Software Configuration Management Framework Specification

Document Identification	Software Configuration Management Framework Specification
Document Version	0.1.4
Document Type	Specification
Project Name	Software Configuration Management Framework
Project Identification	SCMF
Document Owner	Sergii Shmarkatiuk

1. Contents

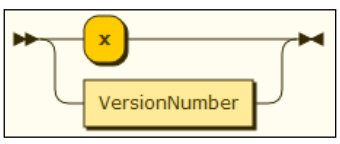
2. Glossary

2.1. Conventions

- EBNF (Extended Backus-Naur Form) convention syntax description (plaintext representation) is marked throughout the text of this document with green color:

```
VersionCompound ::= "x" | VersionNumber
```

- EBNF (Extended Backus-Naur Form) convention syntax description (visual representation) is represented in following form:



- *Version template* examples are marked throughout the text of this document with grey color:

```
N.?.?.L
```

- *Version template* element examples are marked throughout the text of this document with pink color:

?

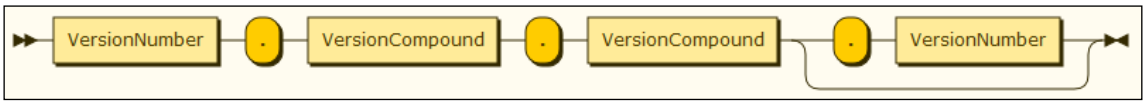
- *Version* examples are marked throughout the text of this document with blue color:

`1.x.x`

2.2. Definitions

- ***SCMF (Software Configuration Management Framework)*** – list of conventions and best practices used for the effective Software Configuration Management activities (version control, deployment management, build management, continuous integration, dependency management, branches management, merge management, release management) organization and its effective maintenance/support.
- ***Version*** – special type of marker used for the purpose of distinguishing between different kinds of artifacts which have been produced/created at one of the *SDLC stages.* Usually consists of leading *version number*, trailing *version number* and one (or two, in the case of *integration artifact version*) *version compounds* separated by the period:

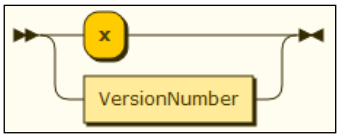
```
Version ::= VersionNumber "." VersionCompound "."VersionCompound ("."VersionNumber )?
```



Picture 1

  - ***Version compound*** – part of *version*, separated from other *version compounds* with the period ("**.**" symbol). *Version compound* value is either *version number placeholder* or v*ersion number*:

```
VersionCompound::= "x" | VersionNumber
```

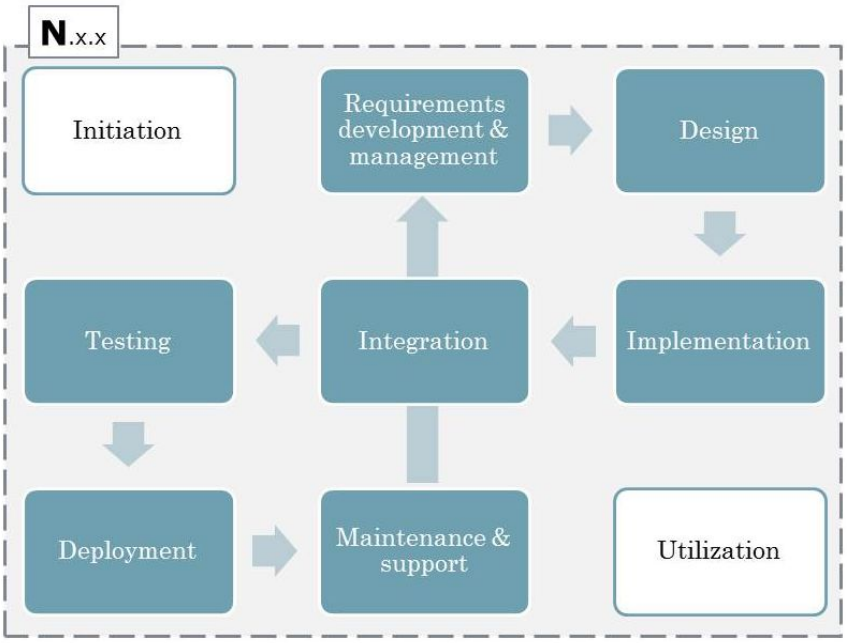


Picture 2

Version compound is usually represented by question mark ("**?**" symbol) as a part of a *version template*. Example: `N.?.?.L`.

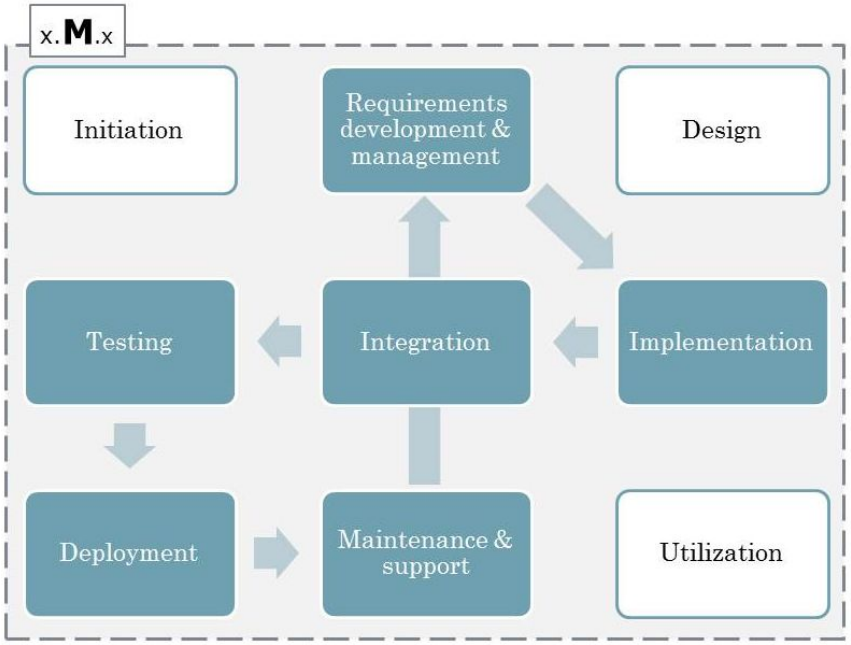
    - ***Version number placeholder***– '**x**' symbol used as a placeholder for *version number* value. It is used instead of actual *version number* in the case when numbering is not applicable to the current context. For example, *support branch version* can have value `1.x.x`. '**x**' symbols show that *release version number* and *build version number* are not applicable to the context of *support branch* versioning (because there are no corresponding versioned *release artifact* and *build artifact*). Thus, initial *trunk* state can be marked with `x.x.x` version as long as numbering is not applicable to the initial *trunk* context at all (there are no versioned artifacts which correspond to the *major version number*, *release version number* and *build version number*).
    - ***Version number*** – integer used for incremental version numbering; it is a numeric representation of corresponding *SDLC iteration*; also it represents successive instance of some *versioned artifact*. It usually starts from 0 (even in case of *major version number*, see paragraph 3). *Version number* is usually represented as a part of a *version template* using one of the following symbols: '**N**', '**M**', '**K**', '**L**'. Usage of specific letter depends on the context. Example: `N.M.K.L`.

