

Wikiprint Book

Title: jpf-statechart Translation Rules

Subject: Java Path Finder - StatechartTranslationGuidelines

Version: 1

Date: 03/07/2013 11:04:44 PM

Table of Contents

TracNav	3
Introduction	3
Installing JPF	3
User Guide	3
Developer Guide	3
MJI	3
Projects	3
About	4
jpf-statechart Translation Rules	4
Translation Rules	5

TracNav

- [JPFWiki - Welcome Page](#)

Introduction

- [What is JPF](#)
- [Testing vs model checking](#)
- [Random Example](#)
- [Race Example](#)
- [JPF classification](#)

Installing JPF

- [System requirements](#)
- [Download snapshots](#)
- [Download repositories](#)
- [Create site.properties](#)
- [Install NetBeans IDE plugin](#)
- [Install Eclipse IDE plugin](#)
- [Building and testing](#)

User Guide

- [Application Types](#)
- [JPF Components](#)
- [Configuring JPF](#)
- [Running JPF](#)
- [JPF Output](#)
- [The JPF API](#)

Developer Guide

- [Design](#)
- [Choice Generator](#)
- [Partial Order Reduction](#)
- [Attributes](#)
- [Listener](#)

MJI

- [Mangling for MJI](#)
- [Bytecode Factory](#)
- [Logging](#)
- [Report](#)
- [Embedded](#)
- [JPF tests](#)
- [JPF project layout](#)
- [Create a JPF project](#)
- [Coding Conventions](#)
- [Hosting update site](#)

Projects

- [jpf-core](#)
- [jpf-actor](#)
- [jpf-awt](#)
- [jpf-awt-shell](#)

- [jpf-concurrent](#)
- [jpf-cv](#)
- [jpf-delayed](#)
- [jpf-guided-test](#)
- [jpf-mango](#)
- [jpf-racefinder](#)
- [jpf-rtembed](#)
- [jpf-statechart](#)
- [net-iocache](#)
- [jpf-aprop](#)
- [jpf-numeric](#)
- [jpf-symbc](#)
- [jpf-concolic](#)
- [jpf-symbc-load?](#)
- [jpf-extended-test-gen](#)
- [jpf-parallel-spf?](#)
- [eclipse-jpf](#)
- [netbeans-jpf](#)
- [jpf-inspector](#)
- [jpf-shell](#)
- [jpf-template](#)
- [jpf-trace-server](#)
- [standard NB example](#)
- [Summer Projects](#)
- [External Projects](#)
- [Change\(B\)log](#)

[About](#)

- [About this Wiki](#)
- [About the Mailing Lists](#)
- [About the Development Process?](#)
- [About the Repository?](#)
- [How to Contribute](#)
- [JPF contributor account](#)
- [Events](#)
- [Presentations](#)
- [Papers](#)
- [FAQ](#)
- [History?](#)
- [Support](#)
- [People?](#)
- [Playground](#)
- [Table of Context](#)

jpf-statechart Translation Rules

For now, we only translate state charts. The two major requirements for the translated code are

- low model checker overhead (translated code should have closed state space, ideally only using the active state, CG sequence position and action-side effects for state matching)
- readable (code should not contain any constructs that are not in the diagrams) We especially want to avoid all auto-named constructs. Whatever is not named in the diagram (init states, end states, forks, joins, i.e. all pseudo-nodes) should be kept as an attribute (name, field etc.)

