Run Language

Version 41336de

Yurii Rashkovskii <yrashk@gmail.com>

June 13, 2021

This is a working draft of the Run Language motivation, specification and reasoning.

1 Introduction

Clarity in business processes spanning organizational structures, software systems and domains is an ongoing concern. Runaway complexity, underspecified procedures lead to erroneous actions or lack of appropriate actions, increase management overhead and inflate costs of automation and its maintenance.

Run is a modern, high-level language for defining, deploying and maintaining the lifecycle of executable business processes that is designed to address such concerns.

2 Background

There are a lot of existing business process languages, including but not limited to BPDM, BPEL4WS, BPML, BPMN, BPSS, EPML, OWL-S, PNML, UML Activity Diagram, WS-CDL, WSCI, WSCL, WSFL, XLANG, XPD, XPDL, YAWL

In the past few decades, a lot of efforts have been made to formalize workflow patterns into a language that can be used by non-technical and technical stakeholders alike in order to maintain a shared understanding of processes, reasoning and decision making. This has resulted in development and adoption of such languages as BPEL, BPMN, YAWL and alike.

In this section, for the purpose of clarity, we'll focus on BPMN as a baseline. It's a well-established and documented language supported by multiple software vendors.

There are a few important properties that BPMN has:

- Visual language (flow diagrams) that provides a comprehensive, bird's-eye view
- Execution semantics that allow for a mechanical interpretation (*execution*) of documents
- Introduction of useful standard workflow primitives such as events, boundary events, timers, compensations and so on to enable standardized handling of complex cases

BPMN does come with some trade offs:

Examples: *Complex Gateway*, what does it do? What's the difference between a *signal* and a *message*? How does message delivery differ based on the kind of *correlation* used?

- The flow elements have their execution semantics defined and one needs to know to construct the flow in a correct manner. Therefore, it implies the need for a translation between processes laid out in a natural language to the language of activation of flow elements according to their execution semantics. So, instead of describing what *needs to occur*, one needs to model according to what the flow element or their composition will do. It gets especially challenging when flow element's own naming does not reveal its function.
- Authoring diagrams is not the most natural user experience. A significant amount of research needs to be done in order to figure out a truly effective way to author diagrams, considering the amount of possibilities afforded by the standard. It becomes even more difficult if the document needs to be executable, as it'll start requiring changes of authoring mode between the diagram, flow element properties and text-based expressions.
- Authoring programs (because executable documents are effectively programs) as diagrams is not the most anticipated approach for most of software engineers and it is not the best fit for their tooling and practices. While BPMN documents are XML documents and can be edited as text, at that point they become counter-productive as authoring through long-winded XML formalisms makes it difficult to both write and read them, therefore defying the core promise of providing understanding and a high-level view.

- Lack of composition. While almost any workflow can be expressed in BPMN, it can become repetitive if certain patterns are to occur repeatedly. As an example, we can imagine a case for re-trying an activity with an exponential backoff, with a maximum number of retries. It's implementable, but copying it around can quickly become a chore and be error-prone.
- Lack of first-class execution tracing. *TO BE CONTINUED*

We propose that one can derive benefits from patterns identified in by BPMN and similar approaches while mitigating the aforementioned trade-offs by introducing a high level text-based language that focuses on *what needs to occur* as opposed to connecting flow elements that have their own behavior. It will allow to express complex behaviours that don't need to be deconstructed through the prism of flow elements' execution semantics.

The language should be at least somewhat familiar to software engineers to be accepted as a reasonable tool in their toolbox and play well with established practices.

At the same time, the language should bear at least some level of general readability to avoid it being a complete gibberish for those who aren't software engineers.

The language should also retain the high level intent of the authors in order to be transformable to and from different representations, such as visual diagrams (as these are indeed very useful for process comprehension).

This document introduces the **Run Language** to address this proposition.

3 Goals

3.1 Platform Pervasiveness

It is important that Run programs can operate in different environments and platforms, such as servers and workstations, mobile devices, browsers, microcontrollers, etc. Being able to describe processes that span operational backends, consumer and IoT devices allows for most comprehensive capture of the domain.