      Major version number – integer used for incremental version numbering of *support branches*. Major version corresponds to the most long-lasting *SDLC iteration* (*major version development*) – starting from the requirements analysis and development phase, ending with the maintenance & support phase. *Major version number* is usually represented as a part of a *version template* using symbol '**N**':



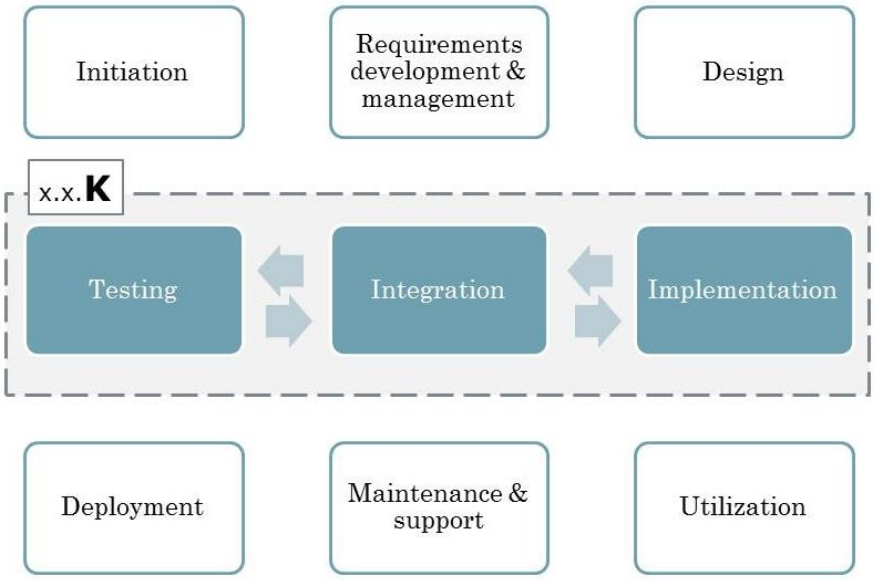
Picture 3

- ***Release version number*** – integer used for incremental version numbering of *release branches* using *major inheritance scope*. Release version corresponds to *SDLC iteration* (*release version development*) starting from the requirements analysis and development phase, ending with the maintenance & support phase, omitting design phase (sometimes it might be reduced to "implementation → deployment" *SDLC iteration*). *Release version number* is usually represented as a part of a *version template* using symbol '**M**':



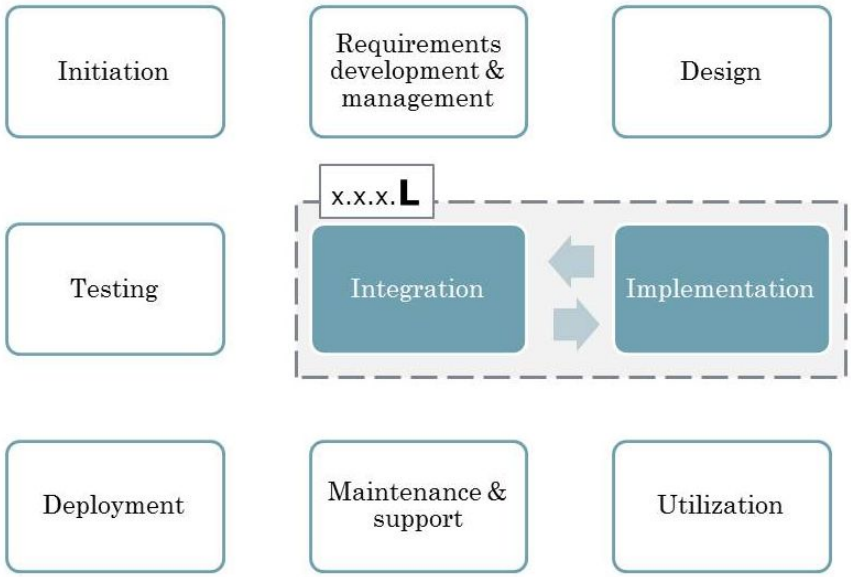
Picture 4

- ***Build version number*** – integer used for incremental version numbering of *build artifacts (*using *precursory inheritance scope)* or *release artifacts (*using *release inheritance scope)*. Build version corresponds to *SDLC iteration* starting from the implementation phase, ending with testing phase. *Build version number* is usually represented as a part of a *version template* using symbol '**K**':



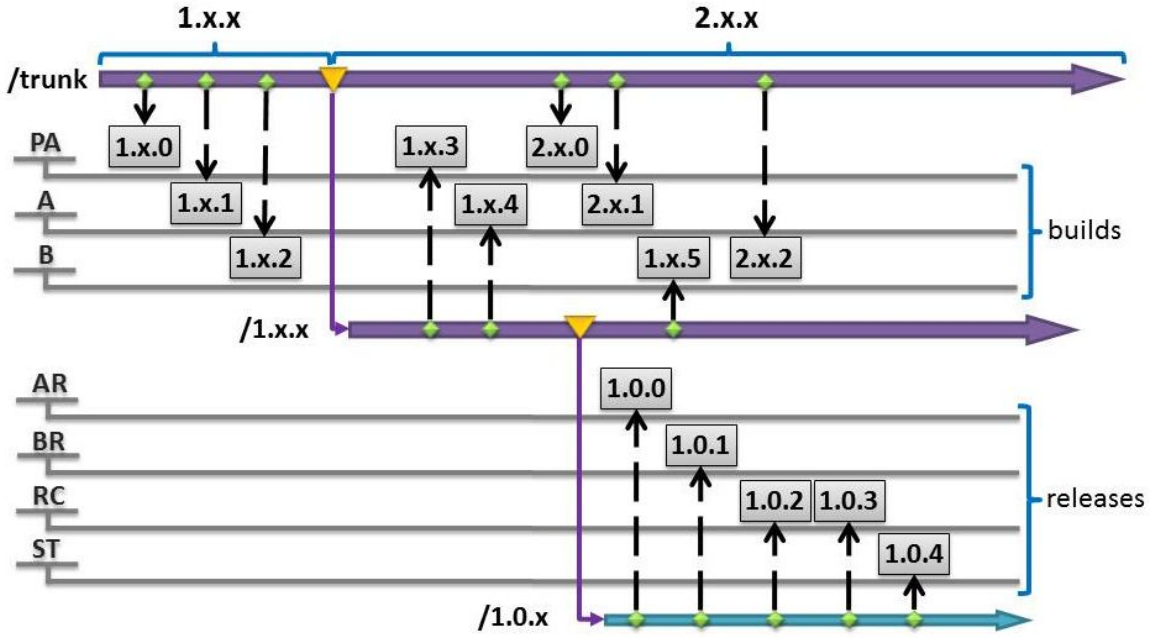
Picture 5

- ***Integration version number*** – integer used for incremental version numbering of *integration artifacts* using *integration scope*. Integration version corresponds to *SDLC iteration* starting from the implementation phase, ending with integration phase. *Integration version number* is usually represented as a part of a *version template* using symbol '**L**':



