

---

# Little-JIL 1.0

## Language Report

A. Wise

16-Apr-98

### Introduction

Little-JIL is an *agent coordination* language. Programs in Little-JIL describe the coordination and communication among agents that enables them to perform a process.

### Agents and steps

A Little-JIL *agent* is an autonomous entity that is an expert in some part of the process described by a Little-JIL program. An agent may be human (e.g., a programmer in a software development process or a ticket agent in a trip planning process) or automated (e.g., a recompilation tool or a flight reservation system), in either case, an agent may be assigned work and is required to report back the success or failure of the work when the work is done.

A *step* is a specification of a unit of work that is assigned to an agent. Each step may contain a specification for the information and resources that are required (e.g., a detailed design for a programming task), pre-requisites that must be satisfied before an agent can begin the work (e.g., that all members be present for a meeting to start), the decomposition of the work into smaller steps (if appropriate), and post-requisites to check that the work was completed correctly (e.g., that a ticket is issued when a plane reservation is made).

Each step is assigned to an agent by posting it onto an *agenda* for the agent. Each agent has one or more agendas that the agent can examine to determine the work assigned to it.

### Relationship to JIL

Little-JIL concepts are drawn from JIL, a rich, full-featured, process language also under development at LASER. Little-JIL was created by selecting a minimal subset of the semantic features we felt makes JIL unique, adding a graphical syntax, and refining the language for agent coordination.

Making Little-JIL a subset of JIL makes it simpler to learn while allowing process programmers to easily migrate their Little-JIL programs to JIL when they need the additional flexibility and expressiveness that the larger language provides.

The major semantic features selected from JIL include:

- The blending of proactive and reactive control mechanisms through the sub-step mechanism and pre- and post-requisites,
- Using resources as a means of constraining and managing process execution, and
- The use of JIL steps as programming language scopes.

Experimentation led us to extend this initial language to include additional features, but only after careful thought. There are many features from JIL that *would have been nice*, but only those that were shown *indispensable* by our experimentation were made part of Little-JIL.

This approach to design omitted some of the common characteristics of conventional programming languages. The most obvious omissions include:

- Typical imperative programming statements, and
- Type declaration mechanisms.

## Visual notation

Little-JIL is a visual language and programs are written in Little-JIL by drawing a graphical depiction of the program. This report documents the semantics and the graphical representation of Little-JIL. Some information in a Little-JIL program may not be directly represented in the graphical depiction. Such information is said to be “attached” to the depiction and is connected to the depiction in a Little-JIL editor via hyper-linking, or, in a static representation, via call-outs.

## Acknowledgments

The definition of Little-JIL would not have been possible without the discussion and insight of all the members of the LASER Process Working Group: Y. Dong, B. Lerner, E. McCall, L. Osterweil, R. Podorozhny and S. Sutton.

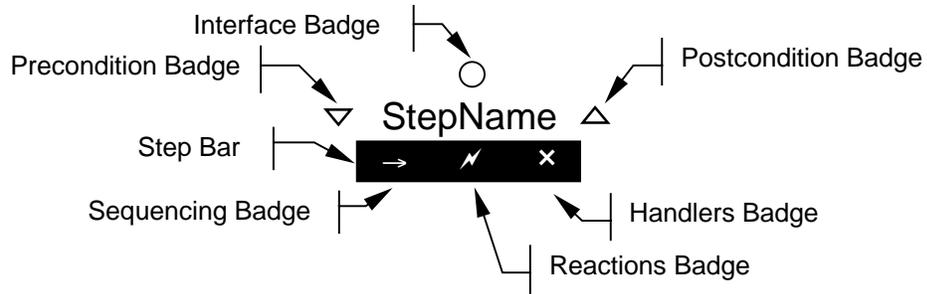
## Steps

A step is the basic building block of Little-JIL programs. A step represents a unit of work in the process and may be decomposed into sub-steps.

