FiniteSatUSE User Manual

BGU Modeling Group

September 27, 2011

The FiniteSatUSE (`http://www.cs.bgu.ac.il/~modeling/?page_id=314`) is an extension of the USE system http://www.db.informatik.uni-bremen.de/projects/USE/ that provides correctness methods about finite satisfiability problems in UML class diagrams.

1 Running FiniteSatUSE

FiniteSatUSE (USE) is implemented in Java. In order to run the **FiniteSatUSE** tool, enter the following command at the prompt (assuming the current working directory is the directory of the FiniteSatUSE.jar file).

```
java -jar FinteSatUSE.jar
```

Now you are asked to enter a USE file path. You have the option to exit by typing ***end***.

```
Please enter a USE file path to examine or type end to exit:
```

At this point you are asked to enter one of the following commands at the prompt to invoke **FiniteSatUSE** 's implemented methods on the chosen file.

```
Please enter one of the following commands:
------------------------------------------------------------------------------------
-P                  Propagate the disjoint constraints within the class hierarchy cycles.
-save <file name>   Save the modified model (should be used only with -P flag raised).
-S                  Show the model.
-D                  Detect finite satisfiability problems.
-I                  Identify cause of finite satisfiability problems (critical cycles).
-C                  Identify class hierarchy cycles with disjoint or complete  constraints
--------------------------------------------------------------------------------------
```

Example 1 *Running Example*

```
> java -jar FinteSatUSE.jar

                           Welcome to the FiniteSatUSE- 2011
==============================================================================================


Please enter a USE file path to examine or type end to exit:
```

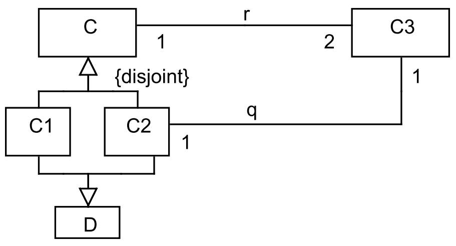

Figure 1: A class diagram

```
C:\Research\ReasoningProject\VitalyNewVersion\Tests\PropgationTestCases\example.use

Please enter one of the following commands:
---------------------------------------------------------------------------------------------
-P                      Propagate the disjoint constraints within the class hierarchy cycles .
-save <file name>       Save the modified model (should be used only with -P flag raised).
-S                      Show the model.
-D                      Detect finite satisfiability problems.
-I                      Identify cause of finite satisfiability problems (critical cycles).
-C                      Identify class hierarchy cycles with disjoint or complete
                        constraints
---------------------------------------------------------------------------------------------
-P -D -I -C -S -save C:\Research\ReasoningProject\VitalyNewVersion\Tests\PropgationTestCases\example_updated.use
Propagation:
Done!

The new added GSs are:
gs GSname gs_disjoint_1 type disjoint super D subClasses C1, C2

The new updated GSs are:
There is no updated GSs

Propagation execution time: 29 ms.
****************************************************************************
The Model:
model example

class C
attributes
end

class C1 < D, C
 attributes
end

class C2 < D, C
 attributes
end

class C3
attributes
end

class D
attributes
end

association r1
```

```
between
 C[1] role r1_1
 C3[2] role r1_2
end

association r2
between
 C3[1] role r2_1
 C2[1] role r2_2
end

gs GSname DefaultName0 type disjoint super C subClasses C1, C2
gs GSname gs_disjoint_1 type disjoint super D subClasses C1, C2

****************************************************************************
Detection:
The model is not finitly satisfiable

Detection execution time: 55 ms.
****************************************************************************
Identification:

Critical cycle alert!
assoc_r1 -> C -> C2 -> assoc_r2 -> C3 -> assoc_r1

The involved constraints in the cycle are:
Association r
Minimum multiplicity constraint 1.0 next to C2 and maximum multiplicity constraint 1.0 next to C3
Association r
Minimum multiplicity constraint 2.0 next to C3 and maximum multiplicity constraint 1.0 next to C

Class Hierarchies:
Super: C,  Sub: C2

Graph repesentaion of the cycle:
(C : C2)[weight=1.0], (C2 : assoc_r2)[weight=1.0], (assoc_r2 : C3)[weight=1.0], (C3 : assoc_r1)[weight=1.0],
 (assoc_r1 : C)[weight=0.5]

Cycle weight: 0.5

Continue looking for more cycles? (yes/no/all)
all
Identification Complete

Total identification time:5161 ms.
****************************************************************************
Class Hierarchy Cycles Identification:
The model is not finitely satisfiable. There is no need for class hierarchy cycles identification
Do you still want to show the cycles (yes/no)
yes
Class hierarchy cycle Alert! C, C1, D, C2, C
Continue looking for more circles? (yes/no/all)
all
Class hierarchy cycle Alert! D, C1, C, C2, D

Total identification time:7739 ms.

--------------------------------------------------------------------------------
Overall execution time: 12993 ms.
--------------------------------------------------------------------------------

Please enter a USE file path to examine or type end to exit:
```

```
end
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
                            Bye
           BGU Modeling Group Development Team 2011
              http://www.cs.bgu.ac.il/~modeling/
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

2 Specifying a Class diagram with USE

This section presents the expanded USE grammar that defines class diagrams for supporting GS constraints, qualifier, subsetting, redefinition, and XOR constraints. Grammar details are omitted that are irrelevant to the tool purposes (but optional in the language), such as operation, attributes, and constraints are omitted[1]. We also present several language constraints that are not specified in USE. The syntactic specification follows USE *EBNF* (Extended Backus-Naur Form) style. The table below presents the EBNF symbols.

Usage	**Notation**
definition	::=
nonterminal	⟨...⟩
alternation	|
option	[...]
repetition	{...}
grouping	(...)
terminal strings which are also symbols of the *EBNF*	′...′

We distinguish three kinds of syntax rules:

1. USE rules marked by the word "USE".
2. Rules that have been extended and are marked by the word "Extended".
3. Rules that have been added to USE specification and are marked by the word Added.

2.1 Defining a UML Model (Extended)

Every UML Model has a name and an optional body.