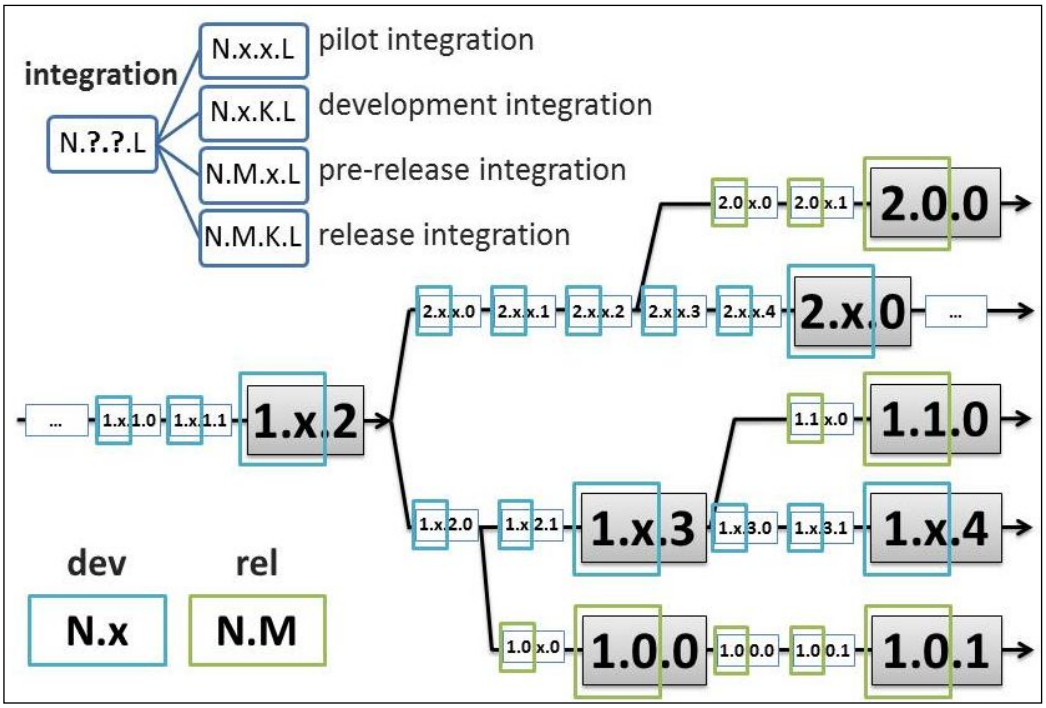
Picture 6

- ***Artifact version*** – *Version* of artifact (result of the build process) produced as a result of one of *SDLC stages*. Has one of the following *version templates*: `N.?.K` or `N.?.?.L`.
  - ***Build/release artifact version*** – *artifact* having following *version template*: `N.?.K`.
    - ***Build artifact version*** – version of *build artifact* corresponding to following *version template*: `N.x.K`
    - ***Release artifact version*** – version of *release artifact* corresponding to following *version template*: `N.M.K`



Picture 7

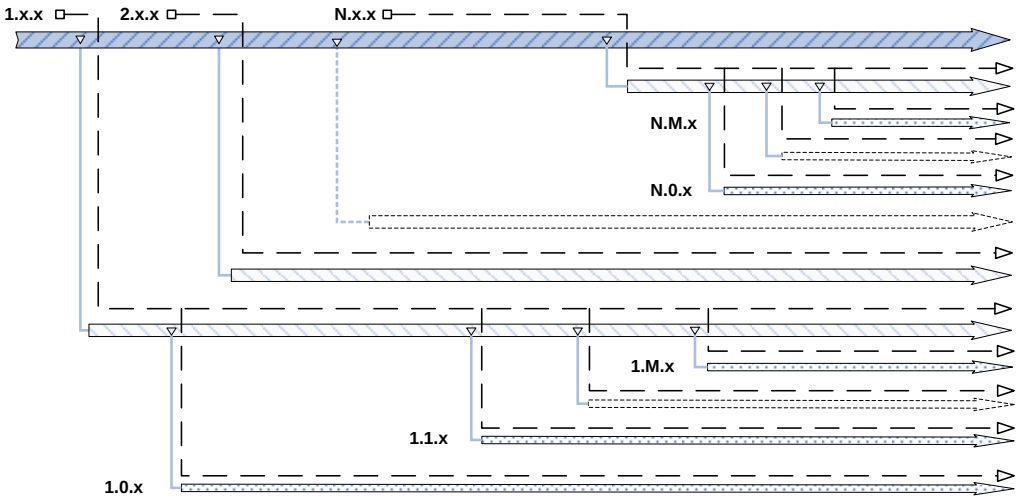
- ***Integration artifact version*** – version of *integration artifact* corresponding to following *version template*: `N.?.?.L`.
  - ***Pilot integration artifact version*** – version of *integration artifact* corresponding to following *version template*: `N.x.x.L`.
  - ***Development integration artifact version*** – version of *integration artifact* corresponding to following *version template*: `N.x.K.L`.
  - ***Pre-release integration artifact version*** – version of *integration artifact* corresponding to following *version template*: `N.M.x.L`.
  - ***Release integration artifact version*** – version of *integration artifact* corresponding to following *version template*: `N.M.K.L`.



Picture 8

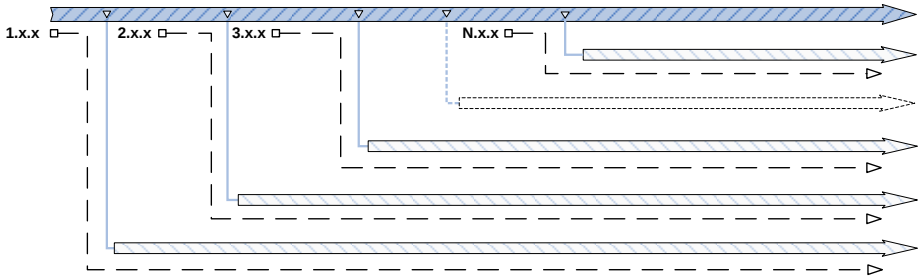
- ***Branch version*** – *version* of repository branch. Could be of two main types: *support branch version* and *release branch version*. Has following *version template*: `N.?.x`.
  - ***Support branch version*** – branch (*support branch*) having following *version template*: `N.x.x`.

        - ***Release branch version*** – branch (*release branch*) having following *version template*: `N.M.x`.
    - ***Starting version*** – *version* used by default for the *trunk* codebase of newly started project/development (new version control repository has been allocated).
    - ***Inherited version*** – *version* part, which has been inherited from the parent entity. For example, *release branch version* `1.0.x` inherits its *major version number* from parent *support branch* `1.x.x` as long it cannot be created without this parent branch. Another example: *release artifact* `2.3.6` inherits *major version number* (2) and *release version number* (3) from parent *release branch* `2.3.x`.
    - ***Imaginary version*** – *version* assigned to the *trunk* codebase depending on the repository state or current development phase (see Picture 13, *imaginary versions* are outlined by black dashed rectangles). *Codebase inheritance* definition section contains corresponding details.
    - ***Version template*** – notation used for referencing version sets (several *versions* at once). For example, versions `1.x.x`, `3.x.x` and `1.0.x` can be referenced at once using *version template* `N.?.x`.
- ***Artifact*** – result of build process which can be used for subsequent installation/*deployment*. There could be three main *artifact* types: *build artifact, release artifact* and *integration artifact.*
    - ***Build artifact*** – *artifact* produced during *precursory major version development* stage.
    - ***Release artifact*** – *artifact* produced during *release version development* stage.
    - ***Integration artifact*** – *artifact* produced during *integration* stage.
- ***Versioned artifact*** – entity stored in version control system (there are two types of such entities: *branches* and *tags*) which can be used as a starting point for build process and, therefore, for producing *artifact*. *Versioned artifact* has the same *version* as the corresponding *artifact* it represents.
    - ***Branch*** – *versioned artifact* used for the purpose of versioning project source codebase.
        - ***Trunk*** – main *branch* used for versioning source codebase related to the latest *major version development* stage.
        - ***Support branch*** – *branch* used for versioning source codebase related to the *major version development* stage.
        - ***Release branch*** – *branch* used for versioning source codebase related to *release version development* stage.
        - ***Experimental branch*** – *branch* used for versioning any kind of codebase without regard to any of the *SDLC stages*.
    - ***Tag*** – *versioned artifact* used for the purpose of snapshotting source codebase used for producing *build/release artifact*.
- ***Version inheritance scope*** – specific stream interval (or conjunction of subsequent stream intervals) used as a parent entity for creating child entities and providing corresponding *inherited version* for child entities. *Version inheritance scope* is closely related to corresponding *SDLC stage*.
    - ***Major inheritance scope*** – *version inheritance scope* based on *codebase inheritance* concept. It consists of stream intervals involved into *major version development* including *release branches* (see Picture 9, *major inheritance scope* is marked with black dashed line):



