Little-JIL 1.5
Language Report

LASER Process Working Group

3-Oct-06

Introduction

Little-JIL is an *agent coordination* language. Programs in Little-JIL describe the order of, and the communications between, units of work called steps. By assigning steps to agents, a Little-JIL program assists the agents in the completion of 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). A step also specifies how it should respond to events that may occur during its execution, and errors that may be reported during the execution of the decomposition.

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 other languages

Because Little-JIL is designed as a coordination language, it omits some of the common characteristics of conventional programming languages. For example, Little-JIL lacks:

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

©1997—2006, Laboratory for Advanced Software Engineering Research, University of Massachusetts, Amherst.

Instead, these features are represented as external factors to be provided by the agent environment in which Little-JIL is used. For example, within LASER, we have developed a Little-JIL environment (which we call Juliette) that uses Java to provide these features.

This separation is not new. The separation of coordination from computation was central to the design of the Linda coordination language, however Little-JIL offers several features that we believe makes the language particularly suited to the domains of agent coordination and process programming. Most significantly:

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

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 “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 and evolution of Little-JIL would not have been possible without the discussion and insight of all the members of the LASER Process Working Group, including: A. Cass, Y. Dong, A. Elssamadisy, B. Hawkes, H. Lee, B. Lerner, E. McCall, D. Miller, A. Ninan, L. Osterweil, R. Podorozhny, M. Raunak, J. Sieh, S. Sutton, and A. Wise.

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032, by U.S. Department of Defense/Army and the Defense Advanced Research Projects Agency under Contract DAAH01-00-C-R231. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U. S. Army, the U.S. Dept. of Defense, or the U.S. Government.

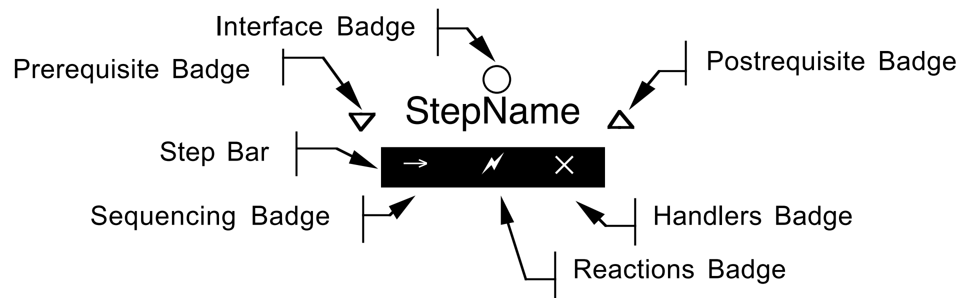
Steps

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

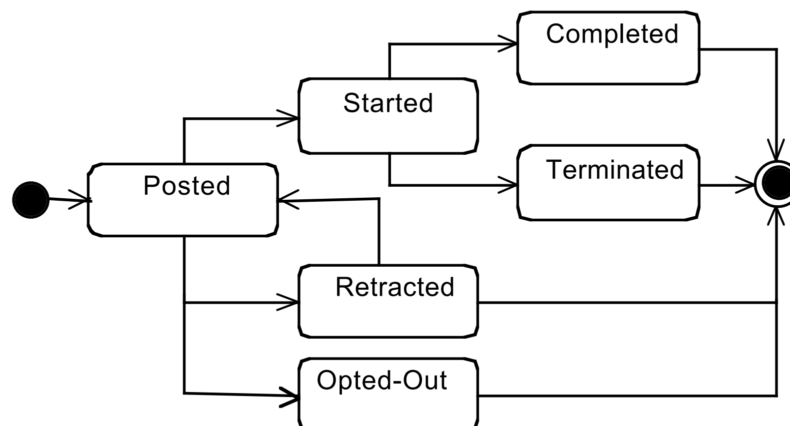
A step icon represents each step in a Little-JIL program.



The Little-JIL step icon includes a step name, and may be annotated with *badges*, which are vehicles for providing additional information about the step or to indicate the control flow within the step. Information on each of these elements is described below.

Step execution

At run-time, the steps in a Little-JIL process program are treated as templates, from which instances are created. After instantiation, conceptually, a Little-JIL step instance is in one of five states: posted, retracted, started, completed, or terminated. Optional steps have a sixth state: opted-out. Normally, a step moves through the states: posted, started, and completed.



- **Posted**: When the proactive or reactive elements of the process indicate that a step is eligible to be started, the resource manager is queried to determine that

the resources required by the step exist, the steps parameters are initialized, an agent is assigned, 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 appropriate sub-steps are posted. The sequencing badge of the step specifies the order in which the sub-steps are posted.

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

- **Completed**: A completed step is one whose work was finished successfully. When the work specified by a step is finished, the post-requisite is checked, resources are released, 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.