Every Little-JIL program has a *root step* that represents the entire process. This step is decomposed as far as necessary to describe the process.

## Declaring steps

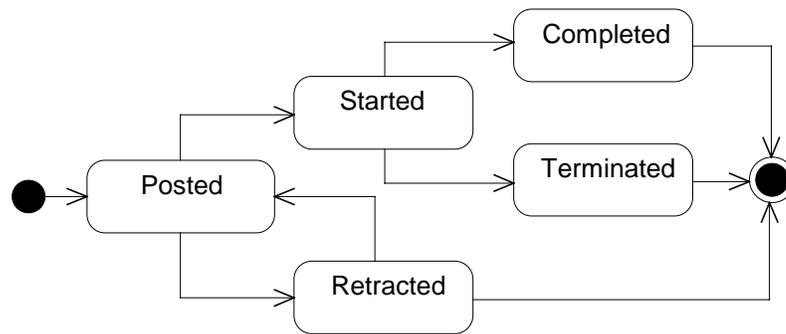
Each step in a Little-JIL program is represented by a step icon.



The Little-JIL step icon includes a step name that uniquely identifies the step, and may be annotated with *badges*, which are vehicles for providing additional information about the step or indicate the control flow within the step.

### Step execution

Conceptually, after instantiation a Little-JIL step is in one of five states: posted, retracted, started, completed, or terminated. Normally, a step moves through the states: posted, started, and completed.



- **Posted:** When the proactive and reactive elements of the process indicate that a step is eligible to be started, the resource manager is queried to determine that the resources and agent specified by the step exist and the step is posted to the agenda of the assigned agent to indicate that there is work to be done.
- **Started:** A step is started when the agent indicates that it wishes to begin the work specified by the step. When a step is started, the resources specified by the step are acquired and the pre-requisite is checked. If the resources are acquired and the pre-requisite is successfully executed, then the work is allowed to begin.

For a step with sub-steps, started means the sub-steps are posted. The order in which the sub-steps are posted is specified by the sequencing badge of the step.

For a step without sub-steps (a leaf), a started step is one that is being performed by an agent.

- **Completed:** When the work specified by a step is finished, the post-requisite is checked and the step's parent (if any) is informed that the step is done.

A step with sub-steps is completed when all of its sub-steps are retracted, completed, or terminated, and its post-requisite has been successfully executed.

A step with no sub-steps is completed when the agent informs the agenda manager that the work is done and its post-requisite is completed.

- **Terminated:** A terminated step is one that failed to complete its specified work. Step termination may occur as a result of an exception within the step, thrown by its requisites, or propagated up from a sub-step. A step that cannot acquire its resources is always terminated.
- **Retracted:** A retracted step is one that is removed from an agenda after having been posted but without being started by an agent. Steps are usually retracted as a consequence of their being unchosen alternatives in a choice step, but steps may also be retracted (and potentially reposted) as a result of exceptions.

## Sequencing

The execution of a step and its sub-steps is controlled by the step's sequencing. Steps may have one of five types of sequencing associated with them: none, sequential, parallel, choice or try. The sequencing for a step is represented by the *sequencing badge* on the left side of the step bar.

### None



If the sequencing badge for a step is empty, the step cannot have sub-steps, and is performed entirely by the agent assigned the step.

---

Example activity with no sequencing: call the pizza place.

---

### Sequential



If the sequencing badge is an arrow, the step sequencing is *sequential*.

A sequential step posts each of its sub-steps in order from left to right, posting the next sub-step when the previous one completes. A sequential step is complete when all of its sub-steps have completed.

---

Example sequential activity: go to the bank and then to the market.

---

## Parallel



If the sequencing badge is two horizontal lines, the step sequencing is *parallel*.

A parallel step posts all of its sub-steps concurrently, and is complete when all of its sub-steps have been completed. It is important to note that a parallel step indicates that the sub-steps *could* be done in parallel, not that they must.