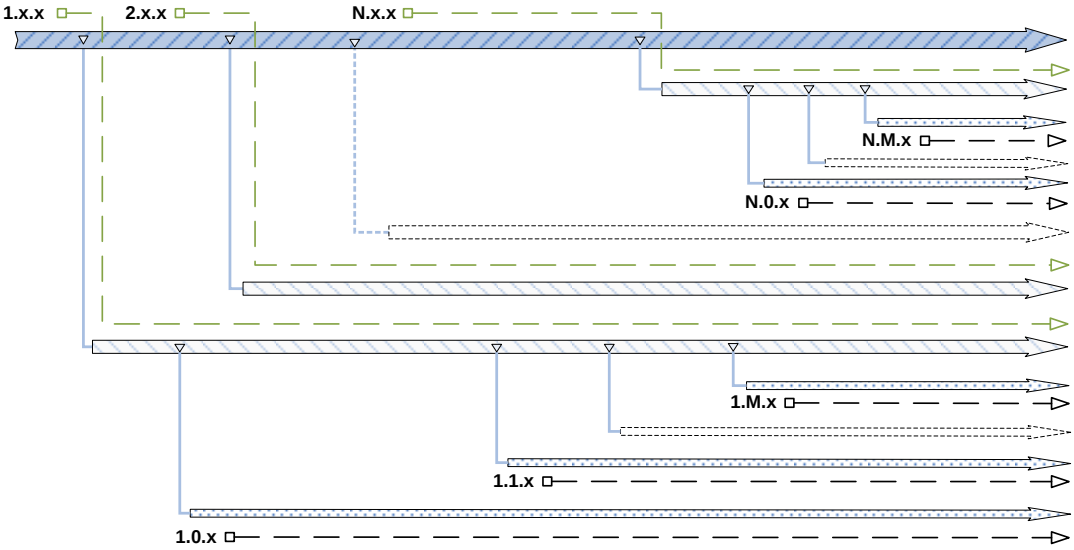
Picture 9

- ***Precursory inheritance scope*** – *version inheritance scope,* which is based on *codebase inheritance* concept. It consists of stream intervals involved into *major version development* used for producing *build artifacts* (see Picture 10, *precursory inheritance scope* is marked with black dashed line):



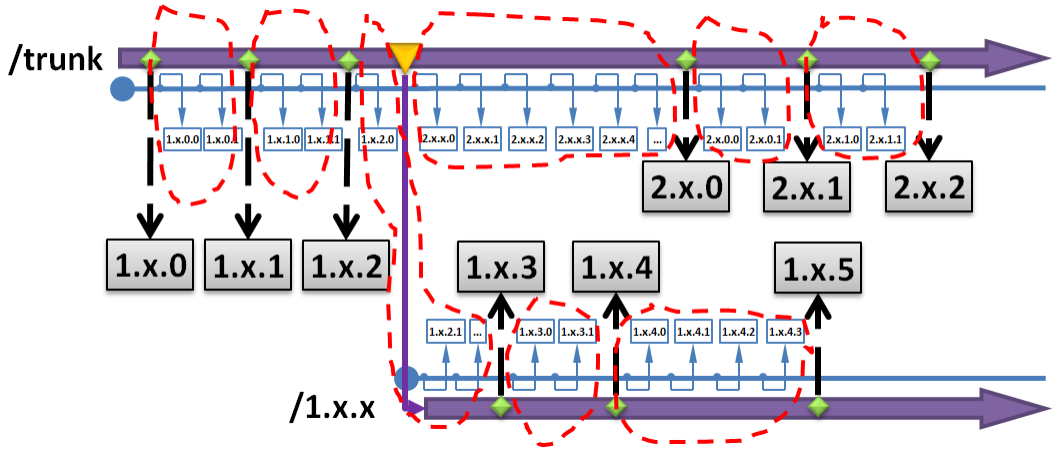
Picture 10

- ***Release inheritance scope*** – *version inheritance scope* consisting of stream interval involved into *release version development* used for producing *release artifacts*. Unlike *major inheritance scope* or *precursory inheritance scope*, stream interval for r*elease inheritance scope* corresponds to one whole *release branch* (*major and precursory inheritance scope* correspond to several subsequent stream intervals and different branches; see Picture 11, *major inheritance scope* is marked with black dashed line).



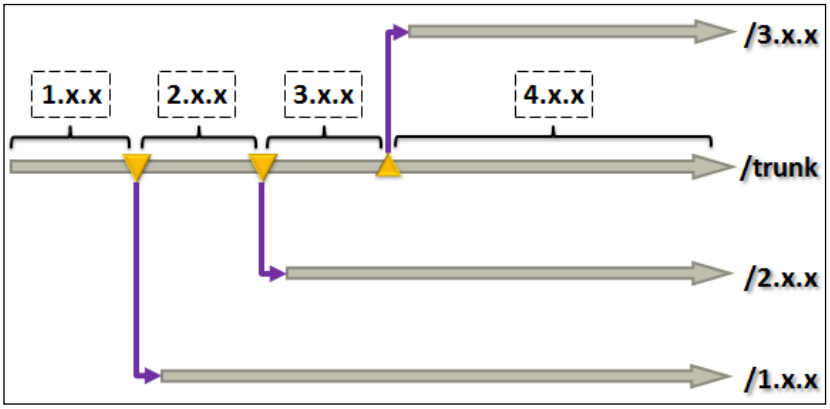
Picture 11

- ***Integration inheritance scope*** – *version inheritance scope* of the stage lasting from the moment of previous *build/release artifact* delivery to the next *build/release artifact* delivery (see Picture 12, *integration inheritance scopes* are marked with dashed red outlines):



Picture 12

- ***Codebase*** – all source code corresponding to specific versioned state of a single *branch*. In other words, *codebase* corresponds to all content that has been stored in a *branch* at some specific moment of time.
- ***Codebase inheritance*** is a concept founded on the principle of versioning latest *major version development* in *trunk*. Once next *major version development* has been initiated, versioning of previous *major version development* should be transferred into separate *support branch* (see Picture 13):