```
⟨umlmodel⟩   ::=  model ⟨modelname⟩ [⟨modelbody⟩]
⟨modelname⟩  ::=  ⟨name⟩
```

For example

[1]See the original USE specification at:`http://www.db.informatik.uni-bremen.de/projects/USE/use-documentation.pdf`

```
model academic
...
```

The model body consists of at least one class definition and an arbitrary number of association definitions. Enumeration definitions are allowed at the top of the body. OCL constraints may be defined (omitted from this specification).

```
⟨modelbody⟩ ::= {⟨enumerationdefinition⟩}
                ⟨classdefinition⟩
                {⟨classdefinition⟩ | ⟨associationdefinition⟩ |
                ⟨associationclassdefinition⟩}
                {⟨gsconstraintdefinition⟩}
```

2.2 Specification Elements

The following sections describe all available elements, which can be used in the model body.

2.2.1 Enumeration (USE)

Enumerations may be added at the top of the model body

```
⟨enumerationdefinition⟩ ::= enum ⟨enumerationname⟩ '{' ⟨elementname⟩ {,⟨elementname⟩} '}'
⟨enumerationname⟩       ::= ⟨name⟩
⟨elementname⟩           ::= ⟨name⟩
```

Example 2 *An enumeration definition with seven elements.*

```
enum Week {Su, Mo, Tu, We, Th, Fr, St}
```

2.3 Classes (USE)

A class has a name and optionally attribute and operation definitions. Class may be defined as an abstract class and it may have hierarchy relations with another classes. We omit from the syntax the rules for attribute and operation definition.
Syntax

```
⟨classdefinition⟩ ::= [abstract] class ⟨classname⟩ [< ⟨classname⟩ {,⟨classname⟩}]
                      end
⟨classname⟩       ::= ⟨name⟩
```

Note: The " < " notation denotes class hierarchy relation.

Constraint: 1) Class name is unique in a model. 2) Circular class hierarchy is disallowed.

Example 3 *The USE specification of Figure 2.*

```
model classHierarchies
class C
end

class C1 < C
end

class C2 < C
end
```

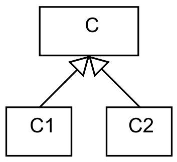

Figure 2: Class Hierarchies

2.3.1 Associations (Extended)

The USE definition requires a name, at least two participating classes [2], multiplicity information. Role names are optional. We extend the definition to include *qualifier*, association specialisation and the new UML 2 inter association constraints *subsetting* and *redefinition*[3]. All are optional .

⟨associationdefinition⟩	::=	⟨binaryassociation⟩ | ⟨naryassociation⟩
⟨binaryassociation⟩	::=	(association | composition | aggregation) ⟨associationname⟩ [< ⟨associationname⟩ {, ⟨associationname⟩}] between ⟨classname⟩ [⟨multiplicity⟩] [role ⟨rolename⟩] [ordered] [⟨subsettingconstraint⟩] [⟨redefinitionconstraint⟩] [⟨qualifier⟩] ⟨classname⟩ [⟨multiplicity⟩] [role ⟨rolename⟩] [ordered] [⟨subsettingconstraint⟩] [⟨ redefinitionconstraint⟩] [⟨qualifier⟩]
⟨naryassociation⟩	::=	(association | composition | aggregation) ⟨associationname⟩ between

[2]It is possible to define n ary associations. However, the FiniteSatUSE does not support verification with n-ary associations.

[3]The extension has been done before the new released version 3 of USE which support these new constraints

