CVL 1.2 User Guide

Table of Contents

1 Overview

1.1 What is Common Variability Language (CVL)?

The Common Variability Language is a generic and separate language for modeling variability in models in any Domain Specific Language (DSL) that is defined based on Meta Object Facility (MOF). In CVL approach we have three models:

- The base model – a model described in a DSL.
- The variability model – the model that defines variability on the base model.
- The resolution model – the model that defines how to resolve the variability model to create a new model in the base DSL.

A base model can have several variability models and each variability model can have several resolution models. With variability model and resolution model properly defined, the user can run our generic CVL model-to-model transformation to generate new resolved models which conform to the base DSL.

1.2 What does CVL tool support provide?

The CVL language is developed under Eclipse and the CVL tool support is provided in terms of Eclipse plug-ins. The CVL tool support includes the following:

- A graphical editor and a tree editor for creating and viewing the CVL model including the variability model and the resolution model.
- A resolution model generator that is integrated with the CVL graphical editor. A resolution model can be generated from the variability model elements that the user wants to include in the resolution model by selecting them in the CVL graphical editor.
- The CVL model-to-model transformation to generate new resolved models in the base DSL.
- A set of APIs, which can be implemented in a base DSL editor for integration with CVL editor. The CVL-enabled base DSL editor supports highlighting, creating fragments from selection and etc.

1.3 What are the concepts in the CVL language?

The CVL language supports multi-stage model configuration by supporting expressing variability into two conceptually distinctive layers:

- The user-centric layer – the variability of user-centric layer expresses the variability of higher feature level, just like in feature modeling. The concepts of CVL (*type*, *composite variability*, *constraint* and *iterator*) are sufficient to mimic feature diagrams (mandatory/optional feature, feature dependencies, XOR/OR and cardinality).
- The product-realization layer – the variability of product realization layer defines lower level, fairly fine-grained but necessary operations that are required to complete the transformation from the base model to the new resolved model. The CVL concepts *value substitution*, *reference substitution*, *fragment substitution* and *boundary point binding* are used to express the variability of the product realization layer.

1.3.1 Fragment Substitution

We find it necessary to explain the most important CVL executable primitive that is used to express the variability in the product-realization layer before we go further with the CVL experience.

A fragment substitution may replace any model fragment (a set of arbitrary model elements and the references among them) with any other model fragment described in the same base DSL. As illustrated in Figure 1, in order to substitute the “Placement Fragment” (model objects in red) in the base model with the “Replacement Fragment” (model objects in blue) in the model fragment library, the following need to be done:

- The CVL language uses the concept “Boundary Point” to specify any arbitrary model fragment. The boundary points record the inbound and outbound references to and from the fragment (marked as “ToP”, “FrP1”, “FrP2”, “ToR”, “FrR1”, “FrR2”).
- Bind the replacement boundary points to the placement boundary points accordingly. In the illustration, “ToR” is bound to “ToP”, “FrR1” is bound to “FrP1” and “FrR2” is bound to “FrP2”.
- Run the CVL model-to-model transformation to generate the resolved model with the fragment replaced.

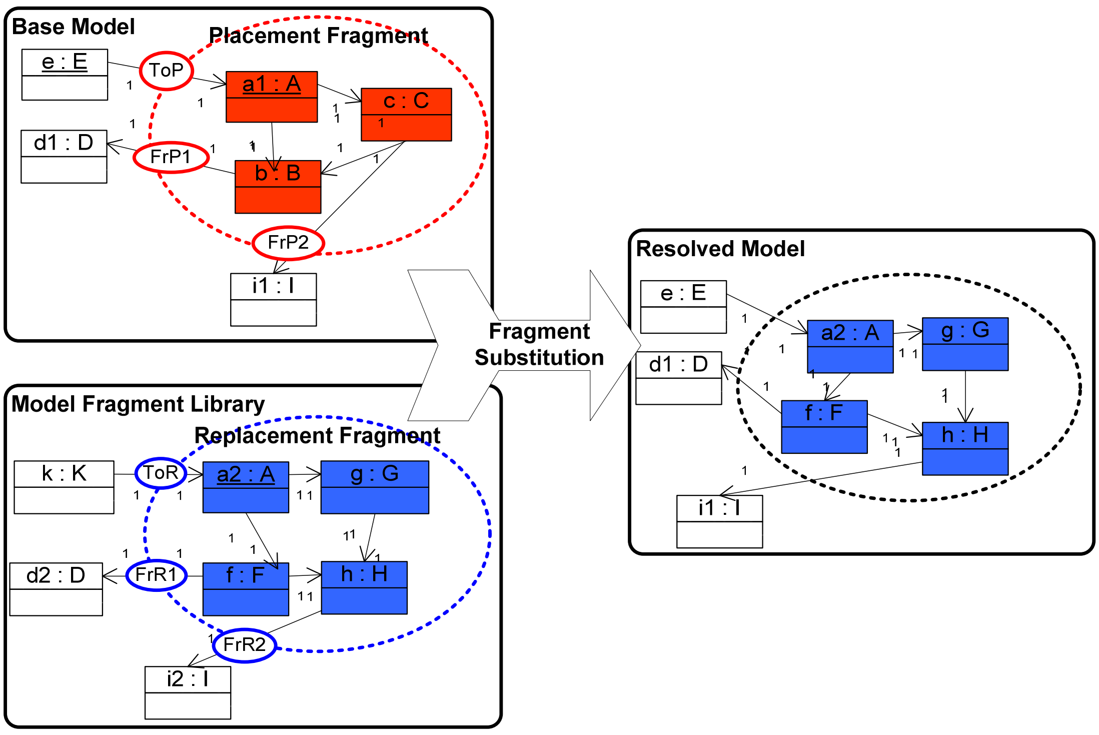


Figure 1 Fragment Substitution

2 Installation

- Check www.variabilitymodeling.org for the latest installation instructions.

3 Getting Started

In this mini-tutorial we will use an example of watch product line to demonstrate how to generate a new product model from a base watch product model using CVL approach. The scenarios of the watch example are provided by MetaCase, originally modeled in their proprietary DSL development framework and later recreated in Eclipse by us using EDAP (a DSL for modeling embedded distributed applications). In the example scenario, the watch product "Sporty" consists of:

- A display "X234"
- A logical watch "TST" which is modeled as a state machine with composite states "Time", "Stopwatch" and "Timer".

