****Predicate 1:** spl(IdSPL)**

**IdSPL identifies the software product line.**

****Predicate 2:** domainRequirement(IdDomainRequirement)**

**IdDomainRequirement identifies the domain requirement.**

**Predicate 3:** isPartOf(R,SPL)

It evaluates if there is a domain requirement **R** that is part of a software product line **SPL**. For example, **isPartOf(R, SPL)** returns true if **SPL** contains **R**.

Formalization: ∀R, ∃SPL. *domainRequirement*(R) ∧ *spl*(SPL) ∧ *isPartOf*(R,SPL)

****Predicate 4:** domainComponent(IdDomainComponent)**

**IdDomainComponent identifies a domain reusable component.**

****Predicate 5:** domainFile(IdDomainFile, Filename, Code, Destination)**

**IdDomainFile identifies the file; Filename contains the filename; Code contains the file code; Destination contains the path in which the file is deployed in the derivation process.**

****Predicate 6**:** isPartOf(F,C)

It evaluates if there is a domain file **F** that is part of a domain component **C**. For example, **isPartOf(F, C)** returns true if **C** contains **F**.

Formalization: ∀F, ∃C. *domainFile*(F) ∧ *domainComponent*(C) ∧ *isPartOf*(F,C)

****Predicate 7:** fragmentationPoint(PointId)**

**PointID identifies the fragmentation point.**

**Predicate 8:** isPartOf(FP,F)

It evaluates if there is a fragmentation point **FP** that is part of a domain file **F**. For example, **isPartOf(FP, F)** returns true if **F** contains **FP**.

Formalization: ∀FP, ∃F. *fragmentationPoint*(FP) ∧ *domainFile*(F) ∧ *isPartOf*(FP,F)

****Predicate 9:** fragment(IdFragment, Filename, Code)**

**IdFragment identifies the fragment; Filename contains the fragment filename; Code contains the fragment code.**

****Predicate 10:** fragmentTypeAdd(IdFragment)**

**IdFragment references a fragment type add (a fragment that injects code).**

****Predicate 11:** fragmentTypeReplace(IdFragment)**

**IdFragment references a fragment type replace (a fragment that replaces code).**

****Predicate 12:** fragmentTypeHide(IdFragment)**

**IdFragment references a fragment type hide (a fragment that hides code).**

**Predicate 13:** isPartOf(FR,C)

It evaluates if there is a fragment **FR** that is part of a domain component **C**. For example, **isPartOf(FR, C)** returns true if **C** contains **FR**.

Formalization: ∀FR, ∃C. *fragment*(FR) ∧ *domainComponent*(C) ∧ *isPartOf*(FR,C)

**Predicate 14:** isAssociatedWith(FP,FR)

It evaluates if there is a fragmentation point **FP** that is associated with a fragment **FR**. For example, **isAssociatedWith(FP, FR)** returns true if **FP** is associated with **FR**.

Formalization: ∀FP, ∃FR. *fragmentationPoint*(FP) ∧ *fragment*(FR) ∧ *isAssociatedWith*(FP,FR)

**Predicate 15:** isAssociatedWith(F,FR)

It evaluates if there is a domain file **F** that is associated with a fragment **FR**. For example, **isAssociatedWith(F, FR)** returns true if **F** is associated with **FR**.

Formalization: ∀F, ∃FR. *domainFile*(F) ∧ *fragment*(FR) ∧ *isAssociatedWith*(F,FR)

****Predicate 16:** applicationFile(IdApplicationFile, Filename, Code, Destination)**

**IdApplicationFile identifies the file; Filename contains the filename; Code contains the file code; Destination contains the path in which the file is deployed in the derivation process.**

**Formula 1:** executeAddFragment(FR)

It evaluates if the fragment **FR** can inject code.

Formalization: ∀FR, ∃FP, ∃F. *fragment*(FR) ∧ *fragmentTypeAdd*(FR) ∧ *fragmentationPoint*(FP) ∧ *applicationFile*(F) ∧ *isAssociatedWith*(FP,FR) ∧ *isPartOf*(FP,F) ⇒ *add*(FR,FP,F)

SWI-Prolog:

|  |
| --- |
| executeAddFragment(FR) :- forall(  (fragment(FR,FRNAME,FRCODE),  fragmentTypeAdd(FR),  fragmentationPoint(FP),  applicationFile(F,FNAME,FCODE,FDEST),  isAssociatedWith(FP,FR),  isPartOf(FP,F)  ),  (add(FRCODE,FP,F,FCODE))  ). |

**Function 1:** add(FR,FP,F)

It adds the fragment **FR** code after the fragmentation point **FP** inside the application file **F**.