---

Example parallel activity: get the milk and the eggs needed to bake a cake.

---

## Choice



If the sequencing badge is a small circle through a horizontal line, the step sequencing is *choice*.

A choice step allows agents to select one of several sub-steps to perform. When one step is selected to be performed, the other sub-steps are retracted. If the sub-step succeeds, the choice is complete. Handlers (see below) can allow the agent to make another selection if the sub-step fails.

---

Example of a choice: either take the bus or drive the car.

---

## Try



If the sequencing badge is an arrow crossed with an 'X', the step sequencing is *try*.

A try step allows agents to try alternative sub-steps left to right until one of them succeeds. When a sub-step succeeds, the try is complete. Handlers are used to specify when to try the next alternative.

---

Example of try: get brown eggs or white ones if brown are not available.

---

## References

*StepName*

Each step in a Little-JIL program is defined exactly once; however, it may be used multiple times (e.g., for recursion). These uses are represented by references. A reference is represented with italicized text and without badges.

Editors are encouraged to provide for navigation between a reference and the referenced step's definition.

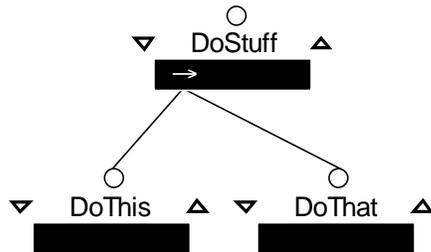
## Sub-steps

Sub-steps of a step are drawn below the parent step and are connected by arcs between the top of the sub-step and the sequencing badge of the parent step.

Editors are encouraged to provide a mechanism to hide the sub-steps of a step, replacing the sequencing badge with an ellipsis (...) to indicate that there is hidden information.

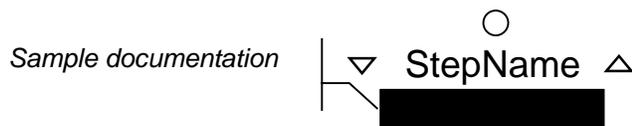
---

Example of sub-step depiction:



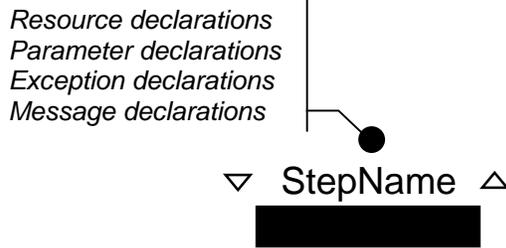
## Documentation and Annotations

Steps may have arbitrary annotations attached to the step bar. If attached via call-out, it should be typographically distinct from the surrounding information (e.g., colored or italicized).



## Step Interface

The interface to a step specifies the resources used by the step, the parameters of the step, and the exceptions and messages that may be propagated from the step. The declarations included in a step interface are documented below.



The interface to a step is attached to the interface badge of a step. If a step has an interface specified the badge should be colored.

## Resources

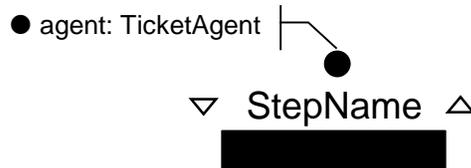
The execution of real-world processes is heavily influenced by the availability of people and materials. For example, an automobile assembly line cannot build cars without parts, or operate without skilled labor; similarly, a dealer cannot sell more cars than the assembly line can produce. Little-JIL process specifications can address these influences through the specification of the resources that manage and constrain the process.

### What is a resource?

Intuitively, a resource is any entity that is required to execute a step. The product of one step may be a resource to another. In our example above, the car is a product of the assembly line, but a resource to the dealer. The set of resources modeled in a process is specified in a resource model. For more information on defining resources, see the resource manager documentation.