We will show how to apply CVL approach to the watch product model "Sporty" to generate another watch product model "Delicia" with the same display as "Sporty" but with an enhanced logical watch "TAST" ("AlarmClock" functionality added). The scenario is illustrated in Figure 2.

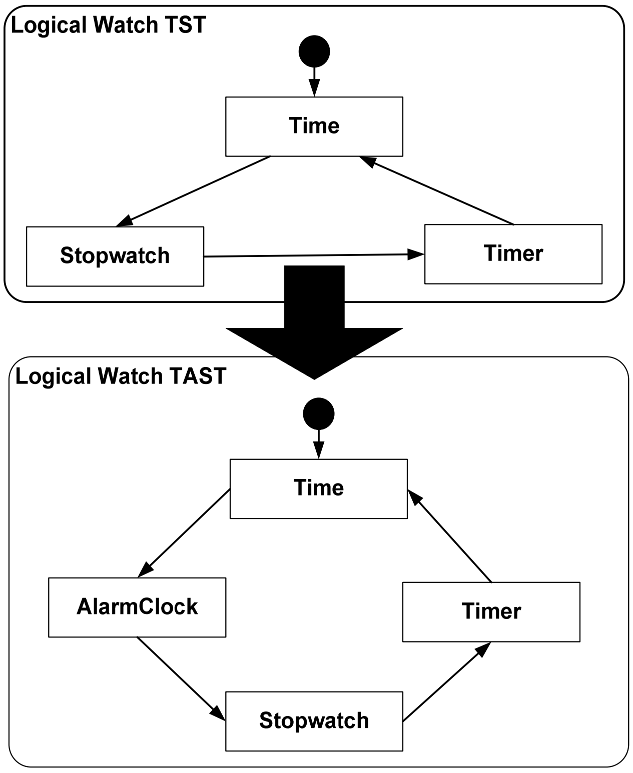
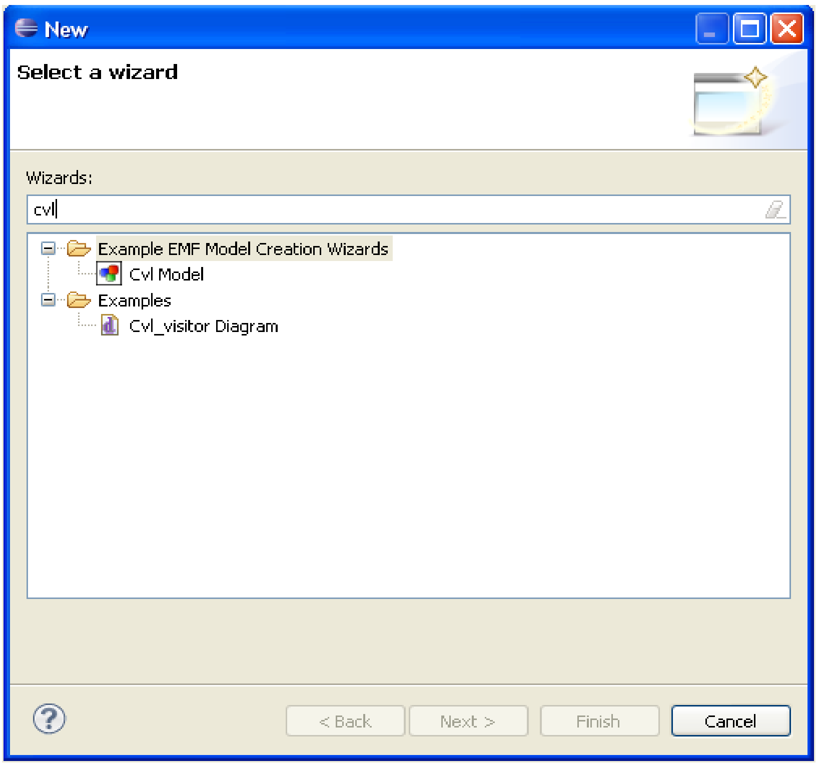


Figure 2 Logical Watch *TST* to *TAST*

3.1 Creating a CVL Model

You can choose to create a CVL diagram (*. cvl_diagram) in the graphical editor or create a CVL model (*.cvl) in the tree-view editor as shown in Figure 3.



New

Select a wizard

Wizards:

cvl

- Example EMF Model Creation Wizards
  - Cvl Model
- Examples
  - Cvl_visitor Diagram