Picture 13

- ***SDLC*** – software development lifecycle. Consists of several *SDLC phases*.
  - ***SDLC phase*** – set of software development activities grouped by one of following general standalone goals to achieve: project initiation, requirements analysis and development, design, implementation (development), integration, testing, deployment (delivery), maintenance and support, system utilization.
    - ***Deployment (delivery) –*** *SDLC phase* including publishing of *build artifacts* to the target environment and configuring it properly in order to run seamlessly on the target environment.
  - ***SDLC iteration*** – set of subsequent *SDLC phases* which can be represented as a cycle (can be repeatedly passed in an established order one or several times). There could be several *SDLC iterations* nested into each other (**TODO:** needs picture).
  - ***SDLC stage*** – stage of software development process producing set artifacts referenced by specific *version template* (has at least one **?** symbol as a part of *version template*). Main feature of *SDLC stage* is that it has one whole consistent *version inheritance scope*.
    - ***Major version development*** – *SDLC stage* which can be referenced by following *version template*: `N.?.?`
      - ***Precursory major version development*** – *SDLC stage* which can be referenced by following *version template*: `N.x.?`
      - ***Release version development*** – *SDLC stage* which can be referenced by following *version template*: `N.M.?`
      - ***Integration*** – *SDLC stage* which can be referenced by following *version template*: `N.?.?.L`
- ***Merge*** – process of integrating two (or more) different *branches codebases* with the purpose of producing consistent resulting *codebase* (representing properly functioning integrated functionality).

- ***Initial repository structure*** **–** default repository directories hierarchy which should be used for the purpose of *versioned project* initiation:

```
/trunk
/tags
      /builds
      /PA
      /A
      /B
      /releases
            /AR
            /BR
            /RC
            /ST
/branches
      /experimental
      /support
      /release
```

- ***Versioned project*** – software/system/module/solution stored in a separate repository having independent versioning stream (independent revisions numbering). Every versioned project is started using *initial repository structure*.

3. Major version number incrementing rules

When development is started in *trunk*, it is supposed that m*ajor version number* starts from 0 (unless another *major version number* has been specified explicitly). *Version* `1.x.x` can be used instead of `0.x.x` as a *starting version* in case when at least three statements from the following list hold false:

1. Design phase is completely omitted (there is no need of design, all requirements are easy to implement, there is no explicitly available design specifications/requirements for the software/system/module/solution being developed).
2. There is no framework involved into the application/system/module/solution development.
3. Development of application/system/module/solution has been started as a result of existing software/system/module/solution decomposition or architectural rework.
4. There is no basic idea/concept which can be considered to be the basis of application/system/module/solution being developed.
5. Development has been started in the form of research & development or experimental project.
6. It cannot be considered to be an official project (no defined scope and project goals, no allocated cost and resources).

Major version number should be incremented when at least one of the following conditions is met:

1. New design specification has been created for the project; it is going to be reworked according to the developed specification. This item covers following cases:
    a. Rewriting *versioned project* from scratch.
    b. Deep refactoring.
    c. Deep project directories restructuring.
    d. Introducing changes which will complexify merging with sibling *branches* (*release branches* and *experimental branches*).
2. Framework is going to be introduced into the existing architectural project solution; it is going to be reworked accordingly.
3. Project solution has been decomposed into several independent modules (having *starting version* `0.x.x`) and can be considered to be the common part (or integration solution) for newly created independent modules.
4. Basic underlying project idea/concept has been changed; project is going to be reworked accordingly.
5. Project gets an official status (resources and costs have been allocated, scope and project goals have been defined). This item also covers the case of changing project development status from R&D/experimental to officially supported.

4. Release version number incrementing rules

Release version number should be incremented when at least one of the following conditions is met:

1. When there is need of promoting artifacts to target environment (for example, production environment, UAT environment or staging environment) for the purpose of making it available for the end user.
2. If list of logically similar requirements has been implemented and it needs to be tested (for example, backlogs approach is used) and made available to the end user.
3. When release deadline is coming. In this case release version number should be incremented in advance (at least ~2 weeks) in order to perform all planned release activities (alpha-testing, beta-testing, deployment, etc).

5. Build version number incrementing rules

Build version number should be incremented when at least one of the following conditions is met:

1. When there is need of producing *artifact* ready for testing.
2. One or more requirements have been implemented and should be included into the next application *deployment/delivery*.
3. When there is need of deploying created *artifact* to any of the target platforms.

6. Integration version number incrementing rules

Integration version number is incremented automatically by continuous integration server using *integration inheritance scope*.

7. Maturity levels

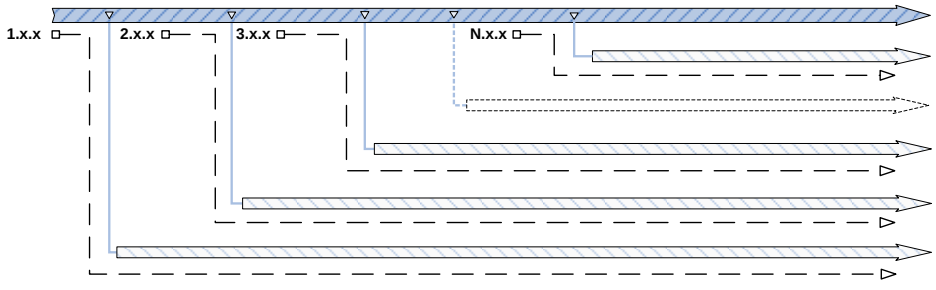
There should be following maturity levels introduced:

- *Build artifact* maturity levels:
  - ***PA*** (pre-alpha) – maturity level of *build artifact* showing that developers use it for internal needs (smoke-testing, basic verification, etc)
  - ***A*** (alpha) – maturity level of *build artifact* showing that it is used by software testing department in order to provide detailed *build artifact* verification report.
  - ***B*** (beta) – maturity level of *build artifact* showing that it can be used for delivery to end-user/customer in order to provide early acceptance testing or end-user verification.
- *Release artifact* maturity levels:
  - ***AR*** (alpha-release) – maturity level of *release artifact* showing that it is used by software testing department in order to provide detailed *release artifact* verification report.
  - ***BR*** (beta-release) – maturity level of *release artifact* showing that it is used for delivery to end-user/customer in order to provide acceptance testing or end-user *release artifact* verification.
  - ***RC*** (release-candidate) – maturity level of *release artifact* showing that it needs some time (it should be fixed release-candidate interval; 1 month, for example) to function in production environment in order to detect critical/major bugs. If any critical/major bugs were found, release-candidate interval counting should start again.
  - ***ST*** (stable) – maturity level of *release artifact* showing that release-candidate interval was successfully passed after last critical/major bug was found during release-candidate phase.