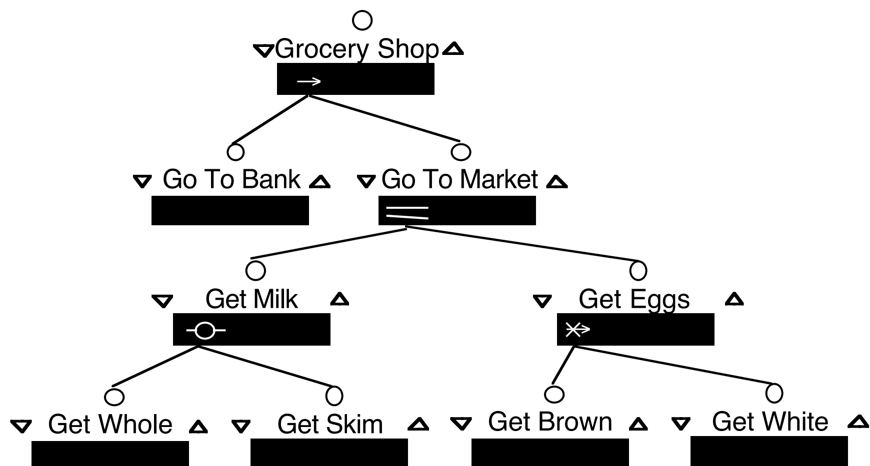
- **Terminated**: A terminated step is one that failed to complete its specified work. Step termination may occur as a result of an exception within the step, thrown by its requisites, or propagated up from a sub-step. A step that cannot acquire its resources is always terminated. As with completed, a terminated step releases its resources.

- **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. When a retracted step is reposted, it may be posted to a different agent than the previous instance.

- **Opted Out**: An opted-out step is an optional step that the agent has indicated that they will not start.

Sub-steps

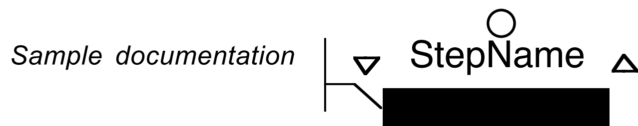
Sub-steps of a step are drawn below the parent step and are connected by arcs between the top of the sub-step and the sequencing badge of the parent step.



References

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

Documentation and annotations



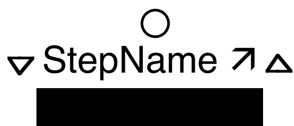
Steps may have arbitrary annotations attached to the step bar. If attached via call-out, it should be typographically distinct from the surrounding information (e.g., colored or italicized).

Modules

Modules provide a mechanism to package and reuse Little-JIL processes. Modules may contain a list of steps that are *exported* (i.e., defined in the module and are to be made available to other modules), and a list of steps that are imported, (i.e., used by the module, but are not defined within the module

Exported steps

Exported steps are identified by the inclusion of an export arrow after the name in the step declaration.



Imported steps

Imported steps are identified by the inclusion of an import arrow before the name in the step reference.

Sequencing

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

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: Go to the bank.

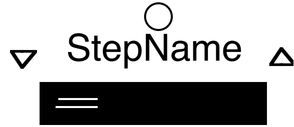
Sequential

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

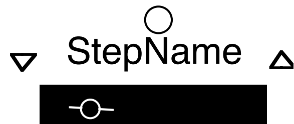


If the sequencing badge is two horizontal lines, the step sequencing is *parallel*.

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 milk and eggs.

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: get either skim or whole milk.

Try

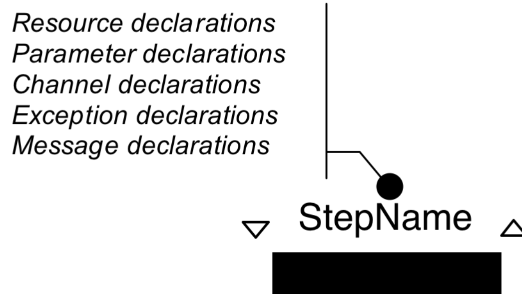
If the sequencing badge is an arrow crossed with an 'X', the step sequencing is *try*.

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.

Step Interface

The interface to a step specifies the resources used by the step, the parameters of the step, the channels the step declares, 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.

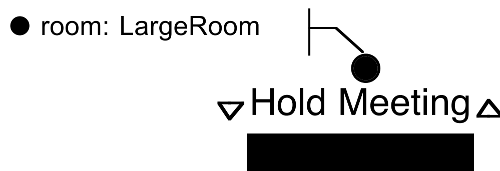
Resources

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 for which there is contention for access. 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.