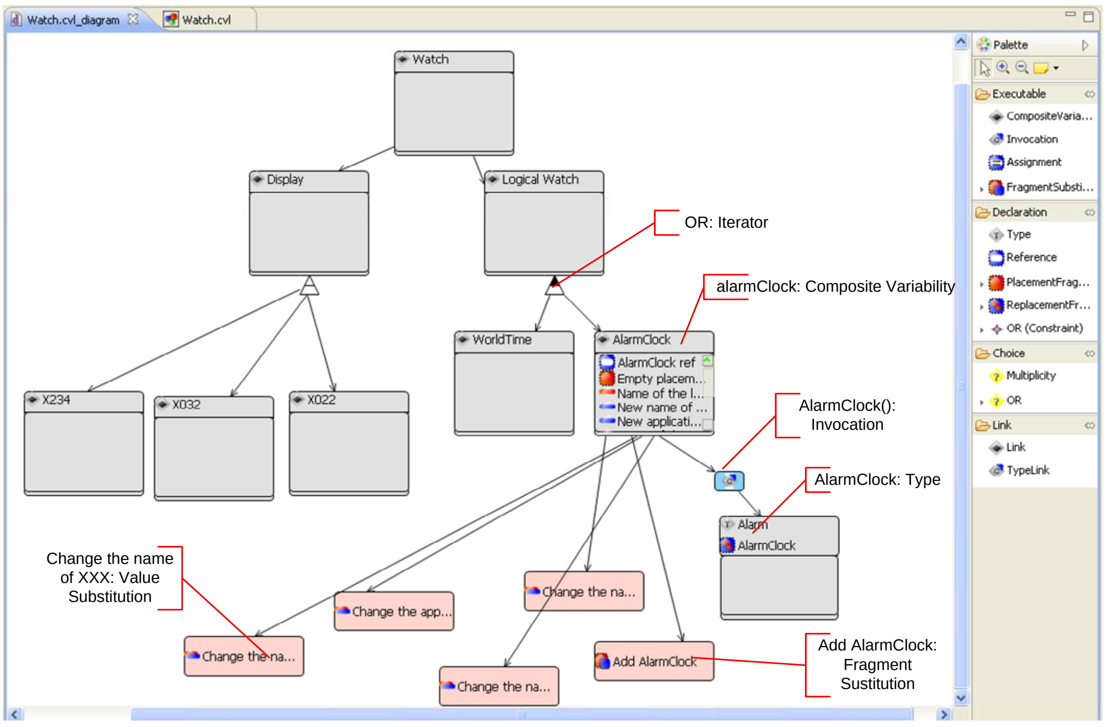
Figure 3 Creating a new CVL model

We recommend you to start with creating a CVL diagram in the graphical editor for the following reasons:

- The CVL has incorporated essential parts of feature diagram in its graphical concrete syntax. Therefore users with experience in feature diagram will find it easy to define variability in the user-centric layer using the feature-diagram-like legend.
- While creating a CVL diagram, a corresponding and synchronized CVL model that can be opened in the tree editor will be automatically generated and synchronized.

3.1.1 Defining Variability in the User-Centric Layer in CVL Diagram

Figure 4 shows the variability model of watch product line in the CVL graphical editor. Each concept appears in the watch family metamodel and the feature model of watch DSL such as “Display “, “Logical Watch” and watch applications, are modeled as “Composite Variability” and the same hierarchy is also assembled. The “XOR” and “OR” iterator defines the variability on the presence of certain features, e.g. the “OR” upon watch applications and the “XOR” upon display of the watch.



The height of the text box and its associated line increases or decreases as you add text. To change the width of the comment, drag the side handle.

Figure 4 CVL Model of the Watch Product Line in CVL Graphical Editor

3.1.2 Defining Variability in the Product-Realization Layer

As shown in Figure 5, the details of the realization of a specific product are described only in the CVL tree-view editor in order to separate the view of low level CVL operations from the high level variability as shown in Figure 4. For instance, in order to express the variability of the existence of a watch application “Alarm clock”, we define a placement fragment for the composite state in the logical watch state machine, the replacement fragment as the “Alarm clock” in the library models, the substitution to replace an empty placement with the “Alarm clock” and all necessary value substitutions on basically names.

