



# EASy Runtime Variability Instantiation Language: Language Specification

**Development Version 0.21 of 16. March 2016**

**H. Eichelberger, K. Schmid**

**Software Systems Engineering (SSE)**

**University of Hildesheim**

**31141 Hildesheim**

**Germany**

*Abstract*

*Deriving configured products in product line settings requires turning variabilities into artifacts such as source code, configuration files, etc. Actually, this can also be done at runtime in terms of re-configuration for adaptive software systems. The EASy Runtime Variability Instantiation Language (rt-VIL) described in this document is an optional extension to the EASy Variability Instantiation Language (VIL). It allows mapping of monitored values to IVML runtime variables, strategy and tactic-based runtime re-configuration and, finally, enactment of the changes to the underlying system. Moreover, it provides all capabilities of VIL and, thus, even allows code instantiation at runtime.*

*In this document we define and explain the concepts of the Runtime Variability Instantiation Language (rt-VIL) for specifying how runtime re-configuration can be performed using EASy-Producer.*

## Version

| | | |
|---|---|---|
| 0.10 | 03. February 2015 – 29. June 2015 | Initial version of rt-VIL. |
| 0.15 | 10. August 2015-15. March 2016 | Fail statement for rt-VIL. |
| 0.20 | 16. March 2016 | initialize for rt-VIL |

## Document Properties

The spell checking language for this document is set to UK English.


## Acknowledgements

This work was partially funded by the

- European Commission in the 7th framework programme through the INDENICA project (grant 257483).
- European Commission in the 7th framework programme through the QualiMaster project (grant 619525).


.

# Table of Contents

# Table of Figures

**No table of figures entries found.**

# 1 Introduction

This document specifies the Runtime Variability Instantiation Language (rt-VIL) in terms of the most current version of the language. While the Variability Instantiation Language (VIL) [3] is intended to support customization and instantiation of variability-rich software before runtime, rt-VIL is intended to support re-configurations and adaptivity at runtime

The runtime variability instantiation language (rt-VIL) is an extension of VIL inspired by concepts of Stitch [13] and S/T/A [14] in order to enable runtime instantiation and runtime reconfiguration / adaptation. Basically, rt-VIL follows the core principles DSPLs [10, 11], i.e., it relies on an explicit product line variability model (IVML), which makes runtime-variability available explicit and enables runtime re-configuration / adaptation through monitoring the execution environment and performing adequate runtime changes through re-configuration. Although traditional pre-runtime product line capabilities are not required in DSPLs [10, 11], we explicitly support product line instantiation at runtime through VIL capabilities. As a valid configuration is also required at runtime, we utilize runtime reasoning (in terms of the EASy IVML reasoning support [7]) for detecting invalid configurations and also for value propagation. For enacting runtime changes to the underlying system, we rely on architectural bindings and a translation of the runtime configuration to system-specific concepts. For more details on the approach, in particular in the context of adaptive real-time data stream processing, please refer to [12].

The remainder of this language specification is structured as follows: in Section 2, we define the syntax and semantics of rt-VIL on top of VIL. Please refer to the VIL language specification for details regarding VIL. In Section 3 we discuss the implementation status, in particular known deviations from this language specification. Finally, in Section 4, we provide the grammar of rt-VIL as a reference.

# 2 Runtime Variability Instantiation Language

The runtime variability instantiation language (rt-VIL) is an extension of VIL inspired by concepts of Stitch [13] and S/T/A [14] in order to enable runtime instantiation and runtime reconfiguration / adaptation. Basically, rt-VIL follows the core principles DSPLs [10, 11], i.e., it relies on an explicit product line variability model (IVML), which makes runtime-variability available explicit and enables runtime re-configuration / adaptation through monitoring the execution environment and performing adequate runtime changes through re-configuration. Although traditional pre-runtime product line capabilities are not required in DSPLs [10, 11], we explicitly support product line instantiation at runtime through VIL capabilities. As a valid configuration is also required at runtime, we utilize runtime reasoning (in terms of the EASy IVML reasoning support [7]) for detecting invalid configurations and also for value propagation. For enacting runtime changes to the underlying system, we rely on architectural bindings and a translation of the runtime configuration to system-specific concepts. For more details on the approach, in particular in the context of adaptive real-time data stream processing, please refer to [12].

### 2.1.1 Reserved Keywords

In rt-VIL, the following keywords are reserved and must not be used as identifiers. Please note that this set of reserved keywords is complemented by the keywords of VIL and those defined by the common VIL expression language.

- **`rtVilScript`** (replacing **`vilScript`**)
- **`breakdown`**
- **`fail`**
- **`objective`**
- **`persistent`**
- **`refail`**
- **`strategy`**
- **`tactic`**
- **`weighting`**

### 2.1.2 rt-VIL Script

In rt-VIL, a script (`rtVilScript`) is the top-level containing element. This element is mandatory as it identifies the scope for the strategies, tactics and VIL rules to be defined. The definition of a script requires a name in order enable script imports or extensions. rt-VIL scripts can be imported through their name using an import statement. Import statements are given before the script scope. Scripts can be extended akin to classes in object orientation allowing strategies, tactics or rules to be overridden. Further, the definition of a script requires a parameter list specifying the expected information from the execution environment. At least, the source artifact model (carrying the actual state of the runtime components), the actual (runtime) configuration with bound runtime decision variables (from monitoring / analysis), and the target artifact model (containing the instantiated artifacts or components, such as enactment commands) must be provided as parameters.