Declaring resources



All resource declarations have a mode, and a name. There are five modes for resource declarations: acquisitions, uses, collections, iterators, and collection uses. Additionally, resource declarations that query the resource manager have an

associated resource specification. The resource specification language is external to Little-JIL.†

Resource acquisition

A resource acquisition declares a resource that must be acquired before the step can begin execution. A resource acquisition is identified with a filled dot (●).

Resource use

A resource use is identified with an open dot (❍). A resource use declares a resource that will be provided by another step.

Resource collection

A resource collection is identified by two filled dots (●●). A resource collection is a (potentially empty) subset of the resource model. Resource collection can be used as a vehicle for representing groups of resources within a program (E.g., to model programmer teams.)

Resource collection iterator

A resource collection iterator is identified by two dots and an arrow (→●●). A resource collection iterator is a resource collection that shares iteration state between all of its uses. For more information on iterating over resources, see "Cardinality" below.

Resource collection use

A resource collection use is represented by two open dots (○○). A resource collection use declares a resource collection that will be provided by another step.

Agent defined collection

Resource collection uses can also be used as a mechanism to allow agents to select resources by allowing the agent to specify the collection from which the resources will be acquired.

Agents as resources

There is one type of resource that is special to a Little-JIL step, namely its agent. An agent may be declared as a resource for the step, or if no agent is declared, the agent is inherited from the parent step. If a step specifies an agent, the resource manager binds an agent to the step as part of instantiation. Each step must have an agent, therefore every process program must at least declare an agent for the root step.

The agent for a step must be named 'agent'.

† Note: the simple type names (e.g., LargeRoom in the above figure) are not intended to represent any particular resource specification language.

Using resources

If a Little-JIL resource declaration is a resource acquisition, it identified when a step is posted to an agent to ensure that there exist resources that match the step's descriptions, and is acquired (locked for use) when the agent begins the step. Resources 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. The order for resource management operations for a step is undefined; if resources need to be acquired or otherwise managed in a specific order, the Little-JIL step structure can be used to program an ordering.

When acquired, the external resource manager provides a handle for accessing the resource. The agents access these handles in the same manner as parameters.

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 in the resource model.

- **ResourceUnavailable** is thrown if a matching resource exists, but cannot be acquired.

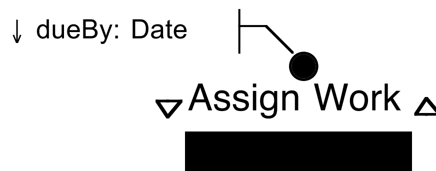
Resource exceptions have an attribute *name* that contains the name of the resource that could not be identified or acquired.

Parameters

Parameter passing is a mechanism that can be used to communicate information 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, mode, and an optional default value..



Parameter names

The name of a parameter is used to identify the parameter. The name is used in parameter bindings, and to communicate with the agent.

Parameter types

A parameter declaration specifies a parameter's type. The type however, is declared externally to Little-JIL.

Parameter modes

Parameters are communicated 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 (↓).

  The value for an in parameter is copied into a step when the step is posted.

- **Out** parameters are represented by an arrow pointing up (↑).

  The value for an out parameter is copied out if the step completes.

- **In/out** parameters are represented by an arrow pointing both down and up (↕).

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

- **Locals** are represented by a diamond (◊).

  A *local* is a parameter that is created within a step to allow communication between its sub-steps. Locals are only visible to the sub-steps of the step in which they are declared.

Default values

If a value is not copied into a parameter when it is posted, the parameter is initialized to the specified default value, as shown below, or if no default is specified, to the default value specified by the external type model.

◊ author: Name = "A. Nonymous"

Write Novel

Passing of parameters

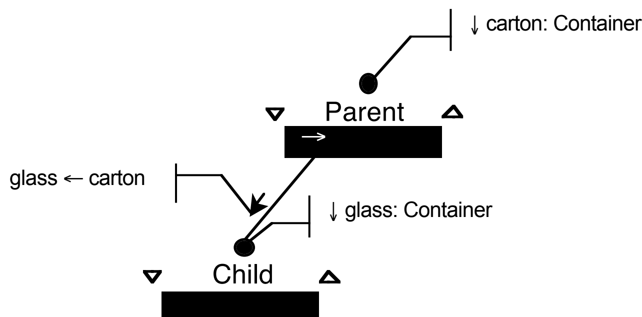
Parameters are passed between steps via parameter bindings. A binding associates a constant or a parameter in a step with a parameter in a sub-step of that step. The binding of a parameter is represented by the notation:

sub-step-parameter arrow *parent-step-parameter* or *constant*

The arrow is matched to the direction the information flows (determined by the mode of the sub-step's parameter. 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, and their modes permit their values to be copied in, or out, as appropriate. For an in/out parameter, the parameter is mode compatible with several different modes, and it is possible to have separate bindings for the in and out operations. The external type model defines type compatibility.

Binding to a constant

A parameter may be bound to a constant value if the parameter mode is in or in/out.