```
                                      ⟨classname⟩ [ ⟨multiplicity⟩ ] [role ⟨rolename⟩] [ordered]
                                      ⟨classname⟩ [ ⟨multiplicity⟩ ] [role ⟨rolename⟩] [ordered]
                                      ⟨classname⟩ [ ⟨multiplicity⟩ ] [role ⟨rolename⟩] [ordered]
                                      {⟨classname⟩ [ ⟨multiplicity⟩ ] [role ⟨rolename⟩] [ordered]}

⟨multiplicity⟩                 ::=    '['*']' | '[' ⟨digit⟩ {⟨digit⟩}  [.. ( * | ⟨digit⟩ {⟨digit⟩})] ']'
⟨subsettingconstraint⟩         ::=    '{' subsets⟨associationendrole⟩ {,⟨associationendrole⟩} '}'
⟨redefinitionconstraint⟩       ::=    '{' redefines⟨associationendrole⟩ {,⟨associationendrole⟩} '}'

⟨associationendrole⟩                  ⟨classname⟩.⟨rolename⟩

⟨qualifier⟩                    ::=    qualifier attributes ⟨attribute⟩ {,⟨attribute⟩}
⟨attribute⟩                    ::=    ⟨attributename⟩ : ⟨qualifiertype⟩
⟨attributename⟩                ::=    ⟨name⟩
⟨qualifiertype⟩                ::=    ⟨enumerationname⟩ | Integer | Real
```

Constraints:

1. Subsetting and redefinition constraints are constraints between association-ends (*properties* in UML 2). In UML 2 subsetting may occur only when the context of the *subsetting property* `conforms to` the context of the *subsetted property.* Redefinition occurs when the context of the *redefining property* is a sub-class of the context of the *redefined property.* For example, in the following class diagram:

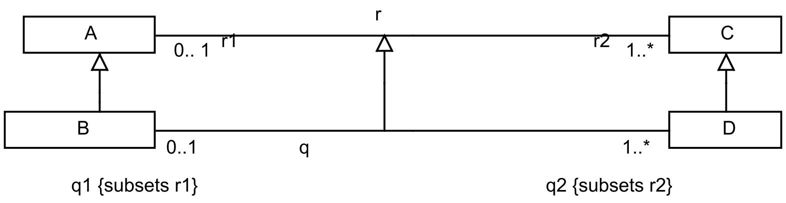

Figure 3: A class diagram with subsetting constraint

The properties (association-ends) $q1$, $q1$ are called the *subsetting properties* (*subsetting association-ends*) where the properties $r1$, $r2$ are called the *subsetted properties.* And:

- A is the context of property r2
- C is the context of property r1

- B is the context of property q2
- D is the context of property q1

Specification of the association q with its *subsetting constraints* is presented below:

```
association q between
```

```
B [0..1] role q1 {subsets C.r1}
D [1..*] role q2 {subsets  A.r2}
end
```

Note: The notation `classname.rolename` specifies the *subsetted property*, where the class `classname` is the *context* of the properties `rolename`.

Similarly for redefinition constrains.

2. Association hierarchies (also association-class hierarchy) raise a unique problem of ***property correspondence*** between the properties of the involved associations. For example, in Figure 4a, the relations between the association ends is not clear. Therefore, the syntax of association specialisation (association-class hierarchy) cannot be a plain class hierarchy syntax, but must include specification of property correspondence as a *subsetting* constraint as shown in Figure 4b. Unfortunately, this is not required by the UML 2 Meta-Model specification, although it is needed in order to avoid confusions.

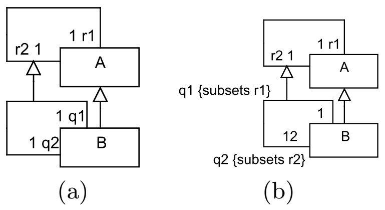

Figure 4: A specialisation of recursive associations

Example 4 *The USE specification of Figure 5*

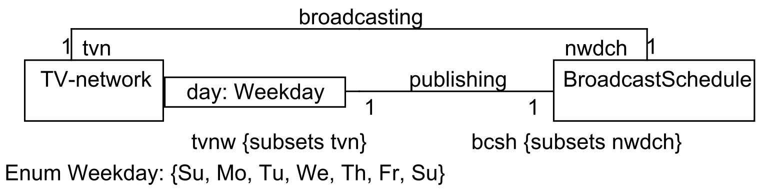

Figure 5: A class diagram with qualifier constraint