In a script, all decision variables of the (runtime) configuration are available through their (qualified) name. As IVML configurations may be partial or even composed dynamically, the actual decision variables, their types and type definitions are not necessarily known at the point in time when the script is specified. Thus, the validity of IVML names can only be determined at execution time of the script when also the actual (runtime) configuration is known. This may complicate the development of VIL scripts as actually unknown identifiers will at least lead to a warning. To support the Adaptation Manager in specifying valid and readable scripts, rt-VIL provides the **`advice`** annotation specifying the actual IVML model. This annotation allows using IVML identifiers instead of variable names and, moreover, maps all IVML types into the rt-VIL type system and enables dynamic dispatch also over IVML types.

**Syntax:**

```
//imports
@advice(ivmlName)
rtVilScript name (parameterList) extends name1 {
  //optional version specification
  //global variable declarations / definitions
  //strategy declarations
  //tactics declarations
  //rule declarations
}
```

**Description of syntax:**

- First, all referenced scripts are imported. Imports support the modularization of rt-VIL scripts, i.e., a top-level script may import separate scripts handling for example different lifecycle phases such as startup, runtime and shutdown of a pipeline. The syntax for an import statement follows VIL.
- An optional advice annotation may declare the underlying variability model to map IVML identifiers and types into rt-VIL. The syntax is akin to VIL. In contrast to VIL, unfrozen fields or attributes can be changed in rt-VIL.
- The keyword `rtVilScript` defines that the identifier name is defined as a new adaptation behavior script with contained strategies, tactics and VIL rules.
- The *`parameterList`* denotes the arguments to be passed to a rt-VIL script for execution. In rt-VIL, the source and target artifact model (given in terms of the type Project), the (runtime) configuration as well as the triggering event are passed in. An implementation shall treat unspecified parameters as undefined, i.e., expressions using this parameter are undefined.
- A rt-VIL script may optionally extend an existing (imported) rt-VIL script. This is expressed by the *`extends`* keyword indicating the name of the extended script. The strategies, tactics and rules of the extended script are available, in particular for overriding.
- Further, optional global variables may be declared or defined, respectively. All types defined by the rt-VIL/VIL type system can be used.

- A rt-VIL script may contain strategies, tactics and VIL rules. While there is no predefined sequence of defining strategies and tactics, the actual definition sequence is taken into account if, however, the respective preconditions does not lead to a unique selection of strategies or tactics at execution time. VIL rules can be used to define functions or instantiation tasks. The sequence of VIL rules is not relevant for execution. Extending scripts may override strategies, tactics and rules by specifying the same signature as the original declaration in an extended script. Further, extending scripts may participate in dynamically dispatched execution by defining new typed alternatives.

**Example:**

```
@advice(QM)
rtVilScript QualiMasterFinancial (Project source,
  Configuration config, Project target,
  AdaptationEvent trigger) {
    // variables, strategies, tactics, etc.
}
```

The example above shows a very basic, empty rt-VIL script. The script is linked against the QualiMaster variability model (just called QM) and called `QualiMasterFinancial`. The script receives the source artifact model, the (runtime) configuration, the target artifact model (that shall be modified) and the actual adaptation event.

### 2.1.3 Strategy

A strategy represents a high-level adaptation objective capturing the logical aspect of planning an adaptation. It encapsulates the relevant sub-strategies and tactics for the realization of the stated adaptation objective. The impact of executing strategies can be recorded in the QoS database and used to realize proactive adaptation.

Below, we discuss the execution semantics of a rt-VIL script in order to highlight the interplay of strategies, tactics and VIL rules.

- First, the initialization of the script is performed. This is done for each rt-VIL script by executing the predefined (overridable) rule only once

  ```
  initialize (Configuration config)
  ```

- Then, the applicable top-level strategies (considering also script parameters as described below) are determined. Multiple strategies may define the same objective so that additional preconditions (including the triggering event type) can be used to differentiate among the applicable strategies. If still multiple strategies are applicable, they are processed in order of their specification. Strategies can handle distinct events, e.g., a startup event to initialize the runtime variables. Initialization of runtime variables such as the active algorithm or functional parameters may also be given in terms of default values in the configuration.

- A strategy can specify multiple sub-strategies and tactics to realize the adaptation towards the actual (failing) objective. The selection of the sub-strategies and tactics during script execution can be static or dynamic. In the

static case, the first applicable sub-strategy or tactic in the given sequence is executed depending on their preconditions. In the dynamic case, a weighting function determines the ranking of the sub-strategies and tactics, which, in turn, determines the sub-strategy or tactic to be executed. The weighting function may consider historical information such as the previous impact depending on the functions provided by specific rt-VIL types. Further, we allow sub-strategies to enable a hierarchical breakdown of the adaptation specification along nested structures of the underlying application and, thus, dynamic execution also on the nested levels.