Structured parameters

While type declarations are external to Little-JIL, some information about the type may be used from within the language. For structured types such as collections and aggregates, bindings may access and update their contents.

Members of a collection

If a parameter is an instance of a collection type, the contents of the collection may be accessed or modified by appending '[]' to the parameter or field name:

parameter ← collection[]

Fields of an aggregate

If a parameter is an instance of an aggregate type, the fields of the aggregate may be accessed or modified by appending a the field name after a period to the parameter name or collection member:

parameter ← aggregate.field

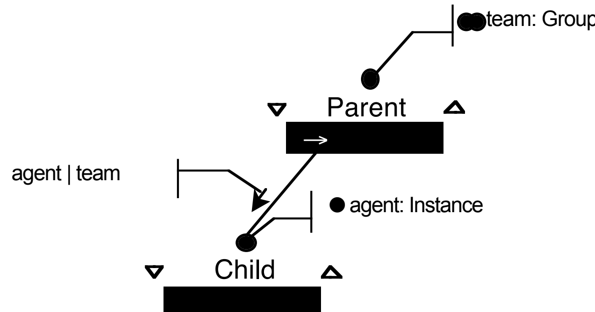
Resource bindings

Resources may also be passed between step using parameter bindings. For purposes of determining compatibility, resource acquisitions may be copied into resource uses and resource uses may be copied into other resource uses. Similarly, resource collections and resource collection iterators may be copied into resource collection uses, and resource collection uses may be copied into other resource collection uses. While all resource declarations may be copied, only resource uses may be assigned.

In addition to being passed around like parameters, resource collections, resource collection iterators, and resource collection uses may be used as constraints on other resource declarations. A constraint restricts the selected resource or resources to be from the constraining collection rather than the entire universe of resources.

Constraints are specified as part of parameter bindings, and are represented by a vertical bar drawn between the resource declaration in the child and the collection in the parent.

Example: The agent for step *Child* is selected from the collection *team*:

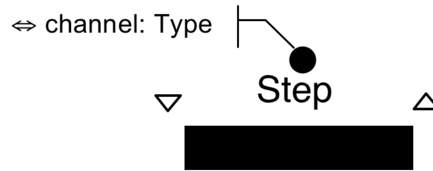


Channels

Channels provide a communication mechanism that is not tied to the hierarchical structure of Little-JIL. The semantics of channels support data-centric synchronization based on those in Linda. While channels can be used in many situations, they are essential for supporting communication between potentially parallel threads of execution.

Declaring channels

Little-JIL steps may declare channels. Each channel has a name and a type. Channels are identified iconically by a horizontal double arrow.



Channel names

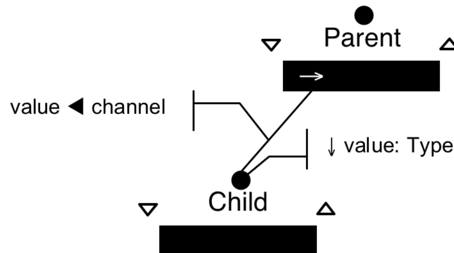
Each channel has a declared name that is used when addressing the channel. The visibility of a channel is limited to the sub-steps of the step in which it is declared.

Channel types

Channels are typed and contain a homogeneous collection of artifacts. If the artifact model supports inheritance it may be used to create heterogeneous channels. Channels are ordered: artifacts are retrieved from the channel "first in, first out" (FIFO). However, order may not be preserved if there are concurrent reads/takes. Channel types may support either insert or replace semantics.

Accessing channels

As with parameter flow, channel operations represented as bindings associated with connectors between steps and are coupled to the life cycle of a step instance.



Write

"Write" copies the object of an out or in/out parameter into a channel when a step instance completes. As with copy-out, if the step instance terminates, the write does not take place. Write is represented with an open triangle with the point to the right (▷).

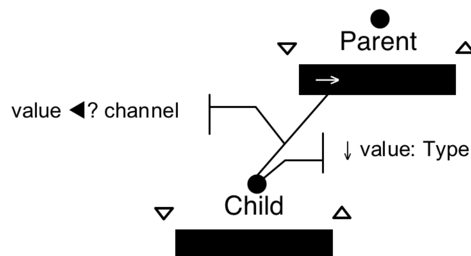
Read

"Read" copies an artifact from a channel into an in or in/out parameter of a step instance. As with copy-in, the read takes place in preparation for posting the step. If the channel is empty, the instance is not posted until an artifact becomes available. Read is represented with an open triangle with the point to the left (◁).