There is one type of resource that is special to a Little-JIL step, namely its agent. Each step must have an agent. An agent may request an agent as a resource for the step, or have it passed down from the parent step. If a step requests an agent as a resource, the resource manager binds a specific agent to a step as part of step instantiation.

### Declaring resources



Resource declarations appear in the top section of a step's interface and are distinguished with a dot (●). A specification is made up of a name for the resource, and a resource description that is written in the language specified by the resource manager. For information on specifying resources, see the resource manager documentation.

The agent for a step must have the name *agent*.

## Using resources

Little-JIL resources are identified when a step is posted to an agent to ensure that there exist resources that match the step's descriptions. They are acquired (locked for exclusive use) when the agent begins the step and may be passed from one step to another or shared between steps via parameter bindings. Resources are released (unlocked) when all steps using the resource complete or terminate. All resource management operations for a step are performed in parallel. If resources need to be acquired or otherwise managed in a specific order, the Little-JIL step structure can be used to program the order of resource management.

Resources are passed to agents via the agenda manager and appear as normal parameters to the agent.

### Resource exceptions

After an attempt is made to identify or acquire the resources, if one or more cannot be identified or acquired, a subtype of **ResourceException** is thrown to the parent of the identifying or acquiring step for each failure:

- **ResourceUnknown** is thrown if no matching resource can be identified.
- **ResourceUnavailable** is thrown if no matching resource can be acquired.

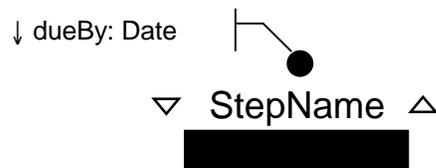
Resource exceptions have an attribute *name* that contains the name of the resource that could not be identified or acquired.

## Parameters

Parameter passing is a mechanism that can be used to transmit objects between parents and children in the sub-step hierarchy of a Little-JIL program.

### Declaring parameters

Little-JIL steps may have parameters. Each parameter has a name, type, and mode. The parameters appear in the center section of the step's interface.



### Parameter names

The name of a parameter is used to identify the parameter. The name is used in parameter bindings.

## Parameter types

A parameter must have a type that corresponds to an agenda attribute type. The standard agenda attribute types and the mechanism for defining new types is described in the agenda manager documentation.

## Parameter modes

Parameters are shared via copy-in/copy-out and have one of four modes: *in*, *out*, *in/out*, or *local*:

- **In** parameters are represented by an arrow pointing down ( $\downarrow$ ).

The value for an in parameter is copied into a step when the step is started, and is discarded when the step completes.

- **Out** parameters are represented by an arrow pointing up ( $\uparrow$ ).

The value for an out parameter is initialized to null when the step is started and is copied out when the step completes.

- **In/out** parameters are represented by both arrows pointing both down and up ( $\downarrow\uparrow$ ).

The value for an *in/out* parameter is copied into a step when the step is started and is copied out when the step completes.

- **Locals** are represented by a diamond ( $\diamond$ ).

A *local* is a parameter that is created within a step to allow the passing of information between the its sub-steps. Except for initialization, locals are only visible to the sub-steps of the step in which they are declared. The value for a local may be initialized via a parameter binding when the step is started, and is discarded when the step completes.

## Passing of parameters

Parameters are passed between steps via parameter bindings. A binding associates a parameter in a step with a parameter in a sub-step of that step. The binding of a parameter is represented by the notation

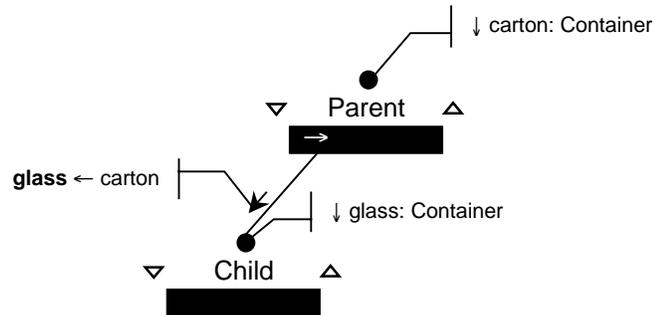
*sub-step-parameter* arrow *parent-step-parameter*

The arrow is matched to the direction the information flows (determined by the mode of the sub-step's parameter). The sub-step's parameter name is boldfaced to indicate that the name is defined in the interface and cannot be changed in the binding.