- Ultimately, a tactic is executed to determine the new settings of the runtime configuration. However, a tactic may fail, e.g., as its changes lead to an invalid runtime configuration detected by runtime reasoning. A tactic may revert to the last valid configuration, signal its failure or even cause the end of the script execution. In case of a failing tactic, the strategy continues with the next applicable sub-strategy or tactic in the sequence of ranking, respectively.
- A strategy succeeds if at least one of its sub-strategies or tactics succeeds. Finally, this determines whether the top-level strategy succeeds or fails.
- If the executed top-level strategy succeeded, the enactment of the runtime configuration starts, i.e., a predefined but overridable VIL-rule (see below) is executed that specifies the mapping from changed decision variables into enactment commands.

**Syntax**:

As discussed above, a strategy can have an objective, preconditions, or a weighting function and denote sub-strategies and tactics that are supposed to handle the situation indicated by the objective. The syntax of the strategy header follows the syntax of VIL rules.

The body of a strategy is separated into three parts, namely

- the objective (along with supporting local variables),
- the breakdown of the strategy possibly utilizing a weighing function into sub-strategies and tasks and
- post processing.

Actually, the objective is optional, in particular to support startup and shutdown strategies, which may need to be executed regardless of objectives. Also the weighting function is optional in order to enable static (reactive) rule-based adaptation. Please note that both, objective and weighting function may rely on VIL rules or basic operations that can be provided by programmed extensions through the rt-VIL type system. Sub-strategies and tactics can be stated using different syntactical forms as indicated in the syntax below, e.g., including a logical guard expression, a descriptive record (supporting the weighting function) or a maximum execution time. Finally, post processing can execute further VIL rules, e.g., to revert to a previous runtime configuration.

```
strategy name (ParameterList) = post : pre {
```

```
  // local variable declarations
  objective expression;
  breakdown {
    weighting (name : expression);
    // optional sub-strategies
    strategy name(ParameterList);
    strategy (guardExpression) name(ParameterList);
    strategy (guardExpression) name(ParameterList)
      with (name = value);
    strategy guardExpression name(ParameterList)
      @numExpression;

    // tactics
    tactic guardExpression name(ParameterList);
  }
  // post processing
}
```

**Description of syntax:**

- The keyword `strategy` indicates on this level the declaration of a strategy.
- A strategy is identified by its *name*, e.g., for referencing the strategy in other strategies.
- A strategy can declare parameters, which can be used within the strategy or for defining the pre- or post-condition. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Top-level strategies refer to the parameters of the containing script, either specifying all parameters declared by the script (in case of a strategy that performs instantiation) or by just declaring the actual runtime configuration and the triggering event. Sub-strategies may declare specific decision variable types instead of the entire runtime configuration. Further, a strategy may declare the specific event type it reacts on as parameter (and as an implicit precondition).
- Akin to VIL rules, a strategy may have logical pre- and post-conditions. As multiple strategies may define the same objective, the precondition can be used to select among the applicable strategies. If given, the post-condition is checked after the execution of a strategy to determine (in addition to the objective) whether the application of the strategy was successful. If neither pre- nor post-condition are given, also the separating colon can be omitted.
- Within the rule body marked by curly brackets, first local variables can be declared and initialized. In particular, information that is relevant to the calculation of the objective or the weighting function can be collected here, i.e., these variables are intentionally visible to the entire strategy block.
- The objective states a logical condition utilizing observed quality properties (via the runtime variables of the configuration) in expressions such as their

target range or utility / cost functions. Please note that the objective is a part of the precondition, which is syntactically highlighted due to its importance for adaptation.

- The breakdown block is executed once to collect the sub-strategies and tactics. This is required to enable the collection of an unknown number of alternatives at design time due to the openness of the configuration in QualiMaster as discussed above. In a second step, the identified sub-strategies and tactics are ranked and executed.
    - The weighting function enables the dynamic selection of the contained sub-strategies and tactics for both, reactive adaptation (without updated common knowledge) and for proactive adaptation (with prediction or updated common knowledge such as QoS impacts). The weighting function dynamically determines the ranking of the sub-strategies and tactics. Basically it is a function that maps the alternatives (sub-strategies or tactics) under consideration of the actual system state (taken from the runtime configuration or the runtime components) to a real value, e.g., using a utility-cost calculation. Thereby, it acts like an iterator over the alternatives and selects the option with maximum value with access to additional information given in the listing of the sub-strategies and tactics. In particular, rt-VIL operations provide access to the historical impact of a given alternative, the prediction of a certain quality property, or the aggregated quality of a (desired) pipeline. If specified, the weighting function must be given directly at the beginning of the breakdown block.
    - The remainder of the block declares the alternative sub-strategies and tactics. As described above, sub-strategies and tactics can be guarded by an expression, which influences the ranking. Sub-strategies and tactics are referenced by their signature and, in case that a sub-strategy or tactic is used multiple times, a descriptive record (given in terms of name-value pairs) can be given to support the calculation of the weighting function. Please note that the union of all name-value pairs must be consistent over the types implicitly assigned to the names via the individual value expressions used. If no name-value expressions are used, then a Strategy or Tactic instance (see Section 2.2.1) is passed to the weighting function. If at least one name-value expression is used, the implicit record is created and passed to the weighting function. Finally, a maximum execution time (in terms of an integer expression) can be stated, i.e., a relative time bound within the sub-strategy or tactic is either completed or it fails. Timeouts not given as a positive value are ignored. If a tactic succeeds but does not change the configuration, the next tactic in ranking will be executed.
- The post processing part allows to execute further VIL rules and operations, e.g., to revert to a previous successful runtime configuration. In particular, the post processing part can explicitly cause a strategy to fail (see Section