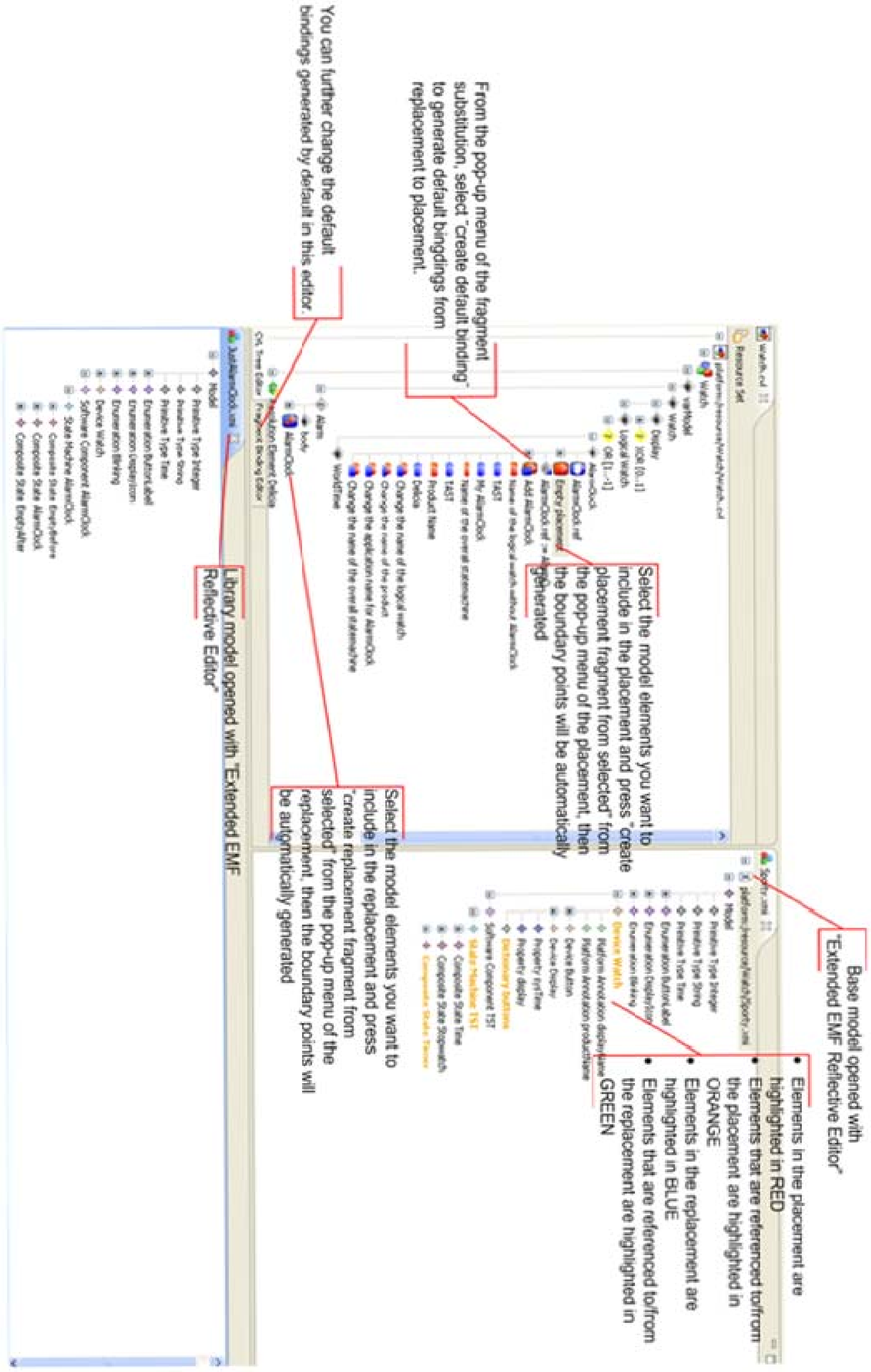

You can further change the default bindings generated by default in this editor.
From the pop-up menu of the fragment substitution, select "create default binding" to generate default bindings from replacement to placement.
Select the model elements you want to include in the placement and press "create placement fragment from selected" from the pop-up menu of the placement, then the boundary points will be automatically generated
Select the model elements you want to include in the replacement and press "create replacement fragment from selected" from the pop-up menu of the replacement, then the boundary points will be automatically generated
Library model opened with "Extended EMF Reflective Editor"
Base model opened with "Extended EMF Reflective Editor"
• Elements in the placement are highlighted in RED
• Elements that are referenced to/from the placement are highlighted in ORANGE
• Elements in the replacement are highlighted in BLUE
• Elements that are referenced to/from the replacement are highlighted in GREEN

Figure 5 CVL Model in the CVL Tree-View Editor

3.1.2.1 Loading Base Model and Library Model into the CVL Editor

Open the CVL model in the CVL tree-view editor. Load the base model where you will define the placement fragment/value/object and the library model where you will select the replacement fragment by choosing “Load Resource” in the right-click pop-up menu.

3.1.2.2 Open Base Model and Library Model in the Proper Editor

If the DSL you are working with has a CVL-enable graphical editor by implementing a set of APIs of CVL, the editor will support highlighting and creating fragments from selection. Open the base model and the library model using the CVL-enable editor if applicable, or else open the base model and the library model with “Extended EMF Reflective Editor”.

3.1.2.3 Creating Fragment Substitution

Creating Placement Fragment

Select the base model elements that you want to include in the placement fragment and choose “create placement fragment from selected” from right-click pop-up menu. The boundary points of the placement fragment will be generated automatically.

You can also see that in the base model, the elements that are included in the placement fragment are highlighted in red and the elements that are referenced from/to the placement fragment are highlighted in orange. Note that it is possible to define a “Placement Fragment” inside a “Replacement Fragment” in case of the need for further customization of the “Replacement Fragment”.

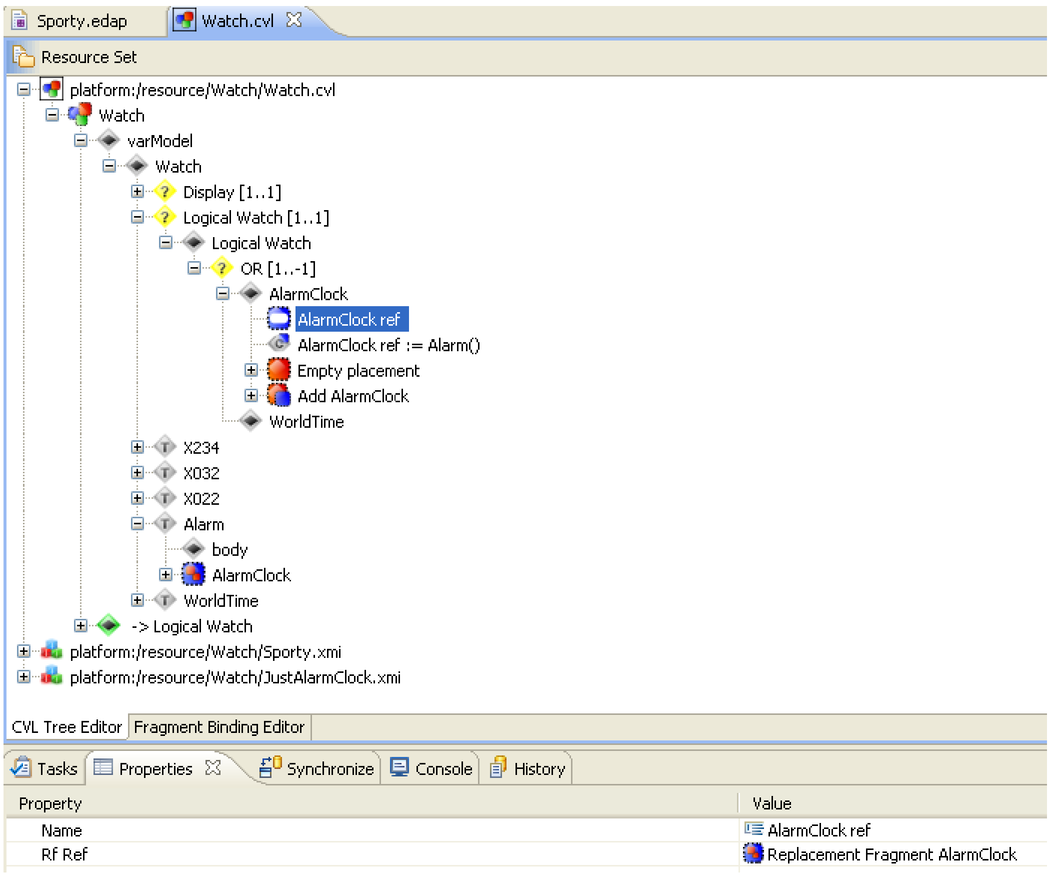
Creating Replacement Fragment

Select the model elements in the library model that you want to include in the replacement fragment and choose “create replacement fragment from selected” from right-click pop-up menu. The boundary points of the replacement fragment will be generated automatically.

You can also see that in the library model, the elements that are included in the replacement fragment are highlighted in blue and the elements that are referenced from/to the replacement fragment are highlighted in green.

In which way you can retrieve the instance of the replacement fragment depends on where you define the replacement fragment. If the replacement fragment is not defined inside a Type, the instance of the replacement fragment is then as available as it is. If the replacement fragment is defined inside a Type, we need to do the following to get an instance of the replacement fragment:

- Create a “Replacement Fragment Reference” to assign a name to the instance of the replacement fragment that the invocation of the type will return (see Figure 6).