```
model qualifiedModel
enum Week {Su, Mo, Tu, We, Th, Fr, St}

class TvNetwork
end
```

```
class BroadcastSchedule
end

association broadcasting
between
TvNetwork [1] role tvn
BroadcastSchedule [1] role content
end

association publishing
between
TvNetwork [1] role bcsch qualifier attributes day:Week
BroadcastSchedule [1] role tvnw
end
```

Example 5 *The USE specification of Figure 6*

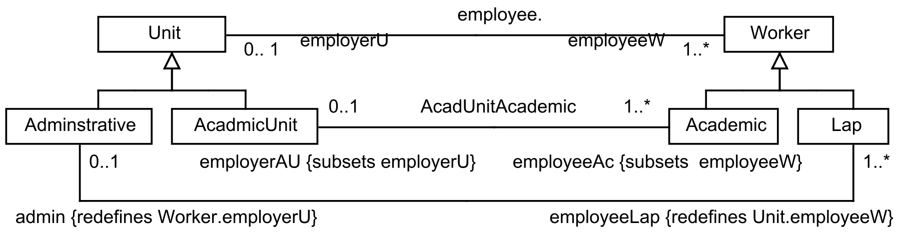
employee.
Unit
Worker
0.. 1
employerU
employeeW
1..*
Adminstrative
AcadmicUnit
0..1
AcadUnitAcademic
1..*
Academic
Lap
0..1
employerAU {subsets employerU}
employeeAc {subsets employeeW}
1..*
admin {redefines Worker.employerU}
employeeLap {redefines Unit.employeeW}

Figure 6: A class diagram with subsetting and redefinition constraints

```
model academic
class Unit
end

class AcadmicUnit < Unit
end

class Worker
end

class Academic < Worker
end

association employee between
Unit[0..1] role employerU
Worker [1..*] role employeeW
end
```

```
association AcadUnitAcademic <  employee between
AcadmicUnit[0..1] role employerAU {subsets Worker.employerU}
Academic [1..*] role employeeAc {subsets Unit.employeeW}
end
```

Example 6 *The USE specification of Figure 7*

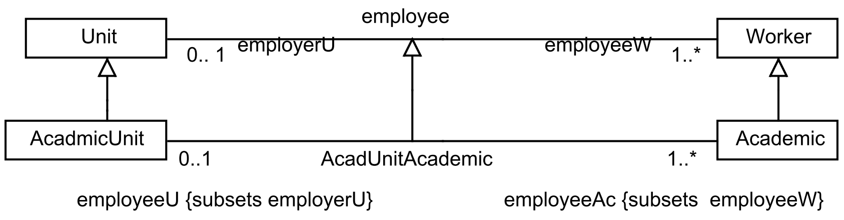

Figure 7: An association specialisation

```
model academic
class Unit
end

class AcadmicUnit < Unit
end

class Adminstrative < Unit
end

class Worker
end

association employee between
Unit[0..1] role employerU
Worker [1..*] role employeeW
end

association AcadUnitAcademic between
AcadmicUnit[0..1] role employerAU {subsets Worker.employerU}
Academic [1..*] role employeeAc {subsets Unit.employeeW}
end
```

2.3.2 Association class (Extended)

Association classes combine the body elements of classes and associations.