Translation Rules

The informal set of translation-rules are:

(R1) state chart system gets translated into one Java compile unit

(R2) every simple or composite state from the diagram gets translated into one class (that extends gov.nasa.jpf.sc.State)

(R3) sub-states get translated into nested classes. The state hierarchy from the diagram is mapped into a corresponding hierarchy of nested 'State' classes

Rationale:

(A) we need some way to specify the statechart hierarchy, and nested classes are most intuitive/least overhead to do this (also offers "free" visualization support from IDEs)

(B) the nesting allows for using lexical scoping, i.e. fields of super-states can be accessed without getters/setters from within their substates. Names of methods and fields of sub-states don't collide with their peer- or enclosing- states

(R4) sub-states are kept as concrete State fields inside their encapsulating state class. Fields get instantiated by default field initialization (i.e. by implicit default ctor)

Rationale: one-time initialization (at start time) enables efficient state matching and property implementation (cycles)

Field names can be chosen freely, but should correspond to their classes, since they are used in guidance scripts. The field types are the concrete State classes. State instances can also be kept in arrays of concrete State classes.

If there is a need to store states in fields that are just variables, i.e. are not intended to define the structure of the state machine, those fields have to be marked with a @NoSubState annotation.

Rationale: this is to enable setNextState(..) calls to reference non-peer states (e.g. "MyState.foo() {.. setNextState(myState.subState.anotherState)..}")

(R5) initial substate fields get assigned via a call to "makeInitial(state)", like in "MyState myState = makeInitial(new MyState());"

Rationale: while annotations would have been more appropriate, this was the last usage for them, and we rather don't introduce a new mechanism if we can avoid it.

(R6) triggers get translated into instance methods

Rationale: execution/simulation of state machines does not require dynamic creation of event objects. The execution policy is implemented outside the generated code (i.e. in 'StateMachine' implementations that resides in the libraries)

(R7) completion triggers are implemented in a method "public void completion()"

(R8) trigger guards become normal Java if-expressions inside their trigger methods

(R9) /entry, /exit, /do actions are translated into Java methods that are correspondingly named:

- /entry actions : public void entryAction()
- /exit actions : public void exitAction()
- /do actions : public void doAction()

Rationale: we have explicit methods wrapping all action calls do guarantee (a) action order and (b) parameter values specified in the diagram

(R10) transition target states are specified by calls to 'setNextState(state);' within their trigger methods, which have to be done before executing transition actions

Rationale: this is a bit counter-intuitive (it would have been more straight-forward to return the next state from trigger methods), but it is convenient to guarantee proper action execution (fired trigger -> source state exit actions -> transition actions -> target state entry actions). We can't do the exit before we know the target state because it might involve exiting parents, too. For readability, we don't want to turn transition actions into first class objects, but rather keep them as simple expressions inside their trigger methods

(R11) end states are mapped into a general State attribute ('isEndState'), which is set by calling "State.setEndState()" (instead of setNextState) from inside a trigger method leading to an end state

Rationale: if we would use an explicit EndState class and corresponding instances, we would have trouble with concurrent regions. This would force us to create multiple auto-named end states in the composite in order to determine if all regions have finished.

Semantics and Execution Scheme The main requirements are:

1. enable standalone execution (i.e. outside model checker)
2. enable model checker execution without extending the state space

Execution of state machines is controlled by 'StateMachine' instances, which encapsulate trigger selection policy, and hide associated implementation constructs from the model checker (esp. 'Event' objects).

The basic execution principle is that the environment maintains sets of trigger enabling events (either state specific or as a sequence of external events). In a standalone execution, the concrete Environment implementation determines which events enable triggers of the currently active state, picks one of them and calls the corresponding trigger method (via Java reflection). In a JPF model checking context, the set of enabling events is transformed into a ChoiceGenerator, i.e. all enabling events can be evaluated.

Execution proceeds in steps, with each step being processed in the same way

```
get next enabling event to process
copy 'nextActive' set into 'activeState' set
reset 'nextActive' set

terminate if 'activeState' set is empty
    or no transition occurred in previous step and there is no more event

for all states in the 'activeState' set
    reset 'nextState'
    if state has a triggerMethod corresponding to the event
        execute trigger method

    if 'nextState' is set
        add 'nextState' to 'nextActive' set
    else (no transition occurred)
        add state to 'nextActive' set
```