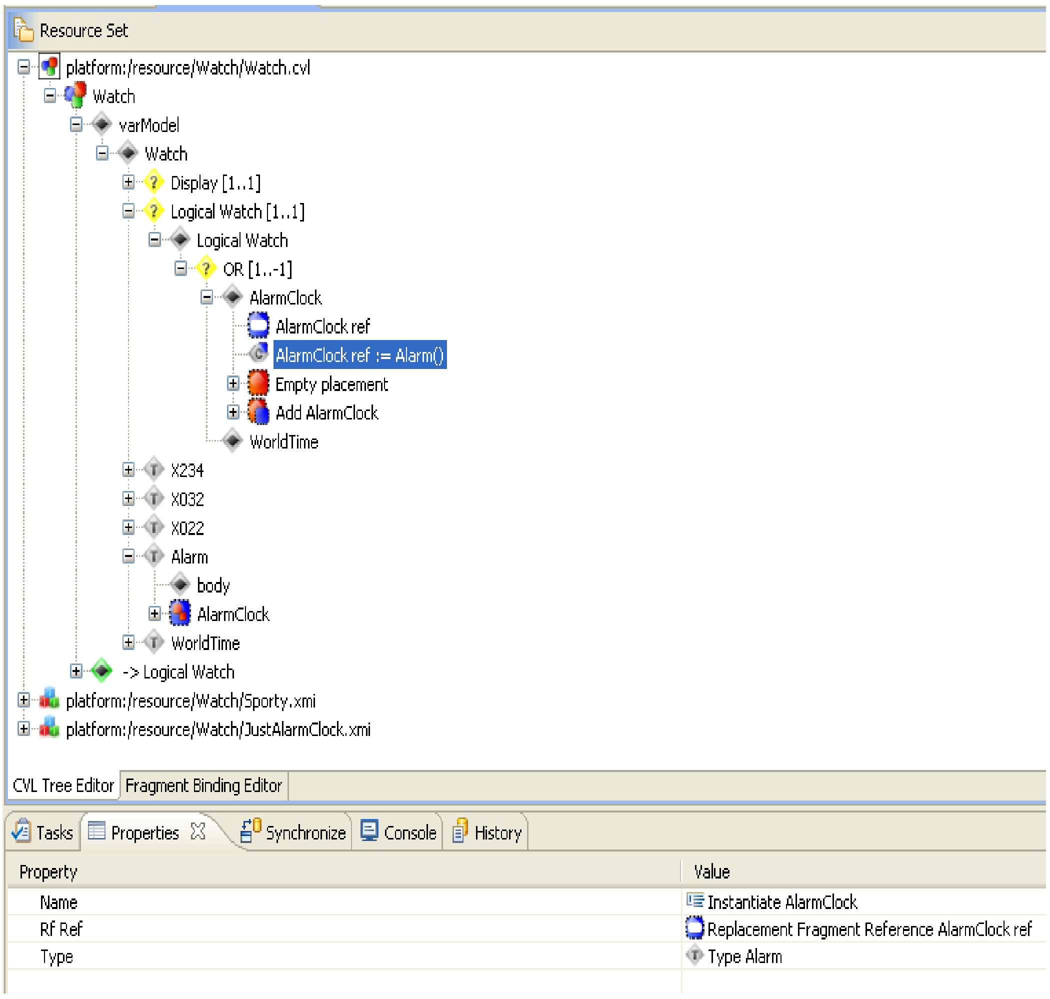


```
platform:/resource/Watch/Watch.cvl
  Watch
    varModel
      Watch
        Display [1..1]
        Logical Watch [1..1]
          Logical Watch
            OR [1..-1]
              AlarmClock
                AlarmClock ref
                AlarmClock ref := Alarm()
                Empty placement
                Add AlarmClock
              WorldTime
        X234
        X032
        X022
        Alarm
          body
          AlarmClock
        WorldTime
    -> Logical Watch
platform:/resource/Watch/Sporty.xmi
platform:/resource/Watch/JustAlarmClock.xmi
```

Property	Value
Name	AlarmClock ref
Rf Ref	Replacement Fragment AlarmClock

Figure 6 Assigning a name to the instance of the replacement fragment

- Invoke the type to get the returned instance of the replacement fragment (see Figure 7).