Take

"Take" is similar to "read" except that the artifact is removed from the channel and ensures that only a single step instance receives the artifact. If the step instance is retracted or terminated without reaching the started state (e.g., in the event of a pre-requisite failure), the taken artifact is returned to the channel. In to *attempt* to preserve the channel order, the artifact is returned at the head if the queue. However, if another artifact is read or taken from the channel between when the first take is performed and when the first artifact is returned (as can be the case for multiple concurrent accesses), the order will not be preserved. Take is represented with a closed triangle with the point to the left (◀).

Non-blocking reads and takes



Reads and takes may be set to be non-blocking, which results in the parameter being bound to the types default (the same value that a local will be set to if there is no initializer) if no data appears on the channel when the step is posted.

Not blocking reads and takes are identified by a question mark after the read or take to identify the action as optional.

Connectors with multiple channel accesses

If there is more than one read and or take on a connector the order of the reads and or takes is undefined.

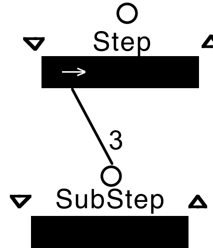
Interaction with parallel and choice sequencing

The semantics for Little-JIL parallel and choice specify that the sub-steps are posted atomically. This will result in the delay in posting of *all* of the sub-steps in the event that any of them are waiting on a channel. If asynchronous posting is desired, the creation of additional scopes may be used.

Cardinality

Cardinality is a mechanism for expressing the optionality or repetition of a step. The number of instances of a step that should be created may be specified statically within a process, be determined by the agent assigned to the steps, or process programmers may indicate that step instances should be created based on the availability of artifacts or resources.

Static and agent bounds



The cardinality of a connector is displayed adjacent to where the connector is attached to the sub-step as shown above. Refer to the table below for the notation used to indicate the cardinality of the sub-step.

When cardinality is applied to an edge, the sequencing badge on the parent determines the execution semantics of the associated sub-step.

?	This step is optional (zero or one times.)
+	Do this step at least one time.
*	Do this step zero or more times.
Number	Do this step exactly number times.
Lower .. Upper	Do this step at least lower times, and at most upper times. Both lower and upper are numbers.
Number+	Do this step number or more times.

Agent control

If the cardinality is not a single static bound (e.g., 3), then during at least some part of the execution, the repeated execution of the step must depend on some external decision maker. If no other decision maker is specified (e.g., a resource bound), the agent assigned to the step controls the execution through the ability to *opt out* of a step instead of starting it. Agents can only opt out of step if the cardinality permits it, so if a step has a cardinality of a range of 3..5, the agent is only permitted to opt out on iterations 4 and 5. If all of the sub-steps of a choice or try step have been opted-out, the exception **NoMoreAlternatives** is thrown.

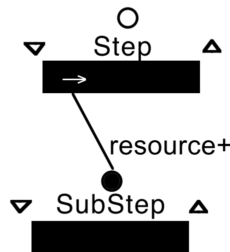
For a step with a sequential or try sequencer, after a step instance completes (or terminates for try), the next step instance is presented until the upper bound is reached, or the agent decides to opt out instead of starting the instance. For parallel or choice sequencers, enough instances are posted to satisfy the lower bound, and then another instance of the step is posted immediately upon the starting of a step instance until enough instances have been posted to satisfy the upper bound. This mechanism rolls out the optional instances sequentially (when one instance is started, another is posted.)

Interaction with continue

The execution semantics of cardinality interact with the exception handling mechanism of Little-JIL when an exception that is thrown during the execution of a step with a cardinality edge is handled with a continue.

Continue will return execution into the iteration unless the iteration has already satisfied its lower bound. For example, if an exception is thrown on the 2nd iteration of a step with a cardinality of 3..5, the third iteration will be posted, but if the 3rd iteration terminates, continue will resume execution *after* the iteration. Note that this means it is impossible to resume an iteration marked with kleene-star (*).

Resource or artifact bounds



The number of instances of a step can be controlled by the contents of a collection or the availability of resources in the resource model. Such bounds are represented by including the name of one of the parameters or resources declared in the step as part of the cardinality. Resource or artifact bounds may be combined with static bounds to express constraints on the minimum and maximum number of instances required. The constraint is written following the resource or parameter name in parenthesis (e.g., parameter (3..5)) except for kleene-star, kleene-plus, and optional, where the parenthesis are omitted (e.g., resource+). The default cardinality of a resource or artifact bound is kleene-plus which means use all available resources or artifacts in the collection and require that there be at least one.

As above, the sequencing badge on the parent step determines whether the steps are posted in parallel or sequentially.

For a sequential or try step, each instance is assigned a unique resource or artifact. The iteration is finished when the contents of the collection or set of available resources is exhausted. For resource bounds, each instance requires an interaction with the resource manager to select a new resource in order to avoid the possibility that resources may be removed from the resource model during execution. A side effect of this interaction is that resources added after the iteration begins may be included in the iteration.