3.2 Compilation

Run is intended to be run using a small specialized VM (*Virtual Machine*) in order to spare each platforms from implementing a parser and a high-level interpreter. This ensures it is easier to implement Run environments on any platform.

Going further, it is highly desirable to make language compilable to other targets, such as *native code* (for **performance**), other languages (for **cohesive integration**) and special environment languages, such as *smart contract languages* in **blockchains** or *verification* languages such as WhyML for **critical processes**.

3.3 Ease of Comprehension

Run programs should be comprehensible in their textual and other representations with little familiarity of its specifics. In other words, programs should be self-explanatory at least at a high-level.

3.4 Intent Retention

To the extent possible, Run programs should be able to retain the original intent of the author without having it conceptualized using a different domain language.

This specification expends an effort to avoid *naming things* where possible in order to avoid such conceptualization.

3.5 Observability

TO BE CONTINUED

3.6 Continuous Capture

Run acknowledges that the world never stops and changes continue to occur. To that end, it is designed to support continuous information passage. This is also known as *streaming*.

4 Typing

Run employs a lightweight static typing system to ensure that the values used throughout its programs can be validated ahead of execution.

TO BE CONTINUED

5 Expressions

This section presents basic expressions (building blocks) of the core language. More forms are introduced in other sections.

5.1 Comments

Comments are considered expressions because they can be used to annotate the program and these annotations can be used when transforming programs to other representations, such as diagrams. Even more importantly, since they are considered first-class expressions, they can be used in place of other *executable* expressions in under-specified programs that can only be executed in simulation that assigns typed values to these expressions.

Line comments start with a double-slash (`//`) or a pound (hash) sign (#) and run until the end of the line or the end of the file (whichever occurs first)

Block comments start with a slash followed by an asterisk (`/*`) and run until a closing asterisk followed by a slash (`*/`). Nested block comments are supported.

```
# This is a line comment
// This is a line comment
/* This is a block comment
   that spans multiple lines
*/
```

5.2 Literals

Literals are expressions that evaluate to themselves.

5.2.1 Booleans

Boolean can be either `true` or `false`.

5.2.2 Strings

Strings are UTF-8 encoded Unicode strings and are enclosed between matching double (") quotes. The double quote character can be escaped with a blackslash (\). Backslash can be escaped with an extra backslash.

Strings can span multiple lines. If follow-up lines are indented to start at a column following the column of the opening quote, the leading whitespace will be removed.

```
"This is a string"
"This is what we call a \"string\""
"This is a blackslash: \\"
"This is a
 multi-line string"
```

5.2.3 Numbers

Numbers can be either integers or floating-point numbers.

5.2.4 Integers

Decimal integers are of the following format: '[-](0-9)+' and can be interspersed with underscore ('_') to ease comprehension ('100_000_000' being equivalent to '100000000'). The underscore can not be adjacent to period ('.') or the optional sign.

Binary integers can be encoded using the notation of leading zero and a lowercase letter 'b' ('0b') followed by a sequences of zeroes and ones.

Octal integers can be encoded using the notation of leading zero and a lowercase leter 'o' ('0o') followed by a sequence of '0-7'.

Hexadecimal integers can be encoded using the notation of leading zero and a lowercase letter 'x' ('0x') followed by a sequence of '0-F'.

All binary, octal and hexadecimal integers can be interspersed with underscore ('_') to ease comprehension as well. They can not, however, start with the underscore.

5.2.5 Floating-point

Decimal floating-point numbers are of the following format: '[-](0-9)+\.(0-9)+' and can be interspersed with underscore ('_') to ease comprehension ('100_000_000.00' being equivalent to '100000000.00'). The underscore can not be adjacent to period ('.') or the optional sign.

5.2.6 Scientific Notation

Scientific notation can also be used. It follows the format of '[-](0-9)+(\.(0-9)+)?[E|e]-?(0-9)+'. It can also be interspersed with underscore ('_') to ease comprehension except adjacent to 'E' or 'e' and the (optional) sign.