2.1.5 for the fail statement). Then the next possible strategy is executed. If there is no further strategy, the execution of the script terminates with an adaptation failure.

At the beginning of a strategy, the rt-VIL execution environment calls the (overridable) VIL rule

```
start (Strategy strategy, Configuration config)
```

The built-in implementation opens a transaction for tracking configuration changes. Finally, the execution environment calls the predefined rt-VIL rule

```
Boolean validate (Strategy strategy, Configuration config)
```

at the end of a (so far) successful tactic in order to validate the results of the executed operations. By default, this predefined rt-VIL rule performs runtime reasoning (on the changed variables). This method can be overridden to realize other forms of validation, e.g., for traditional architecture-based adaptation. If the validation or a strategy fails, the execution environment calls

```
failed (Strategy strategy, Configuration config)
```

to enable rollback or repair operations. By default, a failing strategy leads to a rollback of the previously opened transaction. If the strategy succeeds, the execution environment calls

```
succeeded(Strategy strategy, Configuration configuration)
```

to enable, which closes the previously opened transaction and commits the changed variables.

Please note that successful top-level strategies call further predefined rules for updating the common knowledge and enactment as we describe in Section 2.1.5.

**Example:**

The first example below illustrates a simple reactive strategy, which considers exactly one quality parameter, namely the pipeline throughput. As input, the strategy receives the actual QualiMaster runtime configuration (expressed through the type QM made available through a respective advice) and an unspecified adaptation event. In case that a pipeline does not fulfill an overall pipeline throughput value (here, a fictive configuration value specified as a fixed condition in a VIL rule), it aims at changing the pipeline with highest throughput deviation (`candidate`), with a priority on changing a single algorithm parameter. In the extreme case of no successfully executed tactic, the strategy just performs load shedding on the input of `candidate`. Please note that a top-level strategy is implicitly ended by a validation of the changed variables of the runtime configuration and, in case of success, an enactment of the changed variables.

```
strategy infrastructure (QM config, AdaptationEvent trigger) =
{
  setOf(Pipeline) issues = throughputFailingPipelines(config);
  Pipeline candidate = issues->sortedBy(Pipeline p |
     p.capacity).first();
```

```
  objective null != candidate;

  breakdown {
    strategy changeSingleParameter(candidate, trigger);
    strategy changeSingleAlgorithm(candidate, trigger);
    // extreme fallback
    tactic shedLoad(candidate, trigger);
   }
}
setOf(Pipeline) througputFailingPipelines(QM config) = {
  config.pipelines-> select(p|
    p.throughput < config.minPipelineThroughput);
}
```

The second example illustrates a strategy with dynamic selection of a comprehensive loop-enumeration of the tactics (based on all alternatives defined in the Configuration). As stated by the signature, the strategy handles a regular adaptation on a `FamilyElement` of a pipeline. The objective of the strategy is that no quality parameter of the family element fails (the related definition is not shown in the example). The weighting function targets the optimization of a utility-cost tradeoff given in terms of two VIL functions (not detailed below). Thereby, additional information defined by the listed tactics is used. For illustration, this example simply enumerates all possible tactics for all available algorithm parameters and family members using map expressions, the VIL version of a loop. Please note that a sub-strategy is implicitly ended by a validation of the changed variables in the runtime configuration.

```
strategy changeAlgorithm (FamilyElement elt,
  RegularAdaptationEvent trigger) = {
  setOf(Quality) failed = failedQualities(elt);
  objective failed.isEmpty();
  breakdown {
    weighting (e: utility(e.elt, e.quality)
      - cost(e.elt, e.quality));
    map(Quality q: failed) {
      map(Parameter p: elt.family.parameter) {
        tactic changeParameter(elt, trigger, p)
          with (elt = p, quality = q);
      }
      map(Algorithm a: elt.family.members) {
        tactic changeAlgorithm(elt, trigger, a)
          with (elt = a, quality = q);
      }
    }
  }
}
```

### 2.1.4 Tactic

A tactic defines the steps of adaptation to be carried out in order to achieve the objective defined by a calling strategy. A tactic cannot call other tactics or strategies, just rt-VIL operations or rt-VIL rules.

A tactic can have a pre-condition and a post-condition. A tactic is enabled if its precondition holds. A tactic is successful, if its operations are executed successfully and its (optional) post-condition holds. So far, tactics are rather similar to VIL rules. In contrast, tactics serve as a basis for recording the QoS impact through strategies.