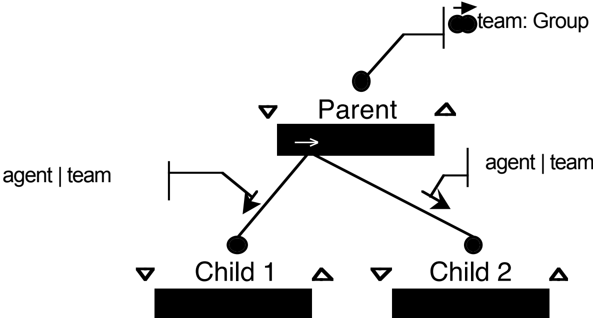
The behaviors of parallel and choice steps are similar to those of sequential and try, except that all iterations are posted when the parent step is started. Note that for resource bounds, unlike in the sequential case, if additional resources that could be part of the iteration are added to the resource model after the parent step is started, they *are not* added to the iteration to avoid the potential for a race condition between completing a step and adding resources.

If a parameter bound to a channel is used to iterate over the contents of a channel, as with resource-bounded iteration, for parallel and choice steps, races are avoided by bounding the iteration to the items in the channel at post time.

Resource collection iterators

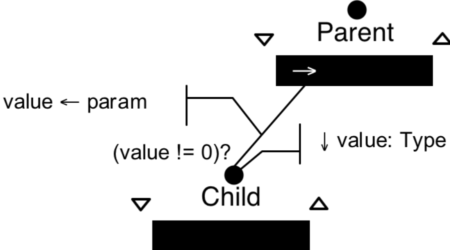
Resource collection iterators may be combined with resource bounds to share iteration state between multiple steps. Resource bounds constrained by a resource collection iterator have the characteristic that when a resource instance is acquired, the instance is not available to be selected by future acquisitions constrained by the same iterator.

Example: The agents for the steps *Child 1* and *Child 2* are both selected from the collection *team*, but cannot be the same agent:



Predicates

Predicates provide a conditional mechanism to control the posting of sub-steps. A predicate appears as a parenthesized expression in the cardinality and are mutually exclusive with the agent and resource or artifact-controlled mechanisms.



Predicate language

Since predicates are dependent on the particular model of the artifact definition language the predicate language is external to Little-JIL.

Predicate evaluation

Predicates are evaluated after all of the input parameters are bound. If the predicate evaluates to true, the sub-step is posted, if false, the sub-step is not posted, and

execution of the parent step continues as if the sub-step did not appear. For example, if the parent is a sequential step, the next step in the sequence is evaluated.

Namespace

Predicates are evaluated within the context of the step to be posted. All parameters are bound before the expression is evaluated except for resources, which are not available since they will not be acquired until the step is started.

Requisites

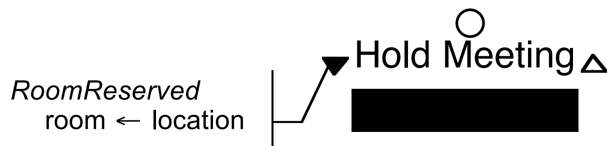
Requisites provide a mechanism to define guards on the entry to and exit from steps. If a requisite fails, the requisite throws an exception that is propagated to the parent of the step, which may have a handler specified to recover from the failure.

Requisites specification

A requisite is a step that is referenced from the pre- or post-requisite badge of another 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.

Steps as requisites

If a requisite is a step, 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

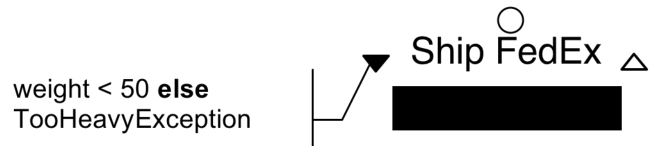
Parameters may be passed to a requisite step. Since a requisite should not have side effects, a requisite may not have *out* or *in/out* parameters. The parameter binding for a requisite is attached to the requisite's name. In a static-representation, the bindings may be shown as indented under the requisite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

Predicates as requisites

Predicates may be used as requisites and are represented by attaching the expression and associated exception type separated by the keyword 'else' to the appropriate requisite badge.



Namespace

As with predicates, the expression of a simple requisite is evaluated within the context of the associated step.

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

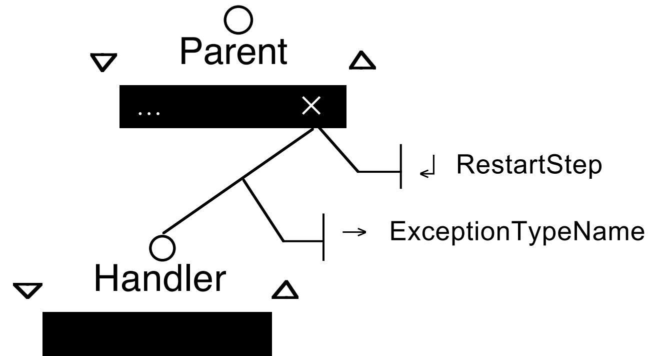
All of the exceptions that can be thrown by a step are specified in the step interface and are marked with a cross (×). Exceptions thrown by the interpreter (e.g., resource and parameter exceptions, see the appendix for a complete list) are implicitly thrown from a step and do not need to be declared in the step interface.