Parameter bindings are attached to a badge drawn on the arcs between steps. The badge should contain arrows drawn to match the union of the parameter modes for the sub-step (e.g., if a step has an *in/out* or an *in* and an *out* parameter, arrows pointing up and down should be drawn).

---

Example of parameter binding:  
Pass parameter *carton* of step *Parent* to step *Child* as parameter *glass*.



### Parameter compatibility

Two parameters are compatible if their types are compatible. The types of two parameters are compatible if the type of a parameter that is copied into is the same as (or a super-type of, if supported by the agenda type system) the parameter that is being copied. Note that this always requires both parameters to be the same type for an in/out parameter.

### Parameter exceptions

Parameters can create implicit control flow dependencies between the sub-steps of a parallel or choice step. If a step is started with an *in* (or *in/out*) parameter that is not yet had a value copied into it, the exception **ParameterUnavailable** is thrown in the newly started step.

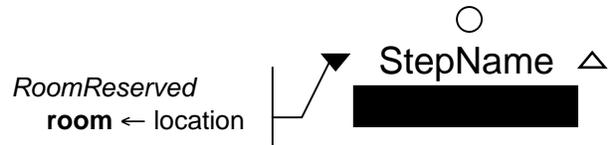
Parameter exceptions have an attribute *name* that specifies the name of the parameter that could not be accessed.

## Requisites

Requisites provide a mechanism to define guards on the entry to and exit from steps. If a requisite fails, the requisite throws an exception that is propagated to the parent of the step, which may have a handler specified to recover from the failure. Since requisites serve as guards, they are discouraged from having side effects.

### Requisites specification

A requisite is a reference to a Little-JIL step. Each step may have a single pre-requisite and a single post-requisite. If a step has multiple pre- or post-requisites, these must be grouped under a common step, and step decomposition is used to specify the order of evaluation. The reference representing the requisite is attached to the pre- or post-requisite badge depending on whether it should be checked before or after step execution. If a step has requisites associated with it, the appropriate badge(s) should be colored.



### Passing parameters to requisites

Requisites are steps and may have parameters. Since a requisite should not have side effects, a requisite may not have *out* or *in/out* parameters. The parameter binding for a requisite is attached to the requisite's name. In a static-representation, the bindings may be shown as indented under the requisite name.

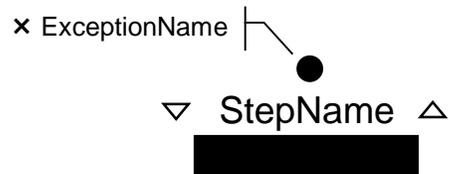
### Requisite execution

If a requisite is executed and throws an exception, it is considered to have failed and the step with which the requisite is associated is terminated.

## Exceptions

Exceptions in Little-JIL can be thrown by the interpreter to indicate that a resource or parameter is unavailable or by agents to indicate that they could not complete a step. Each step specifies the exceptions that can be thrown from the step.

### Exception specification

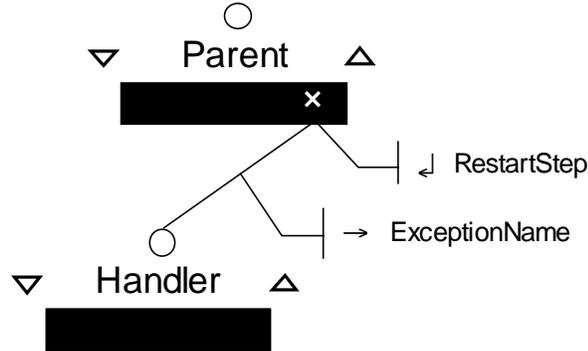


All of the exceptions that can be thrown by a step are specified in the step interface and are marked with a cross (×). Exceptions thrown by the interpreter (e.g., resource and parameter exceptions, see the appendix for a complete list) are implicitly thrown from a step and do not need to be declared in the step interface. Editors are encouraged to support the derivation and display of the exceptions that can be propagated up from a step's sub-steps.

