm in more details (Image 3).

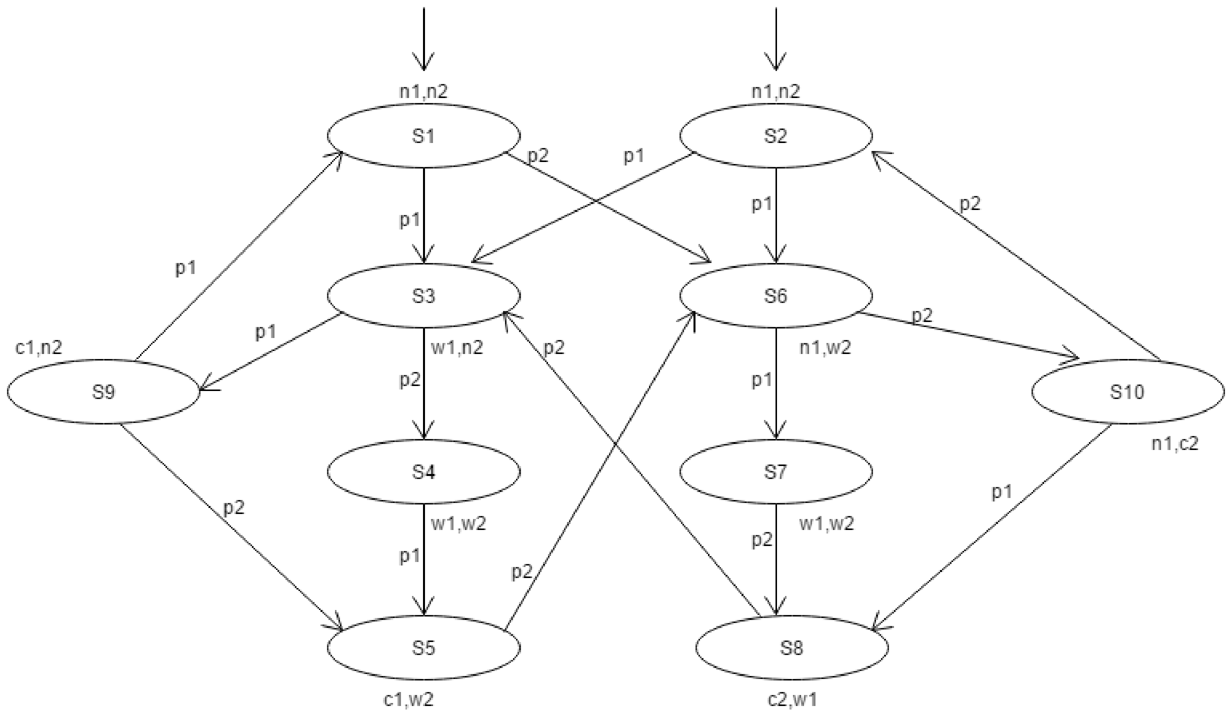


Image 3. Transition system for mutual exclusion algorithm.

p1 and p2 are sets of actions:

p1 = {s9s1, s3s9, s1s3, s4s5, s5s6, s2s3, s6s7, s10s8};

p2 = {s9s5, s3s4, s1s6, s8s3, s7s8, s2s6, s6s10, s10s2}.

Test cases are presented in Table 1. For example, test 5 is a safety property for a mutual exclusion algorithm – process1 and process2 never simultaneously have access to their critical sections. If formula doesn't hold for the model, a counterexample is generated.

№	Formula	Constraint	Expected Output	Actual Output	Counterexample
1	True	Null	True	True	
2	n1	Null	True	True	
3	!n2	Null	False	False	S1
4	!(c1 && c2)	Null	True	True	
5	AG(!(c1 && c2))	Null	True	True	
6	E(n1 U w1)	Null	True	True	
7	E(n1 Up2 w1)	Null	False	False	S1-s6-s10-s8
8	A((n1 || n2) p1Up2 (w1 && w2))	Null	False	False	S2-s6-s10-s8
9	A((n1 || n2) U (w1 && w2))	Null	False	False	
10	E(n2 p2U w2)	Null	True	True	
11	E(n1 U w1)	!n2	False	False	S2
12	A(n2 p2U w2)	Null	False	False	S2-s3-s9-s1-s3-…

Table 1. Test cases.

The Json representation of this model: */test/resources/models/model_mutual.json*. The Json representation of test formulas for this model can be found in */test/resources/formulas/*. The formulas have *formula_mutual_X* format. The test class for this model is called *ModelCheckerTest_MutualExclusion*.

Extensions

In addition to the specified requirements, the following extensions were implemented:

Extra state operators: The model checker allows the user to specify formulae using the following operators, without the need to specify them in their equivalent (and more complicated) formulae:

- **Or**: the or operator can be written in terms of negations and the and operator, but the model checker accepts the or command as input.

Extra path operators: the following operators are handled and verified in addition to the ones specified: next, always and eventually. The user is able to specify the formulae using these operators instead of their more complicated equivalences.

Model validation: The asCTL model checking algorithm implemented works with models without terminal states. If the specified transition system contains terminal states, the model checker will detect that and generate a transition from that state to itself, without any action labels, in order to be able to perform the model checking.

Note that, with the extensions and the algorithm implemented, our model checker is able to validate any CLT formulae, as well as any asCTL formulae with the semantics specified in previous sections.

Conclusion

As a result of this practical, a model checker for *asCTL* was designed and fully implemented. Given a transition system and *asCTL* formula, a model checker computes whether the formula is satisfied or not over the transition system. If it is not satisfied, the correspondent trace is returned to prove it. Besides, constraints can be added so that a model checker restricts the set of paths for verification. Additionally, a couple of improvements were implemented: a model validation and a formula conversion algorithm (conversion from non-ENF form to ENF). The developed model checker was properly tested by using models of various topologies and formulas with various combinations of logical operators.

References

[1] C. Baier and J.-P. Katoen. Principles of Model Checking. The MIT Press, 2008.