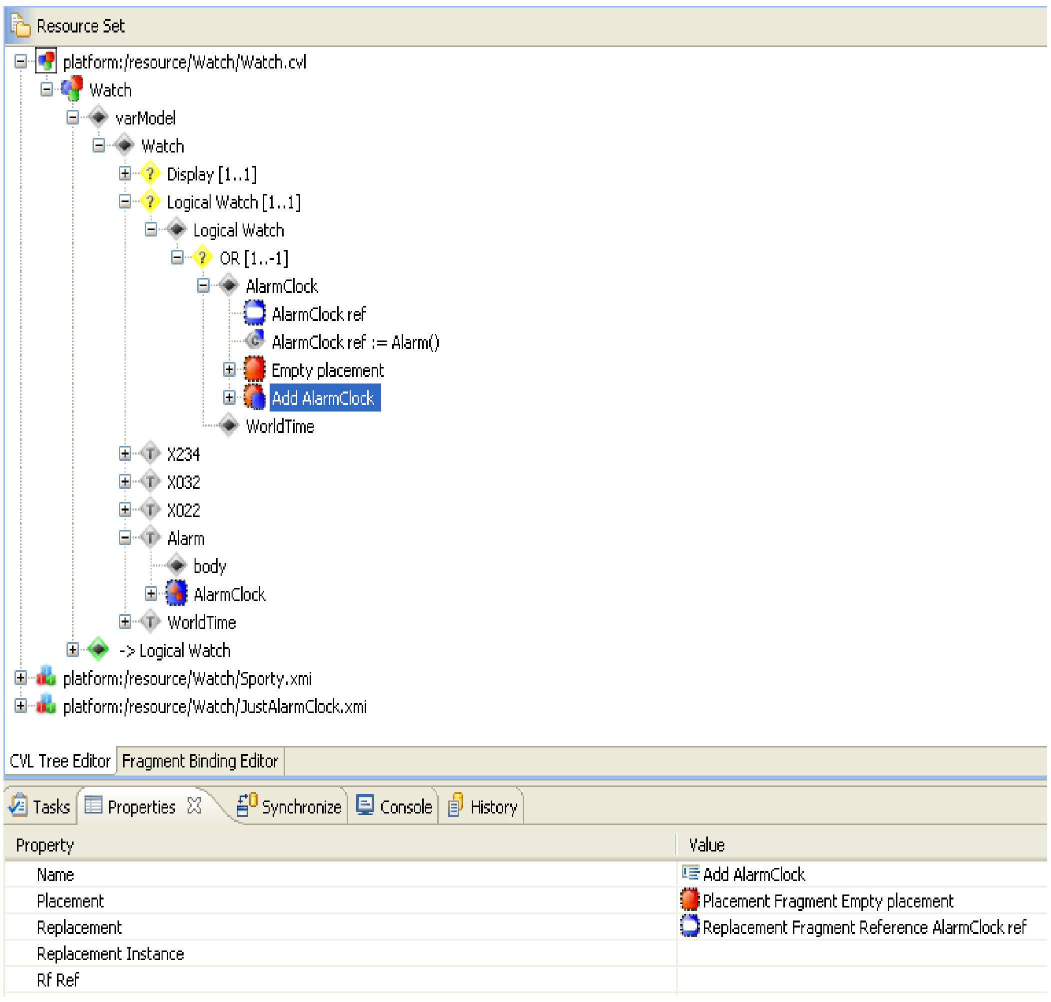


Property	Value
Name	Instantiate AlarmClock
Rf Ref	Replacement Fragment Reference AlarmClock ref
Type	Type Alarm

Figure 7 Invoking a Type to get returned instance of the replacement fragment

Creating Fragment Substitution

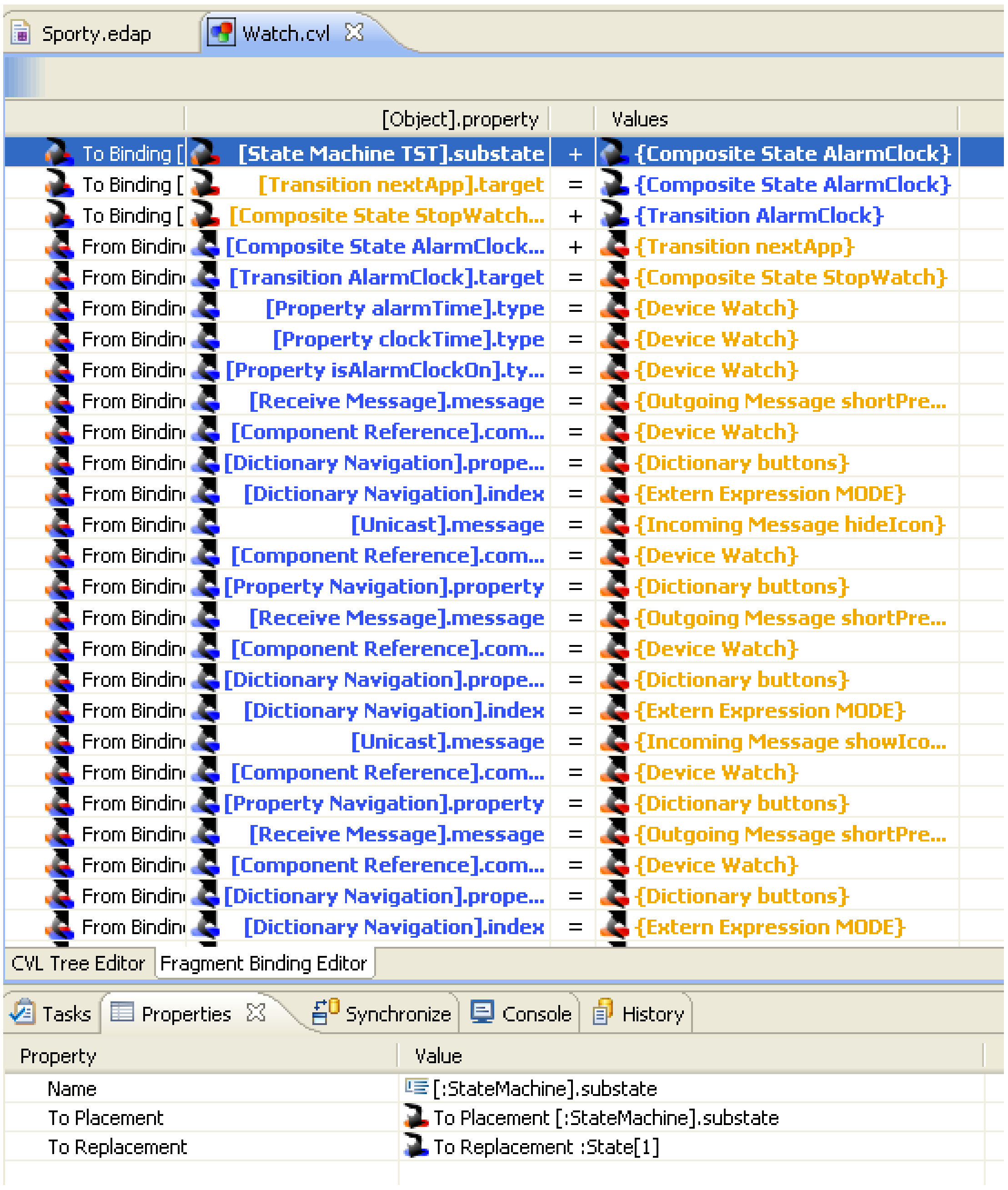
Create the fragment substitution by assigning the correct placement fragment and the instance of the replacement fragment, as shown in Figure 8. Create default boundary point bindings by choosing “Create Default Replacement Bindings” in the right-click pop-up menu. After the default bindings are generated, you can view and edit them in the binding editor (see Figure 9) by choosing “Open Bindings Editor” in the same pop-up menu.



```
Resource Set
platform:/resource/Watch/Watch.cvl
  Watch
    varModel
      Watch
        Display [1..1]
        Logical Watch [1..1]
          Logical Watch
            OR [1..-1]
              AlarmClock
                AlarmClock ref
                AlarmClock ref := Alarm()
                Empty placement
                Add AlarmClock
              WorldTime
        X234
        X032
        X022
        Alarm
          body
          AlarmClock
        WorldTime
    -> Logical Watch
platform:/resource/Watch/Sporty.xmi
platform:/resource/Watch/JustAlarmClock.xmi
```