SWI-Prolog:

|  |
| --- |
| add(FRCODE,FP,F,FCODE) :-  retract(applicationFile(F,FNAME,FCODE,FDEST)),  term\_string(FP,FPS),  sub\_string(FCODE, Before, L, After, FPS),  sub\_string(FCODE, \_, After, 0, Value2),  Initial is Before+L,  sub\_string(FCODE, 0, (Initial), \_, Value1),  string\_concat(Value1,FRCODE,New1),  string\_concat(New1,Value2,New2),  assert(applicationFile(F,FNAME,New2,FDEST)). |

**Formula 2:** executeReplaceFragment(FR)

It evaluates if the fragment **FR** can replace code.

Formalization: ∀FR, ∃FP, ∃F. *fragment*(FR) ∧ *fragmentTypeReplace*(FR) ∧ *fragmentationPoint*(FP) ∧ *applicationFile*(F) ∧ *isAssociatedWith*(FP,FR) ∧ *isPartOf*(FP,F) ⇒ *replace*(FR,FP,F)

SWI-Prolog:

|  |
| --- |
| executeReplaceFragment(FR) :- forall(  (fragment(FR,FRNAME,FRCODE),  fragmentTypeReplace(FR),  fragmentationPoint(FP),  applicationfile(F,FNAME,FCODE,FDEST),  isAssociatedWith(FP,FR),  isPartOf(FP,F)  ),  (replace(FRCODE,FP,F,FCODE))  ). |

**Function 2:** replace(FR,FP,F)

It replaces the application file **F** code surrounded by the fragmentation point **FP** with the fragment **FR** code.

SWI-Prolog:

|  |
| --- |
| replace(FRCODE,FP,F,FCODE) :-  retract(applicationFile(F,FNAME,FCODE,FDEST)),  term\_string(FP,FPS),  sub\_string(FCODE, Before, L, After, FPS),  sub\_string(FCODE, \_, After, 0, Value2),  Initial is Before+L,  sub\_string(FCODE, 0, Initial, \_, Value1),  string\_concat(Value1,FRCODE,New1),  sub\_string(Value2, Before2, L2, After2, FPS),  Initial2 is After2+L,  sub\_string(FCODE, \_, Initial2, 0, Value3),  string\_concat(New1,Value3,New2),  assert(applicationFile(F,FNAME,New2,FDEST)). |

**Formula 3:** executeHideFragment(FR)

It evaluates if the fragment **FR** can hide code.

Formalization: ∀FR, ∃FP, ∃F. *fragment*(FR) ∧ *fragmentTypeHide*(FR) ∧ *fragmentationPoint*(FP) ∧ *applicationFile*(F) ∧ *isAssociatedWith*(FP,FR) ∧ *isPartOf*(FP,F) ⇒ *hide*(FP,F)

SWI-Prolog:

|  |
| --- |
| executeHideFragment(FR) :- forall(  (fragment(FR,FNAME,FRCODE),  fragmentTypeHide(FR),  fragmentationPoint(FP),  applicationFile(F,FNAME,FCODE,FDEST),  isAssociatedWith(FP,FR),  isPartOf(FP,F)  ),  (hide(FP,F,FCODE))  ). |

**Function 3:** hide(FP,F)

It hides the file **F** code surrounded by the fragmentation point **FP**.

SWI-Prolog:

|  |
| --- |
| hide(FP,F,FCODE) :-  retract(applicationFile(F,FNAME,FCODE,FDEST))),  term\_string(FP,FPS),  sub\_string(FCODE, Before, L, After, FPS),  sub\_string(FCODE, \_, After, 0, Value2),  Initial is Before+L,  sub\_string(FCODE, 0, Initial, \_, Value1),  string\_concat(Value1,"/\*",New1),  sub\_string(Value2, Before2, L2, After2, FPS),  sub\_string(Value2, 0, Before2, \_, Value4),  string\_concat(New1,Value4,New2),  string\_concat(New2,"\*/",New3),  Initial2 is After2+L,  sub\_string(FCODE, \_, Initial2, 0, Value3),  string\_concat(New3,Value3,New4),  assert(applicationFile(F,FNAME,New4,FDEST)). |

****Predicate 17:** customizationPoint(PointId)**

**PointID identifies the customization point.**

**Predicate 18:** isPartOf(CP,F)

It evaluates if there is a customization point **CP** that is part of a file **F**. For example, **isPartOf(CP, F)** returns true if **F** contains **CP**.

Formalization: ∀CP, ∃F. *customizationPoint*(CP) ∧ *file*(F) ∧ *isPartOf*(CP,F)

****Predicate 19:** customizationfile(IdCustomizationFile)**

**IdCustomizationFile identifies the customization file.**

**Predicate 20:** isPartOf(CF,C)

It evaluates if there is a customization file **CF** that is part of a domain component **C**. For example, **isPartOf(CF, C)** returns true if **C** contains **CF**.