## Handlers

Exceptions in Little-JIL are handled by handlers. If an exception thrown to a step and the step does not have a handler for it, the step is terminated and the exception is propagated to the parent step.

## Handler specification



Handler steps are drawn below the parent step and are connected by an arc to the handler badge. The specification of the handled exception is attached to the connecting arc. Other handlers are attached to the handler badge. As with sub-steps, editors are encouraged to support the hiding of handlers, displaying an ellipsis (...) when the handlers are hidden.

### Handler exception specification

A handler exception specification contains the type of the exception and optionally a set of attribute name, value pairs where the value may be either a constant or a parameter. The pairs are written

*name = value*

with the name boldfaced to indicate that it cannot be changed.

### Exception matching

An exception matches a specification if the exception is of the same type or a super-type of the type specified and the exception's attributes have the same values as all attributes included in the specification. In cases where an exception matches more than one specification the left-most matching specification is used.

## Handler actions

When an exception is thrown to a step, it is queued until the step has no started sub-steps. When an exception is queued, any posted sub-steps are retracted. When no posted or started sub-steps remain, an attempt is made to match the exception with the handler exception specifications for the step. If a match is found, the corresponding handler is executed. If a step does not have a handler for the exception, the step is terminated and the exception is thrown to the parent of the step.

### Multiple exceptions

Due to the potential concurrency in the execution of the sub-steps of a parallel step, a step may need to handle multiple exceptions. If a step's exception queue contains multiple exceptions, the queue is evaluated in the following order:

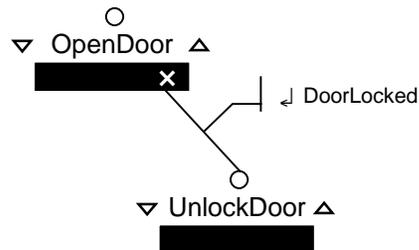
- 1) First, any handler steps for the exceptions in the queue are posted, and the interpreter waits until all of these steps have completed or terminated. If the handler steps themselves throw exceptions, the exception handling process is repeated with the exceptions thrown by the handler steps before continuing.
- 2) Next (or if there are no handler steps in the queue), if there are any exceptions without matching handler specifications or with rethrow control-flow badges, the step is terminated and the unhandled and rethrown exceptions are passed up to the parent. Any other exceptions are discarded.
- 3) Finally, if the step was not terminated in step 2, the control-flow badges (complete, restart, or continue) are processed:
  - a) If any handlers indicate that the step should complete, the step is completed (discarding all other exceptions).
  - b) If any handlers indicate that the step should restart but none indicate that the step should complete, the step is restarted (discarding all other exceptions)
  - c) Otherwise, execution of the step continues. If there are retracted sub-steps then they are re-posted.

### Handler steps

When an exception is handled, the handler may post a *handler step* for the exception to “clean-up” or otherwise react to the exception.

---

Handler example: if the door is locked, unlock it.




---

### Handler control-flow badges

Whether or not a handler posts a handler step, the handler specifies if the executing step should continue, complete, rethrow, or restart.

#### Continuing the step

The exception is discarded and execution continues at the point where the exception was received, re-posting any retracted sub-steps. Continuation is represented by an arrow (→).