```
⟨associationclassdefinition⟩  ::=  [abstract] associationclass ⟨associationclassname⟩
                                   [< ⟨associationclassname⟩ { ,⟨associationclassname⟩}]
                                   between

                                   (⟨classname⟩ | ⟨associationclassname⟩) [ ⟨multiplicity⟩ ]
                                   [role ⟨rolename⟩] [ordered]
                                   [⟨subsettingconstraint⟩] [⟨redefinitionconstraint⟩]

                                   (⟨classname⟩ | ⟨associationclassname⟩) [ ⟨multiplicity⟩ ]
                                   [role ⟨rolename⟩] [ordered]
                                   [⟨subsettingconstraint⟩] [⟨redefinitionconstraint⟩]
```

Constrint The definition of association class is extended to enforce the following constraints:

1. A specification of property correspondence as a *subsetting* constraint for class hierarchies between association classes (Similar to association specialisation).
2. Unlike USE, we disallow a class hierarchy between an association class and a regular class.

Example 7 *The USE specification of Figure 8*

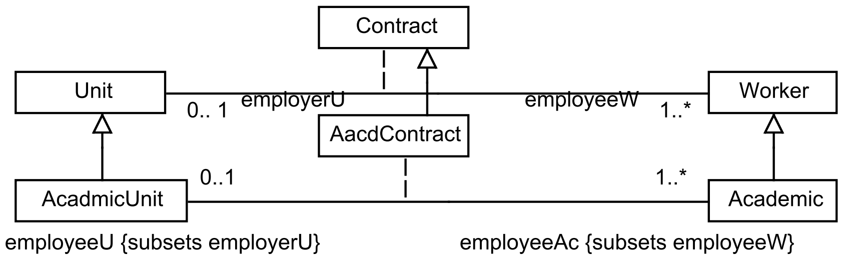

Figure 8: A class diagram with association class hierarchy

```
model academic
class Unit
end

class AcadmicUnit < Unit
end

class Worker
end

class Academic < Worker
```

```
end

associationclass employee between
Unit[0..1] role employerU
Worker [1..*] role employeeW
end

associationclass acadUnitAcademic <  employee between
AcadmicUnit[0..1] role employerAU {subsets Worker.employerU}
Academic [1..*] role employeeAc {subsets Unit.employeeW}
end
```

2.3.3 Generalization-set constraints (Added)

The USE grammar is augmented to support generalization-set constraints.

$\langle gsconstraintdefinition\rangle$ ::= gs [$\langle gsname\rangle$] [$\langle gstype\rangle$]
super
($\langle classname\rangle$ | $\langle associationclassname\rangle$)
subclasses
($\langle classname\rangle$ | $\langle associationclassname\rangle$)
$\{,(\langle classname\rangle$ | $\langle associationclassname\rangle)\}$

$\langle gsname\rangle$ ::= $\langle name\rangle$

$\langle gstype\rangle$ ::= disjoint | complete | overlapping | incomplete
disjoint_complete | disjoint_incomplete |
overlapping_incomplete | overlapping_complete

Constraints Generalization-set constraints must be preceded by correspondly class hierarchy declarations. That is, the classes occurring after the keyword *subclasses* must be declared as subclasses of the class that occurs after the keyword *superclass*, and the declaration must appear before the generalization set deceleration.

Example 8 *The USE specification of the class diagram in Figure 9*

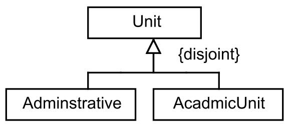

Figure 9: A generalization set constraint

```
model academic
class Unit
end

class AcadmicUnit < Unit
end

class Adminstrative < Unit
end

gs GSname academic type disjoint super Unit subClasses AcadmicUnit, Adminstrative
```

2.3.4 XOR constraint (Added)

We add the xor constraint to USE syntax

$$\langle\texttt{xordefinition}\rangle \quad ::= \quad \texttt{xor}\ \langle\texttt{classname}\rangle\ \langle\texttt{rolename}\rangle, \langle\texttt{rolename}\rangle\ \{, \langle\texttt{rolename}\rangle\}$$

Example 9 *The USE specification of the class diagram in Figure 10*

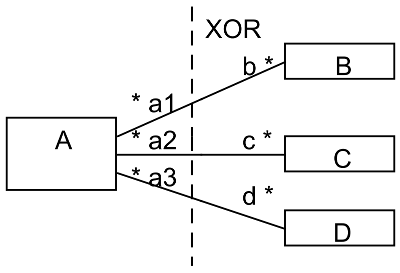

Figure 10: A class diagram with XOR constraint

```
model xormodel

class A
end

class B
end

class C
end

class D
end
```

```
association r
between
A [*] role a1
B [*] role b
end

association q
between
A [*] role a2
C [*] role c
end

association w
between
A [*] role a3
D [*] role d
end

xor A, b, c, d;
```

Acknowledgment We would like to thank Victor Makarenkov, Rami Prilutsky, Vitaly Bichov and Michael Brichko for their help in constructing the ***FiniteSatUSE*** tool.