Formalization: ∀CF, ∃C. *customizationFile*(CF) ∧ *component*(C) ∧ *isPartOf*(CF,C)

**Predicate 21:** isAssociatedWith(CP,CF)

It evaluates if there is a customization point **CP** that is associated with a customization file **CF**. For example, **isAssociatedWith(CP, CF)** returns true if **CP** is associated with **CF**.

Formalization: ∀CP, ∃CF. *customizationPoint*(CP) ∧ *customizationFile*(CF) ∧ *isAssociatedWith*(CP,CF)

**Formula 4:** executeCustomization(CF)

It evaluates if the customization file **CF** can execute customizations.

Formalization: ∀CF, ∃CP, ∃F. *customizationFile*(CF) ∧ *customizationPoint*(CP) ∧ *applicationFile*(F) ∧ *isAssociatedWith*(CP,CF) ∧ *isPartOf*(CP,F) ⇒ *customize*(CP,F)

SWI-Prolog:

|  |
| --- |
| executeCustomization(CF) :- forall(  (customizationFile(CF),  customizationPoint(CP),  applicationFile(F,FNAME,FCODE,FDEST),  isAssociatedWith(CP,CF),  isPartOf(CP,F)  ),  customize(CP,F,FCODE)  ). |

**Function 4:** customize(CP,F)

It customizes the application file **F** code, replacing the surrounded by the customization point **CP** with a new customized code provided by the SPL developer.

SWI-Prolog:

|  |
| --- |
| customize(CP,F,FCODE) :-  write('enter new customized code for:'),  write(CP),  read(USERCODE),  retract(applicationFile(F,FNAME,FCODE,FDEST)),  term\_string(CP,CPS),  sub\_string(FCODE, Before, L, After, CPS),  sub\_string(FCODE, \_, After, 0, Value2),  Initial is Before+L,  sub\_string(FCODE, 0, Initial, \_, Value1),  string\_concat(Value1,USERCODE,New1),  sub\_string(Value2, Before2, L2, After2, CPS),  Initial2 is After2+L,  sub\_string(FCODE, \_, Initial2, 0, Value3),  string\_concat(New1,Value3,New2),  assert(applicationFile(F,FNAME,New2,FDEST)). |

**Predicate 22:** isBoundTo(C,R)

It evaluates if there is a domain component C that is bound to a domain requirement R. For example, isBoundTo(C, R) returns true if C is bound to R.

Formalization: ∀C, ∃R. *domainComponent*(C) ∧ *domainRequirement*(R) ∧ *isBoundTo*(C,R)

****Predicate 23:** setSelected(IdRequirement)**

IdRequirement **identifies a domain requirement which is selected to configure (to be part of a software product).**

****Predicate 24:** product(IdProduct)**

IdProduct **identifies a SPL product.**

**Predicate 25:** isPartOfPS(P,SPL)

It evaluates if there is a product P that is part of a SPL. For example, isPartOfPS(P, SPL) returns true if P is part of SPL.

Formalization: ∀C, ∃R. *product*(P) ∧ *spl*(SPL) ∧ *isPartOfPS* (P,SPL)

**Predicate 26:** isPartOfAFP(F,P)

It evaluates if there is an application file F that is part of a product P. For example, isPartOfAFP(F, P) returns true if F is part of P.

Formalization: ∀C, ∃R. *applicationFile*(F) ∧ *product*(P) ∧ *isPartOfAFP* (F,P)

**Formula 5:** executeCopy(SPL,P)

It executes a copy of all the domain files select to be derived, as new application files.

The process is to: (i) take the information of the domain requirements R which are part of the SPL, (ii) take the information of the domain components C which are bound to the requirements, (iii) take the information of the domain files F which are part of the components. Then, new application files are created based on those domain files, and linked to a specific product P.

This process is only executed for the domain requirements R that were selected to configure **(to be part of a software product**).

Formalization: ∀SPL, ∃R, ∃C, ∃F, ∃P. *(spl*(SPL) ∧ *product*(P) ∧ *domainRequirement*(R) ∧ *domainComponent*(C) ∧ *domainFile*(F) ∧ *isPartOf*(R,SPL) ∧ *isBoundTo*(C,R) ∧ *isPartOf*(F,C) ∧ *setSelected*(R)) ⇒ (***applicationFile*(F)** ∧ ***isPartOfAFP*(F,P))**

SWI-Prolog:

|  |
| --- |
| executeCopy(SPL,PNAME) :- forall(  (spl(SPL),  domainRequirement(R),  domainComponent(C),  domainFile(F,FNAME,FCODE,FDEST),  product(PNAME),  isPartOf(R,SPL),  isBoundTo(C,R),  isPartOf(F,C),  setSelected(R)  ),  (  assert(applicationFile(F,FNAME,FCODE,FDEST)),  assert(isPartOfAFP(F,PNAME))  )  ). |

**Formula 6:** executeFragments(SPL)

It executes the fragments for a SPL derivation based on the selected features.