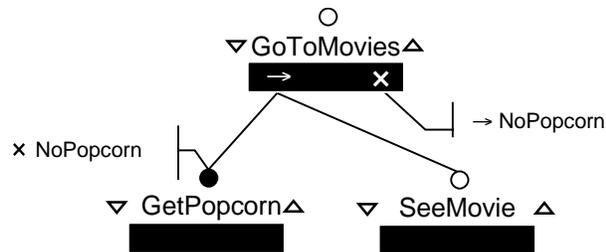
The exact meaning of continuation depends on the context in which it is used:

- Parallel and choice steps re-post all of the sub-steps retracted when exception processing began.
- Sequential and try steps post the next sub-step.

If all sub-steps are completed or terminated and an attempt is made to continue a step then:

- Sequential and parallel steps are completed.
- Choice and try steps are terminated and the exception **NoMoreAlternatives** is thrown.

Example of continue after an exception:  
If we can't get popcorn, we can still see the movie.

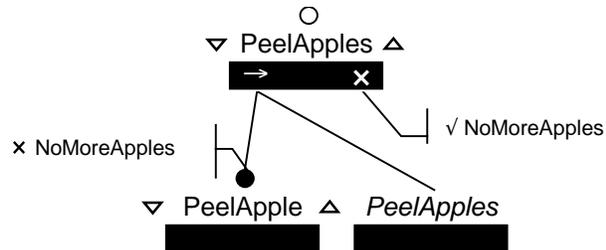


### Completing the step

The exception is discarded and the step is completed; control passes immediately to post-requisite evaluation. Represented by a check (✓).

Note that the execution of a step always requires the checking of the requisites, and this mechanism cannot be used to bypass them.

Example of complete after an exception:  
Peel apples until the bowl is empty. When we run out of apples, we are done.



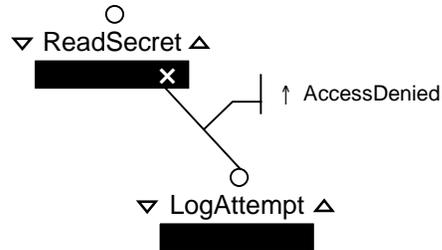
## Rethrowing the exception

Rethrowing an exception terminates the step as if had not handled the exception allowing a handler to respond to an exception without recovering. Represented by an arrow pointing up (↑).

---

Example of rethrowing an exception after a handler:

Read secret information, if access is denied, log the attempt and terminate.



---

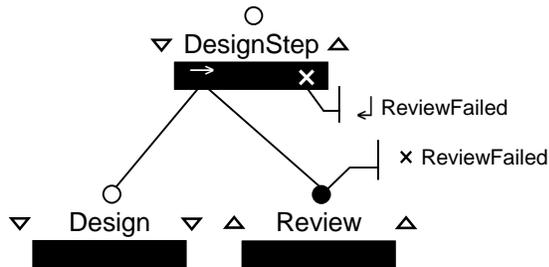
## Restarting the step

Restarting a step discards the exception and begins execution of the step again. When a step is restarted, parameters are re-bound, resources are re-acquired, and pre-requisites are re-checked. Represented by an angled back-arrow (↶).

---

Example of restarting a step after a handler:

Do a design and review, if the review fails, repeat.

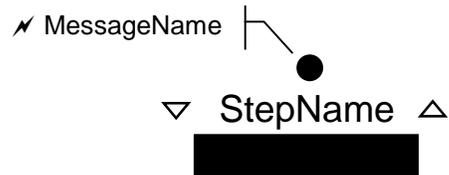


---

## Messages

Messages in Little-JIL can be sent by the interpreter to allow processes to react to their own execution or by agents to indicate events of interest. Each step specifies the messages that may be sent by that step. A step may only send messages when it is in the started state.