Property	Value
Name	Add AlarmClock
Placement	Placement Fragment Empty placement
Replacement	Replacement Fragment Reference AlarmClock ref
Replacement Instance	
Rf Ref	

Figure 8 Creating Fragment Substitution



Sporty.edap | Watch.cvl

	[Object].property		Values
To Binding [	[State Machine TST].substate	+	{Composite State AlarmClock}
To Binding [	[Transition nextApp].target	=	{Composite State AlarmClock}
To Binding [	[Composite State StopWatch...	+	{Transition AlarmClock}
From Bindin	[Composite State AlarmClock...	+	{Transition nextApp}
From Bindin	[Transition AlarmClock].target	=	{Composite State StopWatch}
From Bindin	[Property alarmTime].type	=	{Device Watch}
From Bindin	[Property clockTime].type	=	{Device Watch}
From Bindin	[Property isAlarmClockOn].ty...	=	{Device Watch}
From Bindin	[Receive Message].message	=	{Outgoing Message shortPre...
From Bindin	[Component Reference].com...	=	{Device Watch}
From Bindin	[Dictionary Navigation].prope...	=	{Dictionary buttons}
From Bindin	[Dictionary Navigation].index	=	{Extern Expression MODE}
From Bindin	[Unicast].message	=	{Incoming Message hideIcon}
From Bindin	[Component Reference].com...	=	{Device Watch}
From Bindin	[Property Navigation].property	=	{Dictionary buttons}
From Bindin	[Receive Message].message	=	{Outgoing Message shortPre...
From Bindin	[Component Reference].com...	=	{Device Watch}
From Bindin	[Dictionary Navigation].prope...	=	{Dictionary buttons}
From Bindin	[Dictionary Navigation].index	=	{Extern Expression MODE}
From Bindin	[Unicast].message	=	{Incoming Message showIco...
From Bindin	[Component Reference].com...	=	{Device Watch}
From Bindin	[Property Navigation].property	=	{Dictionary buttons}
From Bindin	[Receive Message].message	=	{Outgoing Message shortPre...
From Bindin	[Component Reference].com...	=	{Device Watch}
From Bindin	[Dictionary Navigation].prope...	=	{Dictionary buttons}
From Bindin	[Dictionary Navigation].index	=	{Extern Expression MODE}

CVL Tree Editor | Fragment Binding Editor

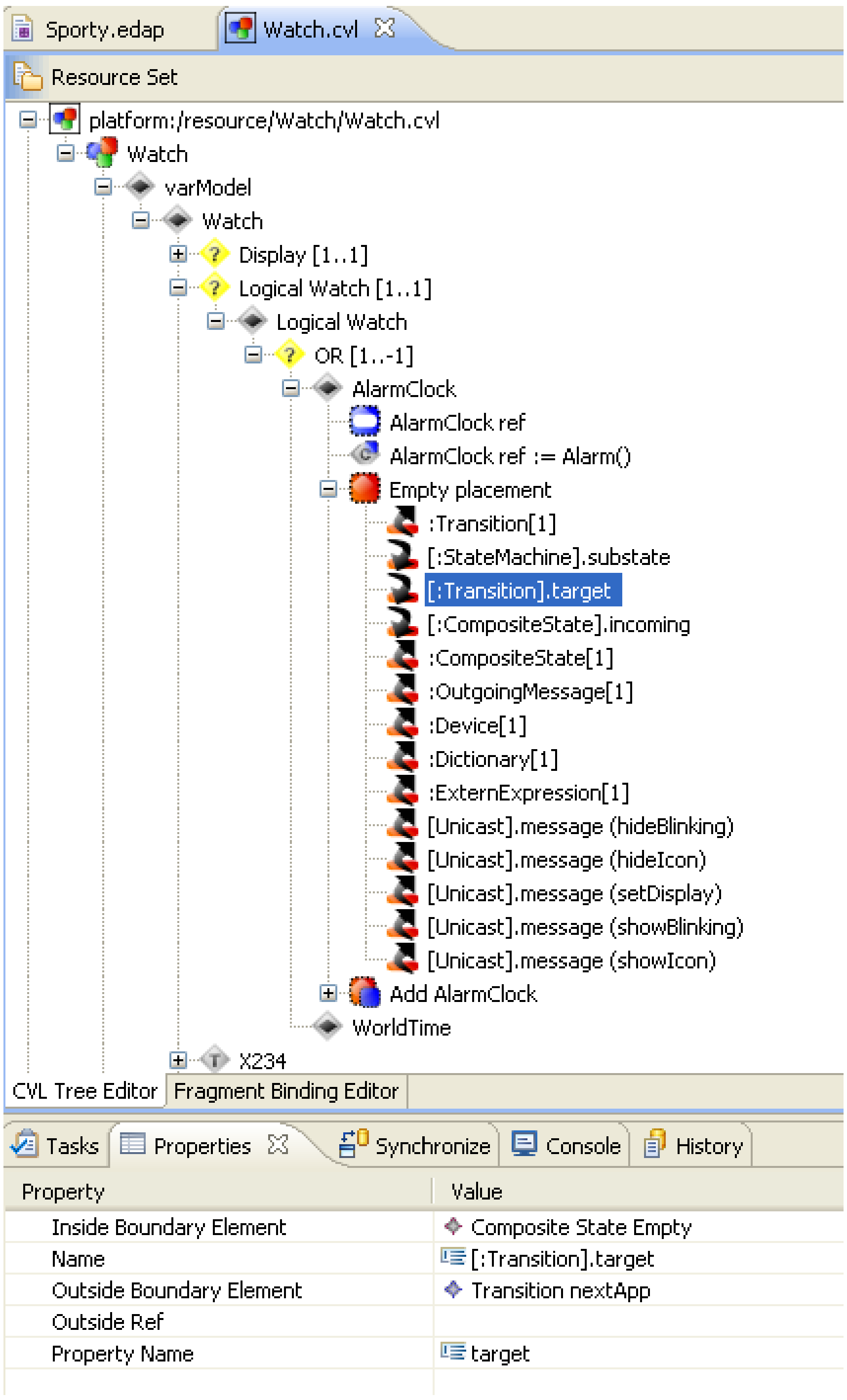
Tasks | Properties | Synchronize | Console | History