Akin to a strategy, a tactic can explicitly fail (see Section 2.1.5 for the fail statement). Then the next possible tactic within the same strategy or the next possible strategy is executed. If there is no further tactic or strategy, respectively, the execution of the script terminates with an adaptation failure.

At the beginning of a tactics, the rt-VIL execution environment calls the (overridable) VIL rule

```
start (Tactic strategy, Configuration config)
```

The built-in implementation opens a transaction for tracking configuration changes. Finally, the execution environment calls the predefined rt-VIL rule

```
Boolean validate (Tactic tactic, Configuration config)
```

at the end of a (so far) successful tactic in order to validate the results of the executed operations. By default, this predefined rt-VIL rule performs runtime reasoning (on the changed variables). This method can be overridden to realize other forms of validation, e.g., for traditional architecture-based adaptation. If the validation or a strategy fails, the execution environment calls

```
failed (Tactic strategy, Configuration config)
```

to enable rollback or repair operations. By default, a failing strategy leads to a rollback of the previously opened transaction. If the strategy succeeds, the execution environment calls

```
succeeded(Tactic tactic, Configuration configuration)
```

to enable, which closes the previously opened transaction and commits the changed variables.

**Syntax:**

```
tactic name (ParameterList) post : pre {
  // local variable declarations
  // runtime configuration changes, rule calls
  // alternative or iterative execution
}
```

**Description of syntax:**

- The keyword `tactic` indicates on this level the declaration of a tactic.
- The *`name`* allows identifying the rule for explicit rule calls or for script extension.

- The *parameterList* specifies explicit parameters which may be used as arguments for precondition rule calls as well as within the rule body. Parameters are given in terms of types and parameter names separated by commas if more than two parameters are listed. Parameter must either be bound by the calling strategy.
- The optional post-condition post specifies the expected outcome of the tactic execution in terms of a logic expression.
- The optional precondition pre specifies whether the tactic is considered for execution at all (in addition to the precondition in the strategy). If neither pre- nor post-condition are given, also the separating colon can be omitted.
- The body of the strategy is specified within the following curly brackets. Local variable declarations, rule calls, alternative and looped execution and, in particular, changes to the runtime variables of the configuration can be specified here.

**Example:**

This example continues the second example from Section 2.1.3 on changing algorithms and parameters of a QualiMaster pipeline. The tactic shown below performs all operations needed to change a tactic, i.e., it transfers the parameter values of the actual family to the new algorithm and finally changes the current algorithm of the actual family. Please note that a tactic is implicitly ended by a validation of the changed variables in the runtime configuration.

```
tactic changeAlgorithm (FamilyElement elt,
  AlgorithmAdaptationEvent trigger,
  Algorithm newAlgorithm) : {
  Algorithm current = elt.family.current;
  // transfer parameter
  map(Parameter p: elt.family) {
    newAlgorithm.algorithm.parameter
      ->select(q|q.name == p.name).first()
         .setValue(p.value);
  }
  // set algorithm
  elt.family = newAlgorithm;
}
```

### 2.1.5 Fail / refail statement

Adaptation strategies or tactics may determine that they cannot proceed and that another (less important) strategy or tactic in the same context shall take over. This can be achieved using the `fail` statement. In some cases, e.g., default dynamic dispatch tactics or strategies it is good to cause an already failing to fail again with the original reason. Therefore, rt-VIL offers the `refail` statement. Using a `refail` statement without previous `fail` statement is ignored. Both statements, `fail` and `refail` can be used within blocks, e.g., alternatives to express a conditional fail.

**Syntax**:

```
fail ”reason”? with intExpression?;
refail;
```

**Description of syntax:**

- The keyword `fail` indicates that a strategy or tactic shall immediately fail. This statement can be used only within tactics or strategies (post processing part).
- An optional textual reason for failing for documentation purposes. An implementation shall pass this information in a suitable way to the user (as an event, as logging, etc.).
- An optional integer expression indicated by the keyword **`with`** representing an error code as machine-readable form of a reason. An implementation shall pass this information in a suitable way to the user (as an event, as logging, etc.). If neither an error code nor a failure reason is given, an implementation may use -1 as default error code.
- The keyword `refail` indicates that a strategy or tactic shall immediately fail with the reason of the previous fail. If there was no previous fail, then refail does not have an effect. This statement can be used only within tactics or strategies (post processing part).

**Example:**

This example is based on the example from Section 2.1.4:

```
tactic changeAlgorithm (FamilyElement elt,
  AlgorithmAdaptationEvent trigger,
  Algorithm newAlgorithm) : {
  Boolean algorithmFound = false;
  // adjust algorithm as shown above
  if (!algorithmFound) {
    fail ”no algorithm found” 123;
  }
}
```

In case of a failure, the next viable tactic or strategy will be chosen. If the respective tactic is selected via dynamic dispatch upon another type, the type-specific tactic (the second one below) will not be executed, but the tactic acting as dispatch base (the first one below). In order to cause a failing of the `changeAlgorithm` tactic above to be effective, the typically empty base tactic contains a refail.

```
tactic changeParameter (FamilyElement elt,
  AdaptationEvent trigger) : {
  refail;
}

tactic changeParameter (FamilyElement elt,
  ParameterAdaptationEvent trigger) : {
  // do the parameter adaptation
```

```
}
```

### 2.1.6 Enactment

In runtime reconfiguration based on a variability model, we must validate the constraints of the variability model upon changes of runtime decision variables as explained above. Further, our approach aims to be independent from the underlying system and, thus, needs to specify the mapping from the runtime configuration to the system under adaptation.