8. Version numbering approach

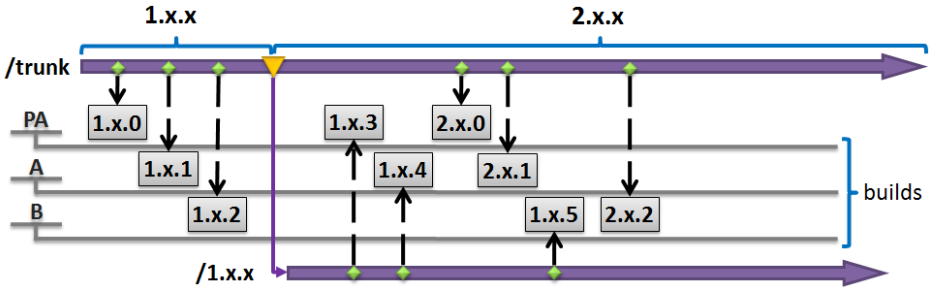
1. Any *versioned project* development should be started with *initial repository structure* creation. List of reasons for *versioned project* initiation is following:
   a. Moving existing project source code into the repository for the purpose of introducing *SCMF* conventions and practices.
   b. *Versioned project* has been started as a result of existing *versioned project* decomposition into several independent modules.
   c. *Versioned project* has been started in order to track implementation of some basic underlying idea/concept.
   d. *Versioned project* has been started as an R&D/experimental project without any underlying idea/concept, but with some goals and vision.

2. After version control repository structure has been initiated, initial development should start in *trunk*. Once the development has been started, t*runk* codebase will be referenced by *imaginary version* with accordance to Major version number incrementing rules.
3. Once the development *imaginary version* has been properly assigned, it is possible to:
    a. Continue *major version development* in *trunk*.
    b. Create *support branches* using *trunk codebase*. *Support branches* can be created only if corresponding decision about incrementing *major version number* has been made (see 3). Once it was decided to increment *major version number* **N** for the ongoing development, there should be following actions performed:
        i. Corresponding *support branch* should be created in `/branches/support/` repository directory (`/branches/support/N.x.x`). All development corresponding to the previous *major version development* `N.x.x` will be maintained in the newly created branch.
        ii. *Imaginary version* for the project *codebase* maintained in *trunk* should be updated to `N+1.x.x` as long as *trunk* will be used for the next *major version development codebase* maintenance.
        iii. Next *major version development* (**N+1**) should be started in *trunk* while previous *major version development* (**N**) should be continued to be maintained using *support branch* `N.x.x`.



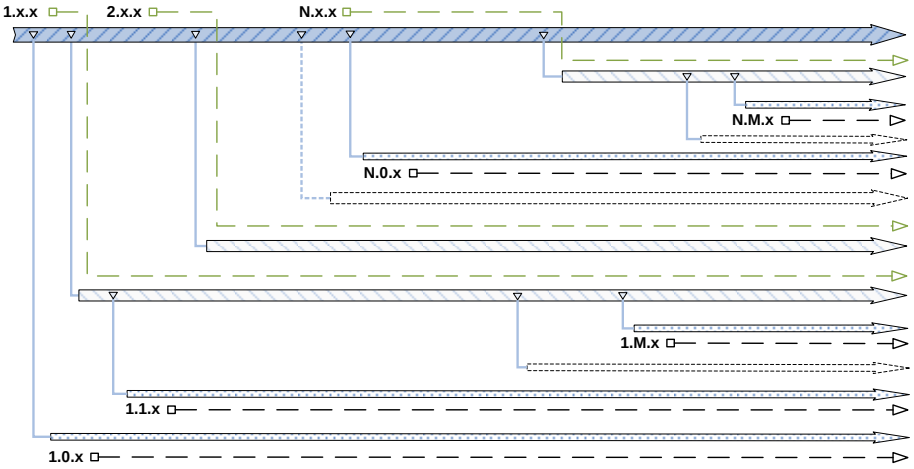
Picture 14

    c. Create *build tags* using either *trunk codebase* or *support branch codebase* (corresponding *precursory inheritance scope* and *build artifact maturity level* **BML** should be used) for the purpose of creating *build artifact*. *Build tag* can be created only if corresponding decision about incrementing *build version number* has been made (see 5). Once it was decided to increment *build version number* **K** for the ongoing development, there should be following actions performed:
        i. Corresponding *build tag* should be created in one of the subdirectories of `/tags/builds/`repository directory(`/tags/builds/`**BML**`/N.x.K`)
        ii. Build process for *build artifact* referenced by version `N.x.K` should be triggered (either manually or using CI-server).



Picture 15

    d. Create *release branches* using either *trunk codebase* or *support branch codebase* (corresponding *major inheritance scope* should be used). *Release branches* can be created only if corresponding decision about incrementing *release version number* has been made (see 4). Once it was decided to increment *release version number* **M** for the ongoing development, there should be following actions performed:
        i. Corresponding *release branch* should be created in `/branches/release/` repository directory (`/branches/release/N.M.x`).
        ii. All development corresponding to the *release version development* `N.M.x` will be maintained in the newly created branch.

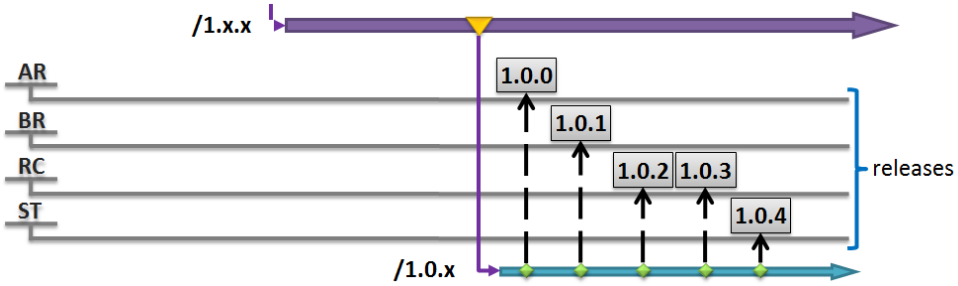


Picture 16

e. Create *release tags* using *release branch codebase* (*corresponding version inheritance scope* and *release artifact maturity level* **RML** should be used). *Release tag* can be created only if corresponding decision about incrementing *build version number* has been made (see 5). Once it was decided to increment *build version number* **K** for the ongoing development, there should be following actions performed:

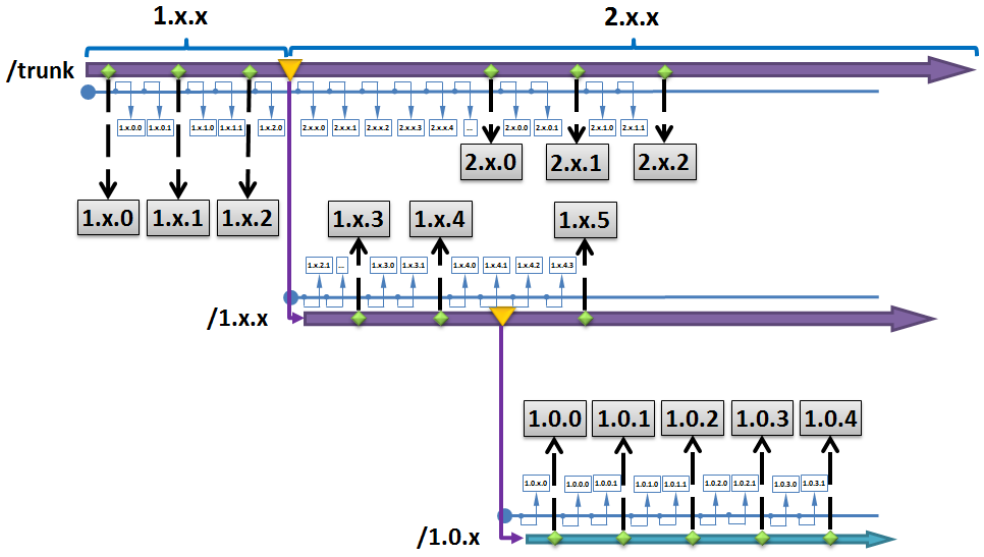
   i. Corresponding *release tag* should be created in one of the subdirectories of `/tags/releases/`repository directory(`/tags/releases/`**RML**`/N.M.K`)

   ii. Build process for *release artifact* referenced by version `N.M.K` should be triggered (either manually or by CI-server) .