## Message specification



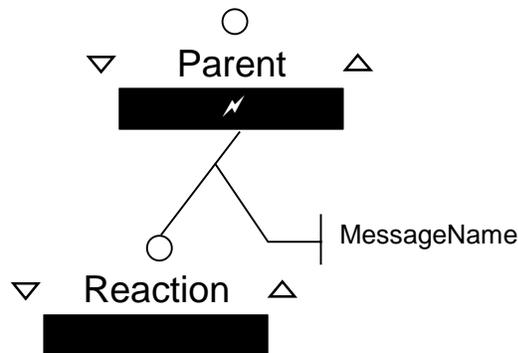
All of the exceptions that can be sent by a step are specified in the step interface and are marked with a lightning bolt (⚡). Messages sent by the interpreter (e.g., step started) are specified in the appendix and do not need to be specified in the step interface.

## Reactions

Reactions provide a mechanism for dynamically responding to the arrival of messages. Messages can be sent by agents, environment services (e.g., an object manager), or by the interpreter in response to events within the process.

## Reaction specification

Reactions are represented by steps drawn below the parent step and are connected by an arc to the reactions badge. The specification of the message is attached to the connecting arc. As with sub-steps, editors are encouraged to support the hiding of reactions, displaying an ellipsis (...) when the reactions are hidden.



Little-JIL reactions post sub-steps in response to messages. A step reacts to messages only while the step is started. Since the sub-steps are posted immediately upon receipt of a message, these steps can always be executed in parallel with the proactive sub-steps and each other.

## Message specification

As with exceptions, a message specification contains the type of the message and optionally a set of attribute name, value pairs where the value may be either a constant or a parameter. The pairs are written

*name = value*

with the name boldfaced to indicate that it is defined within the message cannot be changed within Little-JIL.

### **Message matching**

The matching rules for messages are the same as those for exceptions. Unlike exceptions, messages are sent to all started steps with matching message specifications. When a message can match more than one specification, all of the specifications are considered to match and all of the associated steps are posted.

### **Process messages**

During execution, the Little-JIL interpreter sends the messages in response to events in the execution of the program including the posting, starting, and stopping of steps. The complete list of messages is included in the appendix.

## Appendix: Little-JIL Types

### Exceptions

#### **ProcessException**

An abstract exception type for all exceptions thrown by the Little-JIL interpreter.

Attributes:

SourceStep: the step from which the exception was thrown.

#### **Resource Exceptions**

#### **ResourceException**

Subtype of ProcessException. An abstract exception type for all resource exceptions.

Attributes:

Name: the name of the resource that could not be identified or acquired.

#### **ResourceUnknown**

Subtype of ResourceException. An exception that indicates that no resource could be identified that matches the specification.

#### **ResourceUnavailable**

Subtype of ResourceException. An exception that indicates that no resource could be acquired that matches the specification.

#### **Parameter Exceptions**

#### **ParameterUnavailable**

Subtype of ProcessException. An exception that indicates that a parameter did not have a value when an attempt was made to copy it.

Attributes:

Name: the name of the unavailable parameter

### Messages

#### **ProcessEvent**

An abstract message type for all messages sent by the Little-JIL interpreter.

Attributes:

SourceStep: The step from which the event was sent.

#### **StepStateChangeEvent**

Subtype of ProcessEvent. An abstract message type for all Little-JIL messages representing step state transitions.

Attributes:

ParentStep: The parent step of the step that changed state.

#### **StepPostedEvent**

Subtype of StepStateChangeEvent. A message that indicates that a step has been posted on an agenda.

#### **StepRetractedEvent**

Subtype of StepStateChangeEvent. A message that indicates that a step has been retracted.

#### **StepStartedEvent**

Subtype of StepStateChangeEvent. A message that indicates that a step has been started.

**StepFinishedEvent**

Subtype of StepStateChangeEvent. An abstract message that indicates that a step has finished.

**StepCompletedEvent**

Subtype of StepFinishedEvent. A message that indicates that a step successfully completed.

**StepTerminatedEvent**

Subtype of StepFinishedEvent. A message that indicates that a step terminated with an exception.

Attributes:

Exception: The exception that terminated the step.