In rt-VIL, enactment happens if a top-level strategy succeeds. Then, rt-VIL executes predefined rt-VIL rules (similar to the strategy method), which can be overridden if required. For updating the common adaptation knowledge, e.g., with respect to the impact, rt-VIL calls

```
update(Strategy strategy, Configuration configuration)
  update(Tactic tactic, Configuration configuration)
```

for each succeeded strategy and tactic. Finally, for the successful top-level strategy, rt-VIL executes

```
enact(Project source, Configuration config, Project target)
```

to map and execute the adaptation decisions. The three parameters correspond to the three parameters of the containing script, their types and sequence. Using projection functions of the Configuration type, the runtime configuration can be turned into a subset more appropriate for enactment. Note that there is no predefined enact rule so that it must be defined somewhere in the rt-VIL script.

**Example:**

An illustrating example combining the enactment of algorithm changes with triggering a subsequent wavefront is shown below as a continuation of the example in Section 2.1.4. Basically, the enact method receives the actual runtime configuration, projects it to the changed variables and considers each changed pipeline and the contained pipeline elements through dynamic dispatch (not all VIL rules for the pipeline types are shown). The enactment of the `FamilyElement` creates the respective commands for the system under adaptation, here in terms of a command sequence. Finally, the enactment rule schedules the subsequent pipeline elements for wavefront adaptation.

```
enact (Project source, Configuration config, Project target) {
  QM changed = config.selectChangedWithContext();
  map(Pipeline p: changed.pipelines) {
    map(Source s: p.sources) {
      enact(p, s);
    }
  }
}
enact(Pipeline pipeline, PipelineElement elt) {
```

```
  // dynamic dispatch default – do nothing
}

enact(Pipeline pipeline, FamilyElement elt) {
  Family family = elt.family;
  CommandSequence cmd = new CommandSequence();
  if (null != family.current) { // was changed?
    cmd.add(new AlgorithmChangeCommand(pipeline.name,
      elt.name, family.current.name);
  }
  map (Parameter p: family.parameters) {
    cmd.add(new ParameterChangeCommand(pipeline.name,
      elt.name, p.name, p.value));
  }
  map(Flow f: src) {
    PipelineElement e = f.destination;
    enact(e);
    cmd.add(new ScheduleWavefrontAdaptationCommand(
      pipeline.name, e.name));
  }
  cmd.execute();
}
```

### 2.1.7 Binding Runtime Variables

So far, we just assumed that certain runtime variables are somehow bound and contain values that reflect the system state at runtime. Actually, binding the values can be done in two ways, namely programmatically and through rt-VIL.

Basically, the underlying adaptive system will perform some form of monitoring to gather the system state and to use this information for adapting itself. As part of monitoring and analysis, the system can also directly bind the respective values in the configuration using the programming API of rt-VIL and IVML. However, in this case, changes to the structure of the variability model also require changes of the binding code and, in particular, programming complex bindings for nested or linked structures can be a complex task.

As a solution, the binding can be directly defined in rt-VIL. Therefore, rt-VIL provides the predefined VIL rule

```
bindValues (Configuration cfg, mapOf(String, Double) values)
```

which is initially defined empty and can be overridden if required. Thereby, `cfg` is the runtime configuration before adaptation and `values` are the values provided by monitoring (`mapOf(String, Double) values` must now be declared as an additional parameter of the script). The `bindValues` rule is executed before deriving the values of the global parameters, i.e., executing the first strategy. All changes done by bindValues to cfg are not recorded in the configuration change history.

**Example**

The example below (continuing the previous examples) illustrates the binding of runtime variables using the predefined `bindValues` rule. Actually, the binding is defined in a separate rt-VIL script called mapping to be imported by the main rt-VIL script. As described above, the header declares the specific parameter containing the mapping of monitored values. Actually, it can be rather tedious to work directly on this map. Thus, we assume that the application-specific extension for the underlying system defines a type called `SystemState`, which can be used to wrap the mapping and to access the mapping more conveniently, namely through the observables. An instance of this type is created in the first `bindValues` rule and used in the remainder part of the mapping. Actually, the relevant decision variables are traversed. In this example, the traversal is shown for computing machines (Machine) rather than for pipelines here as this can be done similarly as shown above using dynamic dispatch. The actual values for runtime variables of Machine are assigned in the second `bindValues` rule, which relies on `SystemState` to access the most recent value.

```
@advice(QM)
rtVilScript mapping(Project source, Configuration config,
  Project target, AdaptationEvent event,
  mapOf(String, Real) bindings) {

  // strategies, tactics, enactment

  bindValues(Configuration config,
    mapOf(String, Real) values) = {
    QM qm = config;
    SystemState state = new SystemState(values);
    map(Machine m: qm.machines) {
      bindValues(m, state);
    };
    // go on with relevant parts of the variability model
  }

  bindValues(Machine machine, SystemState state) = {
    machine.bandwidth = state.getMachineObservation(
     machine.name(), ResourceUsage.BANDWIDTH);
  }
}
```

Actually, a rt-VIL script can be executed through the API for just binding values, e.g., to combine monitoring and runtime reasoning over an IVML to detect violated SLAs.