Picture 17

f. Produce *integration artifacts* using *codebase* of *trunk*, *release branches* or *support branches*. CI-server should be properly configured in order to automatically trigger build process for producing *integration artifact* once changes are committed into *trunk*, *release branches* or *support branches*. CI-server should use *integration inheritance scope* in order to assign proper *version* for produced *integration artifact:*



Picture 18

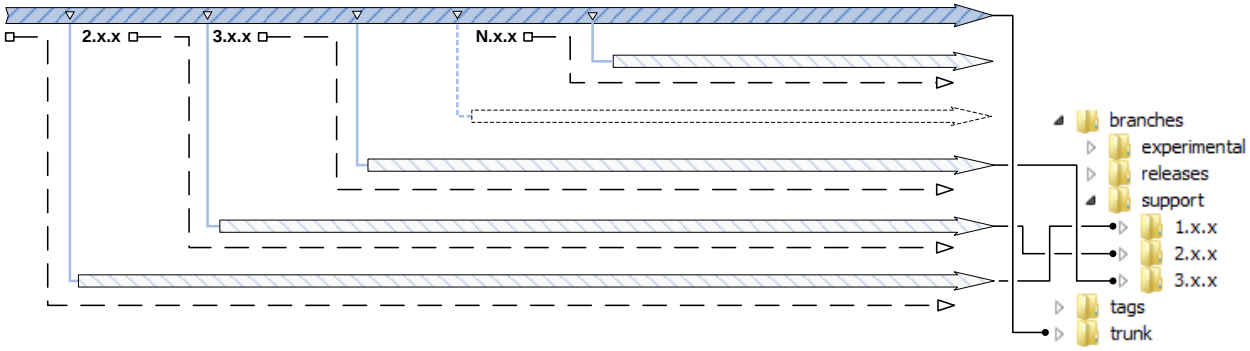
g. Create *experimental branches* using any *codebase*.

9. Build/deployment process customization

1. *Artifacts* should be published into the dependency management repository only as a result of *release version development* (preferably with ***RC*** or ***ST*** maturity levels).
2. Based on specified *maturity levels*, *artifacts* should be preferably deployed to following locations:
    a. ***PA*** – local development server.
    b. ***A*** – local testing server.
    c. ***B*** – remote testing server.
    d. ***AR*** – local testing server (production environment imitation, maybe UAT).
    e. ***BR*** – remote testing server (UAT or production).
    f. ***RC*** – production server.
    g. ***ST*** – production server.
3. Build process performed during *precursory version development* (*release artifacts* and *integration artifacts, **PA**, **A**, **B** maturity levels*) should possibly include following steps:
    a. Compilation
    b. Deployment
    c. Running unit-tests
    d. Initial data loading
4. Build process performed during *release version development* (*release artifacts* and *integration artifacts, **AR**, **BR**, **RC**, **ST** maturity levels*) should possibly include following steps:
    a. Performing static analysis
    b. Compilation
    c. Deployment
    d. Initial data loading
    e. Running unit-tests
    f. Running security tests
    g. Running integrity checks
    h. Running performance tests
    i. Performing dynamic analysis
    j. Documentation generation
    a. Gathering metrics

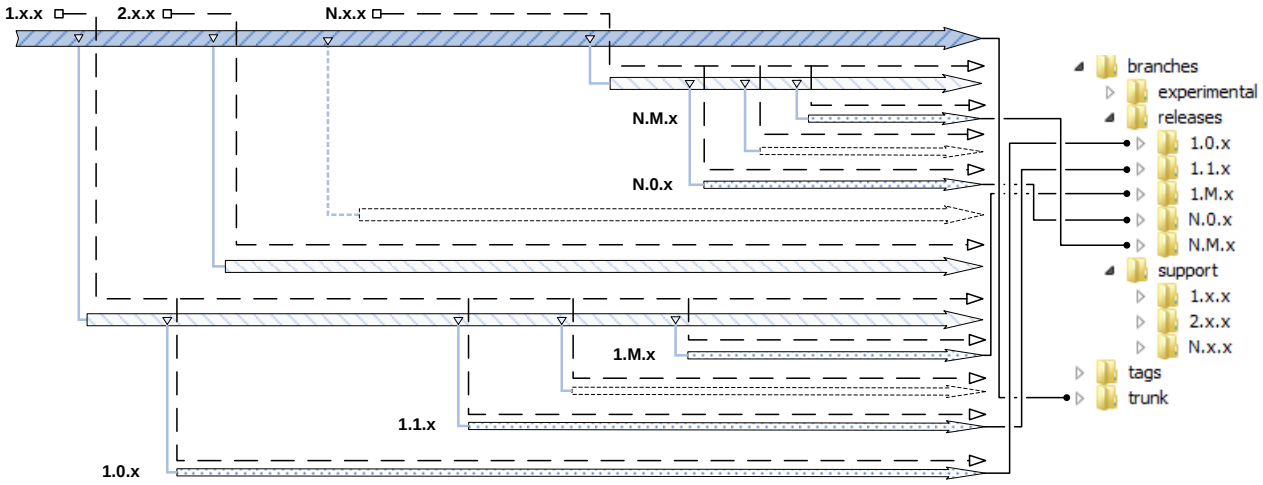
10. Branches inheritance rules

Support branch inheritance rules comply with the *codebase inheritance* rules. Corresponding *versioned artifacts* (`branches/support/N.x.x`) are to be created in version control repository:



Picture 19

Release branch inheritance rules comply with the *codebase inheritance* rules using *major version development scope*. Corresponding *versioned artifacts* (`branches/releases/N.M.x`) are to be created in version control repository:



Picture 20

There are certain branch creation and inheritance restrictions which will be described by filling in table shown below:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk				
	support				
	release				
	experimental				

Figure 1

This table and all following tables represent situations when specific branch types could or could not be inherited from each other. Inheritance rules set which are currently being examined correspond to the cells marked with yellow color. Other cells have corresponding content or color (grey color correspond to logically excluded or impossible cases).