The process is to: (i) take the information of the domain requirements R which are part of the SPL, (ii) take the information of the domain components C which are bound to the requirements, (iii) take the information of the fragments FR which are part of the components, and (iv) execute the fragmentation actions (add, replace and hide). This process is only executed for the domain requirements R that were selected to configure **(to be part of a software product**).

Formalization: ∀SPL, ∃R, ∃C, ∃FR. *(spl*(SPL) ∧ *domainRequirement*(R) ∧ *domainComponent*(C) ∧ *fragment*(FR) ∧ *isPartOf*(R,SPL) ∧ *isBoundTo*(C,R) ∧ *isPartOf*(FR,C) ∧ *setSelected*(R)) ⇒ (***executeAddFragmentType*(FR)** ∧ ***executeReplaceFragmentType*(FR)** ∧ ***executeHideFragmentType*(FR))**

SWI-Prolog:

|  |
| --- |
| executeFragments(SPL) :- forall(  (  spl(SPL),  domainRequirement(R),  domainComponent(C),  fragment(FR,\_,\_),  isPartOf(R,SPL),  isBoundTo(C,R),  isPartOf(FR,C),  setSelected(R)  ),  (  executeReplaceFragment(FR),  executeAddFragment(FR),  executeHideFragment(FR)  )  ). |

**Formula 7:** executeDerivation(SPL,P)

It executes a product derivation SPL for a new product P.

Formalization: ∀SPL, ∃P. *(spl*(SPL) ∧ *product*(P) ∧ ***executeCopy*(SPL,P)** ∧ ***executeFragments*(SPL))**

SWI-Prolog:

|  |
| --- |
| executeDerivation(SPL,PNAME):- spl(SPL),  assert(product(PNAME)),  assert(isPartOfPS(PNAME,SPL)),  executeCopy(SPL,PNAME),  executeFragments(SPL). |

**Formula 8:** customizeDerivation(SPL)

It customizes a SPL derivation based on the customization files and customization points.

The process is to: (i) take the information of the domain requirements R which are part of the SPL, (ii) take the information of the domain components C which are bound to the requirements, (iii) take the information of the customization files CF which are part of the components, and (iv) execute the customizations. This process is only executed for the domain requirements R that were selected to configure **(to be part of a software product**).

Formalization: ∀SPL, ∃R, ∃C, ∃CF. *(spl*(SPL) ∧ *domainRequirement*(R) ∧ *domainComponent*(C) ∧ *customizationFile*(CF) ∧ *isPartOf*(R,SPL) ∧ *isBoundTo*(C,R) ∧ *isPartOf*(CF,C) ∧ *setSelected*(R)) ⇒ (***executeCustomization*(CF))**

SWI-Prolog:

|  |
| --- |
| customizeDerivation(SPL) :- forall(  (  spl(SPL),  domainRequirement(R),  domainComponent(C),  customizationFile(CF),  isPartOf(R,SPL),  isBoundTo(C,R),  isPartOf(CF,C),  setSelected(R)  ),  (executeCustomization(CF))  ). |

**Function 5:** verify(F)

It verifies that the file F contains code grammatically correct.

**Note:** there is not a SWI-Prolog programming definition, because the verify formula vary a lot depending on the file software language. The verification is different for a file written in PHP that for a file written in HTML. The next subsection shows an approach to implement the verify formula through the use of additional software tools.

**Formula 8:** verifyDerivation(SPL)

It executes a SPL verification.

The process is to: (i) take the information of the product P which is part of the SPL, (ii) take the information of the application files F which are part of the product, and (iii) execute the verification.

Formalization: ∀SPL, ∃R, ∃C, ∃F. *(spl*(SPL) ∧ *product*(P) ∧ *applicationFile*(F) ∧ *isPartOfPS*(P,SPL) ∧ *isPartOfAFP*(F,P)) ⇒ (***verify*(F)**)

SWI-Prolog:

|  |
| --- |
| verifyDerivation(SPL,PNAME) :- forall(  (  spl(SPL),  product(PNAME),  applicationFile(F),  isPartOfPS(PNAME,SPL),  isPartOfAFP(F,PNAME)  ),  (verify(F))  ). |