## *2.2 Built-in types and operations of rt-VIL*

In this section we describe the built-in operations of rt-VIL, i.e. types and operations that are specific to rt-VIL and only available in rt-VIL. For describing the types and operations, we follow the conventions of the VIL language specification. Basically, all VIL types, operations and instantiators are also available in rt-VIL. Further, we distinguish among built-in types in Section 2.2.1

### 2.2.1 rt-VIL types

In this section, we detail the operations for the built-in VIL types.

#### 2.2.1.1 RtVilConcept

Represents a the supertype of strategies and tactics to use them, e.g., in weighting functions, update or enactment rule.

- **String getName () / String name ()**
  Returns the name of the *operand* strategy.

No automated conversions or explicit constructors are defined for this type.

#### 2.2.1.2 Strategy

Represents a rt-VIL strategy, e.g., in weighting functions, update or enactment rule. Inherits from `RtVilConcept`.

No automated conversions or explicit constructors are defined for this type.

#### 2.2.1.3 Tactic

Represents a rt-VIL tactic, e.g., in weighting functions, update or enactment rule. Inherits from `RtVilConcept`.

No automated conversions or explicit constructors are defined for this type.

#### 2.2.1.4 Configuration

The Configuration type corresponds to the VIL configuration type. The functions listed below are intended to support runtime instantiation and may change the configuration.

- **Configuration copy()**
  Copies the configuration in *operand*. The variables in the result of this operation can be modified independently from *operand*.
- **Configuration selectFrozen()**
  Projects *operand* to a configuration of frozen variables. A configuration that has been frozen once will remain frozen. This operation is intended to make pre-runtime instantiations explicit.
- **Configuration selectAll()**
  Projects *operand* to all variables (identity projection, i.e., including unfrozen ones). This operation is intended to distinguish between runtime and pre-runtime instantiation scripts. Please note, that a configuration that has been projected once to frozen variables will remain frozen.
- **Configuration selectChanged ()**

Projects *operand* to all changed variables.

- **Configuration selectChangedWithContext()**
  Projects *operand* to all changed variables including their context, i.e., their containing variables, their (frozen) variables. Further, the variables referring to them (including context) and the variables being referenced (including context) are contained in the projection.
- **ChangeHistory changeHistory() / ChangeHistory getChangeHistory()**
  Returns the change history of the *operand*, which is shared among all its projections.
- **Configuration reason()**
  Performs a re-reasoning of the *operand*, preferably by applying incremental reasoning techniques. Please check the result of `isValid()` before using the result configuration for instantiation. Changes through reasoning are tracked in the change history.
- **Boolean isValid()**
  Returns whether the configuration in *operand* is valid and may be considered for instantiation. In particular, performing reasoning on a configuration may result in an invalid configuration.

#### 2.2.1.5 DecisionVariable

The DecisionVariable type corresponds to the VIL DecisionVariable type. The functions listed below are intended to support runtime instantiation and may change their individual value over time.

- **clearValue()**
  Resets the value of *operand* as if it would not have been assigned before. This operation has no effect, if the variable is frozen.
- **Boolean isValid()**
  Returns whether the underlying decision variable of the *operand* is valid, i.e., all constraints are fulfilled. In VIL, this operation always returns `true`.
- **Boolean isEnacting()**
  Returns whether the underlying decision variable of the *operand* is currently enacting, i.e., being subject to a runtime change so that further changes may need to be prevented. In VIL, this operation always returns `true`. In rt-VIL, the return value of a certain decision variable may also include the enactment state of nested variables (depending on the implementation).

#### 2.2.1.6 ChangeHistory

Collects all changes done to a Configuration in a transition-based way.

- **start()**
  Starts a transaction on *operand*. All changes to variables in the configuration will now be stored in that transaction.
- **rollback()**
  Rolls back the changes in the most recent transaction of *operand*.
- **commit()**

Committs all changes collected in the most recent transaction of *operand* into the global change set.

### 2.2.2 Types of the underlying adaptive system

Selected types and operations of the underlying adaptive system can be mapped into rt-IVML in order to obtain certain information or to manipulate the system state during enactment. The mapping is system dependent and must be available during specification time (without execution of the operations) and at runtime.

# 3 Implementation Status

The development and realization of rt-VIL related tools is still in progress. In this section, we summarize the current status.

All functionality described in this document is implemented. However, rt-VIL is an optional experimental component, which is not part of the default EASy-Producer installation. For installing rt-VIL, please select the optional rt-VIL feature during the installation process.