1. *Trunk* cannot be the child of any other branch:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk				
	support				
	release				
	experimental				

Figure 2

2. *Support branch* can be inherited only from *trunk*:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+		
	support		-		
	release		-		
	experimental		-		

Figure 3

3. *Release branches* can be inherited both from *trunk* and *support branches*:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+	+	
	support		-	+	
	release		-	-	
	experimental		-	-	

Figure 4

4. *Experimental branches* can be inherited from any type of branch except *release branch*:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+	+	+
	support		-	+	+
	release		-	-	-
	experimental		-	-	+

Figure 5

5. Resulting branch inheritance rules table will look as follows:

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+	+	+
	support		-	+	+
	release		-	-	-
	experimental		-	-	+

Figure 6

11. Merging approach

11.1. Branch types (merging perspective)

From merging perspective, there are following features of branches:

- *Trunk* – branch containing latest *major version development. S*ource code from other *codebases* tends to be merged into *trunk* in order to stabilize mainline *codebase*. *Trunk* contains changes which can be by all means considered to be the latest development (latest ideas and its implementation which might be presented even though it might be incomplete).
- *Support branch* created as a result of potential complication of merging with sibling branches (*release branches* and *experimental branches*) during parent *codebase* maintenance. It means that decision about support branch creation is made at the moment when it becomes obvious that subsequent development will be incompatible for merging with previously supported *codebases*. Other criteria for *support branch* creation are described in major version number incrementing rules section (see 3).
- *Release branch* created for the purpose of bugfixing and subsequent merging of fixed bugs into the parent branch. Criteria for *release branch* creation are described in release version number incrementing rules section (see 4).
- *Experimental branch* created for the purpose of experimental development which could be easily merged into the parent branch and back. There are no restrictions applied to experimental branches except that they cannot be created using *release branch codebase*.

11.2. Merging rules diagram

Following table represents cases when merge is possible between *branches* of defined types and when it is not possible. There are certain merging restrictions which will be described by filling in table shown below:

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child							
	release	parent							
		child							
	experimental	parent							
		child							

Merging rules set which are currently being examined correspond to the cells marked with yellow color. Other cells have corresponding content (+ or -) or color (grey color correspond to logically excluded or impossible cases, dark grey color correspond to cases excluded by branch inheritance rules, see 10):

	Logically excluded cases of merging
	Cases excluded by branch inheritance and creation rules
+	Allowed case of merging
-	Disallowed case of merging operation

1. Trunk can be only parent branch:

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child							
	release	parent							
		child							
	experimental	parent							
		child							

2. Merging from parent to parent of the same *branch* seems to be merged into itself. It makes sense only as a conflict which is not the case of merge we're considering:

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child							
	release	parent							
		child							
	experimental	parent							
		child							

3. *Branch* cannot be parent to itself:

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child							
	release	parent							
		child							
	experimental	parent							
		child							

4. Cases automatically excluded by branches inheritance rules (see 10):

RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+	+	+
	support		-	+	+
	release		-	-	-
	experimental		-	-	+

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent			-				
		child		-	-				
	release	parent					-		
		child				-	-		
	experimental	parent							+
		child						+	+

5. Cases automatically excluded by branch creation and inheritance rules:

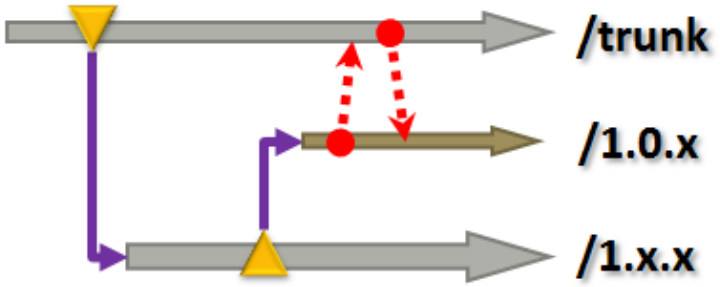
RELATION		child			
	BRANCH TYPE	trunk	support	release	experimental
parent	trunk		+	+	+
	support		-	+	+
	release		-	-	-
	experimental		-	-	+

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child				-		-	
	release	parent			-				-
		child						-	
	experimental	parent			-		-		
		child				-			

6. The is the result of exclusion merging cases by logic and branches inheritance rules (see 10) looks as follows:

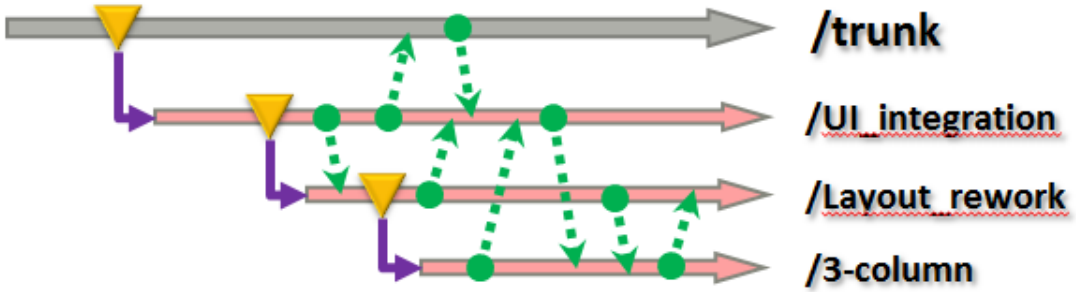
DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent							
	support	parent							
		child							
	release	parent							
		child							
	experimental	parent							
		child							

7. Child to child relation represents sibling *branches*:



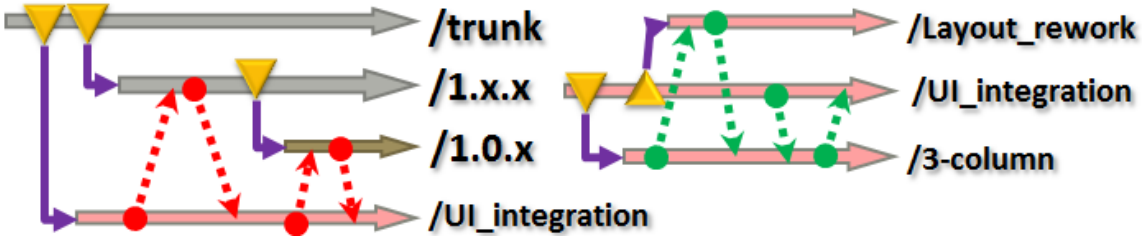
DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent			-		-		
	support	parent					-		
		child	-				-		
	release	parent							
		child	+	+	-				
	experimental	parent							
		child							

8. It is allowed to merge *experimental branches* with any type of parent *branch*:



DESTINATION		to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent			-		-		+
	support	parent					-		+
		child	-				-		
	release	parent							
		child	+	+	-				
	experimental	parent							+
		child	+	+				+	

9. No merge allowed between sibling *experimental* and any type of other branch (except *experimental*):



DESTINATION		to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent			-		-		+
	support	parent					-		+
		child	-				-		-
	release	parent							
		child	+	+	-				-
	experimental	parent							+
		child	+	+	-		-	+	+

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent			-		-		+
	support	parent					-		+
		child	-				-		-
	release	parent							
		child	+	+	-				-
	experimental	parent							+
		child	+	+	-		-	+	+

Figure 7

11.3. Merging approach effectiveness

49	Total number of cells
33	Number of cells with logically excluded cases
20	Number of cells with cases excluded logically and using branch inheritance rules
9	Allowed case of merging
73%	Disallowed cases introduced by merging rules

DESTINATION			to						
	BRANCH TYPE		*trunk*	*support*		*release*		*experimental*	
		RELATION	parent	parent	child	parent	child	parent	child
from	*trunk*	parent			-		-		+
	support	parent					-		+
		child	-				-		-
	release	parent							
		child	+	+	-				-
	experimental	parent							+
		child	+	+	-		-	+	+

Version numbering practice together with merge restriction rules lead to reducing number of possible merges from 33 to 9, which makes it to by**73%**growth in reducing merge overhead in comparison with intuitive approach to branches creation and merging.