5.3 Call

This is one of the most useful building blocks. Since Run Language is mostly seen as a glue to connect various pieces, it needs to call into other systems, as well as being able to re-use building blocks defined in it.

TO BE CONTINUED

5.4 Logical Expressions

- **Logical AND**: `expr1 and expr2` is a boolean expression stating that both `expr1` and `expr2` have to be `true` in order for this expression to be `true`
- **Logical OR**: `expr1 or expr2` is a boolean expression stating that either `expr1` or `expr2` have to be `true` in order for this expression to be `true`
- **Logical NOT**: `not expr` is a boolean expression stating that `expr` should be `false` in order for this expression to be `true`

5.5 Comparison Expressions

- **Equal**: `expr == expr2` is a boolean expression stating that `expr1` must be equal to `expr2` in order for this expression to be `true`
- **Unequal**: `expr != expr2` is a boolean expression stating that `expr1` must not be equal to `expr2` in order for this expression to be `true`
- **Less**: `expr < expr2` is a boolean expression stating that `expr1` must be strictly less than `expr2` in order for this expression to be `true`
- **Less or equal**: `expr <= expr2` is a boolean expression stating that `expr1` must less than or equal to `expr2` in order for this expression to be `true`
- **Greater**: `expr > expr2` is a boolean expression stating that `expr1` must be strictly greater than `expr2` in order for this expression to be `true`
- **Greater or equal**: `expr >= expr2` is a boolean expression stating that `expr1` must greater than or equal to `expr2` in order for this expression to be `true`

5.6 Arithmetic Expressions

TO BE CONTINUED

5.7 Grouping Expresssion

Expressions can be groupped using parentheses: (and). For example, `1 + 2 * 2` evaluates to `5`, but `(1 + 2) * 2` evaluates to `6`.

5.7.1 Precedence

TO BE CONTINUED

5.8 Block

A block expression is simply a grouped sequence of expressions enclosed within curly brackets. It implies sequential interpretation of these contained expressions.

```
{
   // expression E_0
   // ...
   // expression E_n
}
```

5.9 Pure Expressions

All expressions that are idempotent and have no side effects are considered pure. The best way to think of this is to consider a class of expressions that do not alter the state of the execution in any way (even such as making any calls, even if they are idempotent).

5.10 Conditional Expressions

A conditional `if` expression is used to make a choice based on a boolean-typed pure expression (*condition*), followed by a block expression to execute in case if the boolean expression evaluated to `true` and optional `else` conditional or block expression in case if it evaluated to `false`.

Purety of the condition expression is important to ensure that it can be executed any number of times without any effect to the execution

```
if /* bool expression */ { /* expression Esuccess> ... */ }
```

```
if /* bool expression */ {
  /* expression Esuccess ...*/
} else {
  /* expression Efail ... */
}
```

```
if /* bool expression */ {
  /* expression Esuccess ... */
} else if /* bool expression */ {
  /* ...  */
}
```

There are a few reasons why success and failure expressions can only be block expressions:

Firstly, it resolves the ambiguity of `else` branches in the following case (should non-block expressions have been allowed):

```
if condition doSomething() if anotherCondition doSomethingElse() else doTheOpposite()
```

To which `if` condition does the `else` branch belong?

Secondly, it has been known that `if` branches without curly brackets can lead to misperceptions boundaries, so it's best to avoid them.

5.11 Concurrent Expressions

A concurrent expression is formed by an infix *pipe* operator: expr1 | expr2, denoting that `expr1` and `expr2` are concurrent to each other. A leading pipe can also be used to allow for visualized indenting of concurrency:

```
| a()
| b()
| c()
```

These expressions are one of the most important building blocks of Run programs as they allow to describe logic that should happen concurrently, and is a basis for a lot of common workflow patterns.

Concurrent expression produces a continuous output of completion tokens until all concurrent expressions have been completed.

5.12 Definition Expressions

5.13 Iteration