Exceptions are defined with a type external to Visual-JIL, just as parameters are defined.

Handlers

Handlers handle exceptions in Little-JIL. 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



Handlers are drawn below the parent step and are connected by an arc to the handler badge. The continuation and specification of the handled exception is attached to the connecting arc.

Handler exception specification

A handler exception specification contains the type of the exception and optionally a set of attribute name, value pairs. The pairs are written

name = *value*

Exception matching

An exception matches a specification if the exception is of the same as 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 posted or 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.

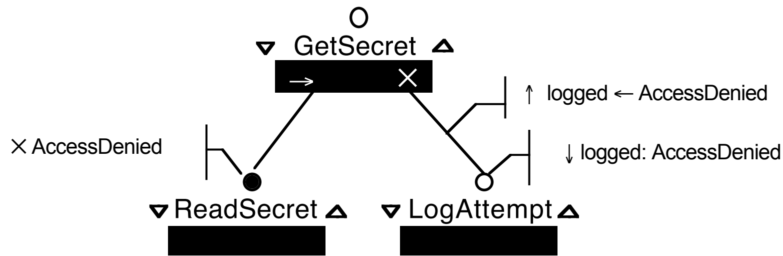
Handler steps

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

Passing exceptions as parameters

Exception objects may be passed into any type compatible *in* or *in/out*, parameter in the handler step. The parameter name and an arrow to the left of the exception declaration represent the binding.

Example: If reading the secret fails, log the attempt, passing the logging step the exception as 'logged.'



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 of the step with the exception handler continues, re-posting any retracted sub-steps. Continuation is represented by an arrow (→).

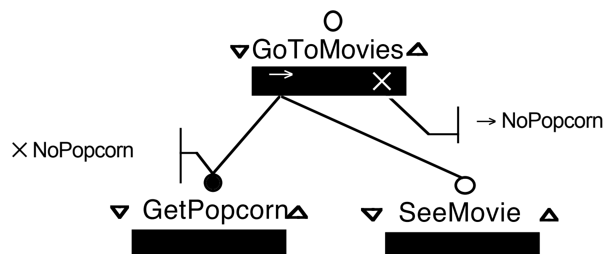
The exact meaning of continuation depends on the context in which it is used:

- 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.

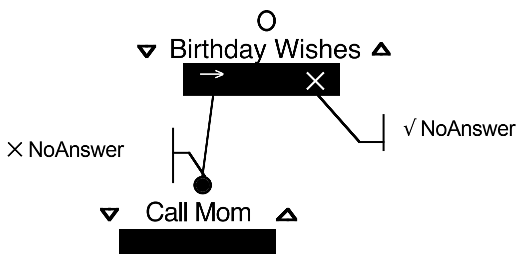
Example of continue after an exception:
If we can't get popcorn, we can still see the movie.



Completing the step

The exception is discarded and control passes immediately to post-requisite evaluation. As a result, postrequisites are evaluated after the complete control-flow is used. Represented by a check (√).

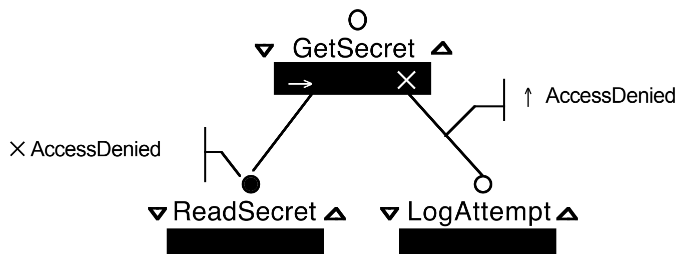
Example of complete after an exception:
Try to call mom on her birthday. If she is not home, that is OK.



Rethrowing the exception

Rethrowing an exception terminates the step as if it 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.

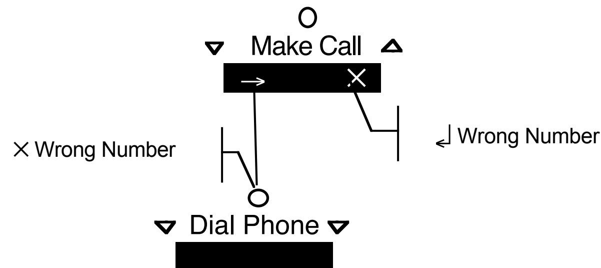


Restarting the step

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

The use of restart is discouraged as a means of implementing rework since it discards valuable context information.