# 4 rt-VIL Grammar

In this section we depict the actual grammar for rt-VIL. The grammar is given in terms of a simplified xText[1] grammar (close to ANTLR[2] or EBNF). Simplified means, that we omitted technical details used in xText to properly generate the underlying EMF model as well as trailing ";" (replaced by empty lines in order to support readability). Please note that some statement-terminating semicolons are optional in order to support various user groups each having individual background in programming languages.

```
ImplementationUnit:
  Import*
  Require*
  LanguageUnit*

LanguageUnit:
  (Advice)*
  'rtVilScript' Identifier
  '(' ParameterList? ')'
  (ScriptParentDecl)?
  '{'
    VersionStmt?
    rtContents
  '}' ';'?

rtContents:
  (
    GlobalVariableDeclaration
    | RuleDeclaration
    | StrategyDeclaration
    | TacticDeclaration
    | TypeDef
    | Compound
  )*

GlobalVariableDeclaration:
  'persistent'?
  VariableDeclaration

StrategyDeclaration:
  'strategy' Identifier
  '(' (ParameterList)? ')'
  '=' RuleConditions?
  '{'
  VariableDeclaration*
  ('objective' Expression ';')?
  ('breakdown' '{'
```

[1] http://www.eclipse.org/Xtext/

[2] http://www.antlr.org

```
    weighting=WeightingStatement?
    BreakdownElement+ '}'
  )
  RuleElement*
  '}'
  ';'?

BreakdownElement:
  VariableDeclaration
  | ExpressionStatement
  | BreakdownStatement

WeightingStatement:
    'weighting' '(' name=Identifier ':' expr=Expression ')' ';'

BreakdownStatement:
  ('strategy' | 'tactic')
  (
    '('LogicalExpression ')'
  )?
  QualifiedPrefix
  '(' ArgumentList? ')'
  (
    'with' '('    BreakdownWithPart (',' BreakdownWithPart)')'
  )?
  (
    '@' Expression
  )?
  ';'

BreakdownWithPart:
  Identifier '=' Expression

TacticDeclaration:
  'tactic' Identifier
  '(' (ParameterList)? ')'
  '='RuleConditions?
  RuleElementBlock
  ';'?

RuleElementBlock:
  '{' IntentDeclaration?  RuleElement* '}'

RuleElement:
  VariableDeclaration
  | ExpressionStatement
  | FailStatement

IntentDeclaration: 3
  'intent' ExpressionStatement
```

[3] Just syntax for now, intended for detailed adaptation logs about the intent of an adaptation step.

```
FailStatement:
  (('fail' STRING? 'with' Expression?) | 'refail') ';'
```

# References


[1] Project homepage AspectJ, 2011. Online available at: http://www.eclipse.org/aspectj/.

[2] Eclipse Foundation. Xtend - Modernize Java, 2013. Online available at: http://www.eclipse.org/xtend.

[3] INDENICA Consortium. Deliverable D2.1: Open Variability Modelling Approach for Service Ecosystems. Technical report, 2011. Available online at http://sse.uni-hildesheim.de/indenica

[4] INDENICA Consortium. Deliverable D2.4.1: Variability Engineering Tool (interim). Technical report, 2012. Available online http://sse.uni-hildesheim.de/indenica

[5] INDENICA Consortium. Deliverable D2.2.2: Variability Implementation Techniques for Platforms and Services (final). Technical report, 2013. Available online at http://sse.uni-hildesheim.de/indenica

[6] Object Management Group, Inc. (OMG). Object Constraint Language. Specification v2.00 2006-05-01, Object Management Group, May 2006. Available online at: http://www.omg.org/docs/formal/06-05-01.pdf.

[7] H. Eichelberger, S. El-Sharkawy, C. Kröher, K. Schmid, IVML language specification. http://projects.sse.uni-hildesheim.de/easy/docs/ivml_spec.pdf [validated: February 2015].

[8] Richard M. Stallmann, Roland McGrath, and Paul D. Smith. GNU Make - A Program for Directing Recompilation - GNU make Version 3.82, 2010. Online available at: http://www.gnu.org/software/make/manual/make.pdf.

[9] The Apache Software Foundation. Apache Ant 1.8.2 Manual, 2013. Online available at: http://ant.apache.org/manual/index.html.

[10] S. Hallsteinsen and M. Hinchey and S. Park and K. Schmid, Dynamic Software Product Lines, IEEE Computer 41 (4), pages 93-95, 2008

[11] M. Hinchey, S. Park, and K. Schmid. Building dynamic software product lines. *Computer*, 45(10):22–26, 2012.

[12] QualiMaster Consortium, Deliverable D4.1, Quality-aware Pipeline Modeling, Technical report, 2014, Available online at http://qualimaster.eu

[13] S.-W. Cheng, D. Garlan, B. Schmerl. Stitch: A Language for Architecture-based Self-adaptation. *J. Syst. Softw.*, 85(12):2860–2875, 2012.

[14] N. Huber, A. van Hoorn, A. Koziolek, F. Brosig, S. Kounev. Modeling Run-time Adaptation at the System Architecture Level in Dynamic Service-oriented Environments. *Serv. Oriented Comput. Appl.*, 8(1):73–89, March 2014.