

Playwright Semantics

1. User

An *offline* user may or may not have an entry in a directory - either way, they are not assigned a password to access the system, and the Role instance is not operated by the system. This may be used for a Role carried out by another Role Interaction Machine, by a non-Playwright system, or not by a computer at all. An offline user may, however, have various *external addresses*, corresponding to different *messaging protocols*. These addresses are used when transferring online Entities via Interactions.

A *machine* user is not intended for interactive use by a person. All Role instances assigned to a machine user are entirely automated by RADRunner.

Search an LDAP directory by name or email to get further details of a user in their organisational context.

2. Role

2.1. Public Roles

A *public* Role type can in principle be instantiated by any user of the system. However, in order to provide centralised control, the only public Role types actually made available in this way are those known to the *root Role* of the system - the first instance of the 42 Role type currently belonging to the boss user.

Hence, public Role types created by other Role instances must first be exported as Playwright, then imported into the root Role, if they are to be made generally available for instantiation.

2.2. Activity Availability

RADRunner is able to use various different algorithms to decide which tasks in a Role are currently available for execution:

Name	Description	Steps	Advantages	Disadvantages
Declarative	Each condition is	1)	Determine	Easy to CPU-intensive

(default)	evaluated after every activity	which pre-conditions hold 2) Determine which activities are available 3) Depending whether the invoker is a user or RADRunner itself, determine which tasks can be executed	d Expressive power (e.g., looping) Can simulate sequencing	
Optimised Declarative	Only the "next" States are evaluated	1) Determine value of each pre-condition for activities following that just executed 2) Determine which of these activities are available 3) Depending whether the invoker is a user or RADRunner itself, determine which tasks can be executed	Caters naturally for sequencing Fast to execute	Difficult to understand Hard to implement looping
Procedural	Treat Role as flowchart	As in a workflow system	Easy to understand Fast to execute	Not a realistic approach to process modelling Hard to implement looping

3. Entity

See [Role resources](#).

4. Condition

A condition is a logical statement defined within a Role, used to:

Trigger an activity

A *precondition* on an activity, if true, enables it for execution

Validate the outcome of an activity

A *postcondition* on an activity, if false, forces it to be cancelled with no changes saved

Validate the outcome of every activity

An *always* condition on a Role forces any activity which violates it to be cancelled

Terminate a Role

A *terminating condition* on a Role, if true, sets the *ready for termination* flag on a Role - the actual termination is effected by the Role's parent, via the task [Stop Terminated Role And Interaction Instances](#)

A particular condition can be used for one or more of these purposes within a Role.

There are 2 types of conditions: [Simple Conditions](#) and [Compound Conditions](#). Either type can be used for any of the above purposes.

4.1. Simple Condition

A *simple condition* is an assertion about entity attribute values within the Role and/or constant values, using the expression operators:

- Equal To
- Less Than
- Less Than Or Equals
- Greater Than
- Greater Than Or Equals
- Starts With
- Contains
- Ends With
- Matches Regular Expression
- In
- Isa
- Is Null
- Not Null
- And
- Or
- Xor
- Nand

4.2. Compound Condition

A *compound condition* is a combination of simple conditions, using the logical operators:

- Not
- And
- Or
- Xor
- Nand

The use of compound conditions makes it possible to define conditions of arbitrary complexity.

5. Activity

An activity executes under transactional control. If any task within it fails, the entire activity will be cancelled, and no changes saved inside the Role.

This feature makes it possible to implement true *distributed heterogeneous transaction management*, for example during B2B web services orchestration. Moreover, there is no necessity to hand-craft a complex error handling framework for each case, as required by other business process management protocols. All that is required is simple *two-phase commit*.

6. Interaction

A Playwright interaction takes messages from one or more sources, and delivers each message to a number of recipients. The recipient list is different for each message:

- Input into an interaction is done via a *give* - this extracts a named entity from a Role instance, which can be nested at any depth within the Role resources.
- Output from an interaction is done via a *get* - a get puts the message from a give into a Role instance, placing it in a destination entity which can be nested at any level within the Role resources.
- An interaction has one or more gives, each of which has one or more gets. Each give and get contains the name of the resource it takes from, or sends to, a Role instance.

6.1. Binding

A give or get transfer is instigated by the execution of a task within the sending or receiving Role instance, of type [Part Interaction Give](#) or [Part Interaction Get](#). To enable a transfer, such a Role instance task must be *bound* to an appropriate give or get in an interaction instance.

Interaction binding is also known as *introduction*.

Playwright Semantics

Interaction binding can be done either *automatically* via the process enactment tasks:

- [Start A Role Instance And Introduce It To This Role Instance](#)
- [Introduce This Role Instance To Another Role Instance](#)

or *manually* via the process definition tasks:

- [Introduce A Giver To An Interaction Instance](#)
- [Introduce A Getter To An Interaction Instance.](#)

Either way, the resource name specified in the give or get must match the message name specified in the corresponding task. This is validated automatically by RADRunner, and the binding fails if the 2 sides do not match.

To use the manual approach above, it is generally necessary first to create the interaction via the process definition task [Start Interaction Instance](#).

6.2. Extension

Interactions can be extended via the addition of gives and gets either in the *type* via the tasks:

- [Add Give To Interaction Type](#)
- [Add Get To Interaction Type](#)

or in a particular *instance* via the tasks:

- [Add Give To Interaction Instance](#)
- [Add Get To Interaction Instance.](#)

6.3. Options

A give may have *Retain original of message in giver?* set to true. If so, the extracted entity will be left in the sender; otherwise it will be deleted from the sender.

A get may have *Retain original of message in getter?* set to true. If so, and a resource of the same name already exists in the receiver, the resource will be added as a new item in a numbered collection under the resource's owner, so as to avoid overwriting the existing resource in the receiver. The name of the new item is generated using a format defined via system property *com.rolemodellers.generatedNameFormat*, for which the default value is *{0}{1,number,#}* (for example, *MyEntity 12*).

An interaction may have *Terminate after single execution?* set to true. If so, it will terminate once all the gives and gets have completed. Otherwise it will be enabled to repeat once all the gives and gets have completed. The setting is made in the interaction type, so that it can be picked up by the automatable tasks [Start A Role Instance And Introduce It To This Role Instance](#) and [Introduce This Role Instance To Another Role Instance](#). However, if an

interaction type is instantiated manually via the task [Start Interaction Instance](#), the setting can be over-ridden for the interaction instance.