Example of restarting a step after a handler:
Make a phone call, if it is the wrong number, re-dial.



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, those exceptions are queued to be re-thrown to the step's parent.

2) Next (or if there are no handler steps in the queue), if there are any exceptions without matching handler specifications, if any of the handlers threw exceptions, or any of the executed handlers have 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 (without processing the other control-flow badges).

- **b)** If any handlers indicate that the step should restart but none indicate that the step should complete, the step is restarted (without processing the other control-flow badges)

- **c)** Otherwise, execution of the step continues. If there are retracted sub-steps then they are re-posted.

Deadlines

A deadline in Little-JIL is a point in time by which a step must have completed.

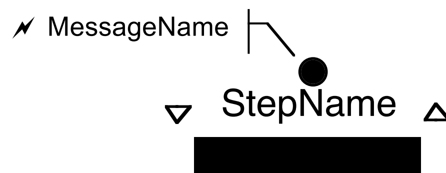
Deadline specification

Deadlines are associated with Little-JIL steps through the specification of the specially named ‘deadline’ parameter on the step. Steps with deadlines are indicated by the inclusion of clock hands on the interface badge for the step. Deadlines are specified as durations measured from the point in time when the step is posted. When a deadline expires, the exception **DeadlineExpired** is thrown to the parent of the step with the expired deadline.

Messages

Messages may be sent during the execution of a Little-JIL program to signal the occurrence of events. Messages in Little-JIL are sent by the interpreter to allow processes to react to their own execution, and may be sent by agents. Each step specifies the messages that the agent may send during the performance of that step. An agent may only send the messages associated with a step when the step is in the started state.

Message specification



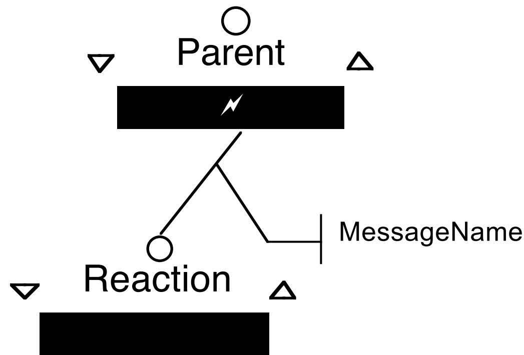
All of the messages that can be sent are specified in the step interface and are marked with a lightning bolt (⌁). Like exceptions, messages are specified in the parameter type model. Messages sent by the interpreter (e.g., step started) are specified in the appendix and do not need to be specified in 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 run-time 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.



The reactions associated with a Little-JIL step instance are only active while the instance is in the started state. Little-JIL reactions post sub-steps in response to messages. Since the sub-steps are posted immediately upon receipt of a message, these step instances can always be executed in parallel with the proactive sub-steps and each other. Similar to a parallel step, reactions delay the completion or termination of a step until the reactions complete; unlike a parallel step, posted reactions are not retracted and must be completed.

Message specification

As with exceptions, a message specification contains the type of the message and optionally a set of attribute name, value pairs. The pairs are written

name = *value*

Message matching

The matching rules for messages are the same as those for exceptions. Unlike exceptions, all started steps with matching message specifications handle messages. As with exception handlers, search for matching message specifications proceeds left to right, and stops when a matching reaction is found. This is similar to the top down search rules in the programming language Prolog.

Passing messages as parameters

As with exception handlers, message objects may be passed into any type compatible *in* or *in/out*, parameter in the reaction step. The parameter name and an arrow to the left of the message declaration represent the binding.

Restrictions on reactions

Step instances posted by reactions are forbidden from throwing exceptions. This restriction prevents a reaction from changing the proactive control-flow specified in a process. This means a reaction cannot have a pre- or post-requisite. However, the

sub-steps of a reaction step may throw exceptions as long as there is a handler for the exception within the reaction sub-tree.

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 instance from which the exception was thrown.

NoMoreAlternatives
Subtype of Process Exception. An exception that indicates a choice or try step cannot be continued as all of the alternatives have been exhausted.

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 exists that matches the specification.

ResourceUnavailable
Subtype of ResourceException. An exception that indicates that all resources matching the specification are being used.

Real-time Exceptions

DeadlineExpired
Subtype of ProcessException. An exception that indicates that the agent assigned to a step did not complete the step in the allotted time.

Messages

ProcessEvent
An abstract message type for all messages sent by the Little-JIL interpreter.
Attributes:
SourceStep: The step instance from which the event was sent.

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

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.

StepOptedOutEvent
Subtype of StepStateChangeEvent. A message that indicates that an optional step has been ‘opted-out.’

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.