Property	Value
Name	[:StateMachine].substate
To Placement	To Placement [:StateMachine].substate
To Replacement	To Replacement :State[1]

Figure 9 Fragment Binding Editor

In particular while editing on the default bindings, the following explanations might help:

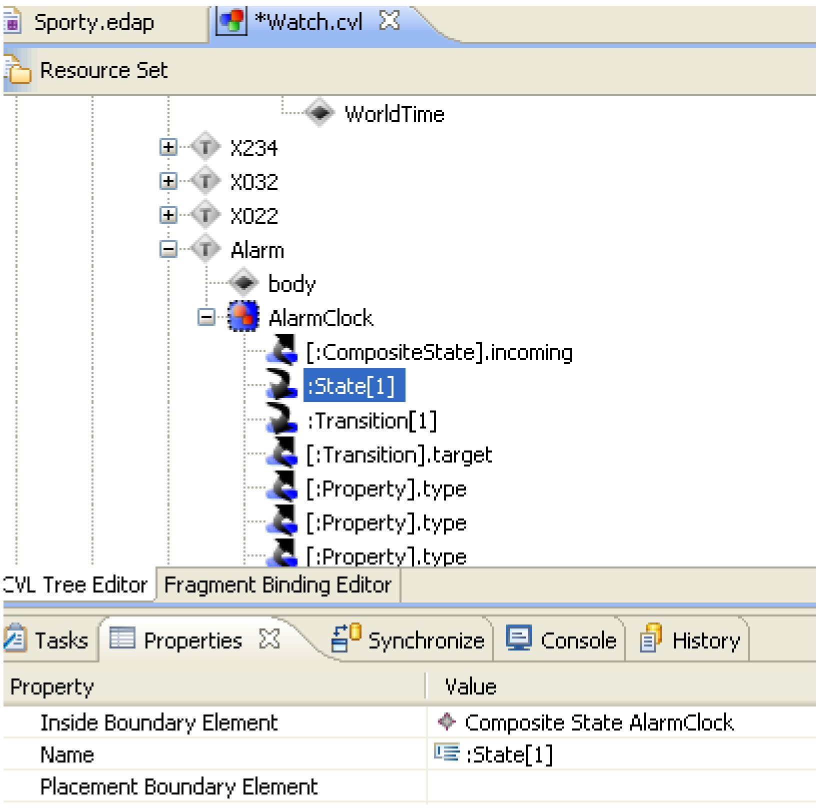
- "ToBinding" is used to bind "ToPlacement" and "ToReplacement". As illustrated in Figure 1, a "ToPlacement" can be bound to a "ToReplacement" as long as they are pointed to "Inside Boundary Element(s)" of the same type regardless of from which type of "Outside Boundary Element(s)" they are pointed. Figure 10 and Figure 11 show one "ToPlacement" and one "ToReplacement" that are bound in the fragment substitution.



```
platform:/resource/Watch/Watch.cvl
  Watch
    varModel
      Watch
        Display [1..1]
        Logical Watch [1..1]
          Logical Watch
            OR [1..-1]
              AlarmClock
                AlarmClock ref
                AlarmClock ref := Alarm()
                Empty placement
                  :Transition[1]
                  [:StateMachine].substate
                  [:Transition].target
                  [:CompositeState].incoming
                  :CompositeState[1]
                  :OutgoingMessage[1]
                  :Device[1]
                  :Dictionary[1]
                  :ExternExpression[1]
                  [Unicast].message (hideBlinking)
                  [Unicast].message (hideIcon)
                  [Unicast].message (setDisplay)
                  [Unicast].message (showBlinking)
                  [Unicast].message (showIcon)
                Add AlarmClock
              WorldTime
        X234
```

Property	Value
Inside Boundary Element	Composite State Empty
Name	[:Transition].target
Outside Boundary Element	Transition nextApp
Outside Ref	
Property Name	target

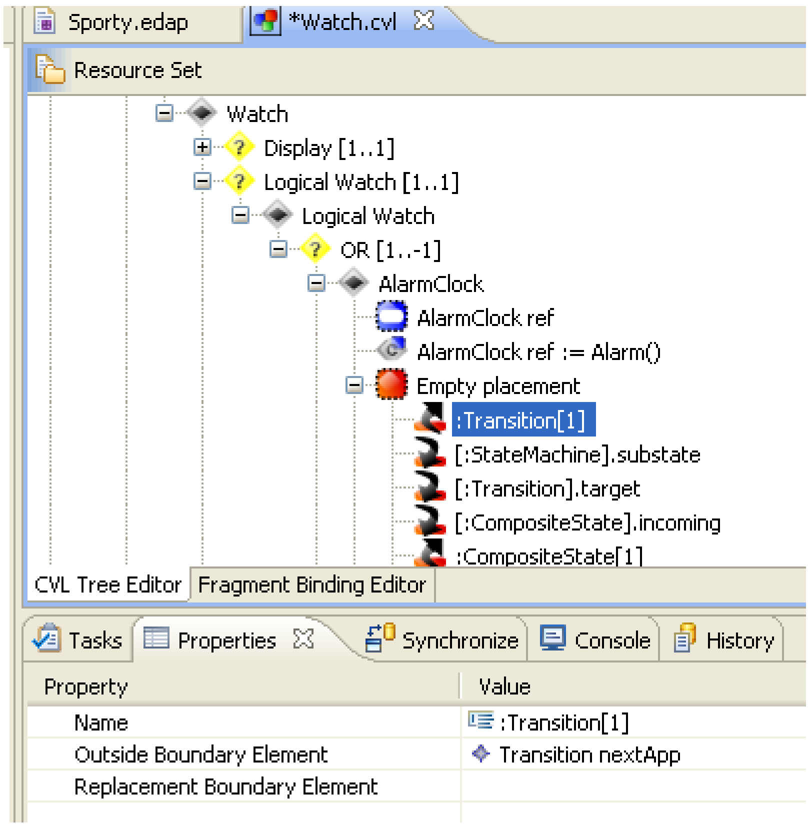
Figure 10 ToPlacement



Property	Value
Inside Boundary Element	Composite State AlarmClock
Name	:State[1]
Placement Boundary Element	

Figure 11 ToReplacement

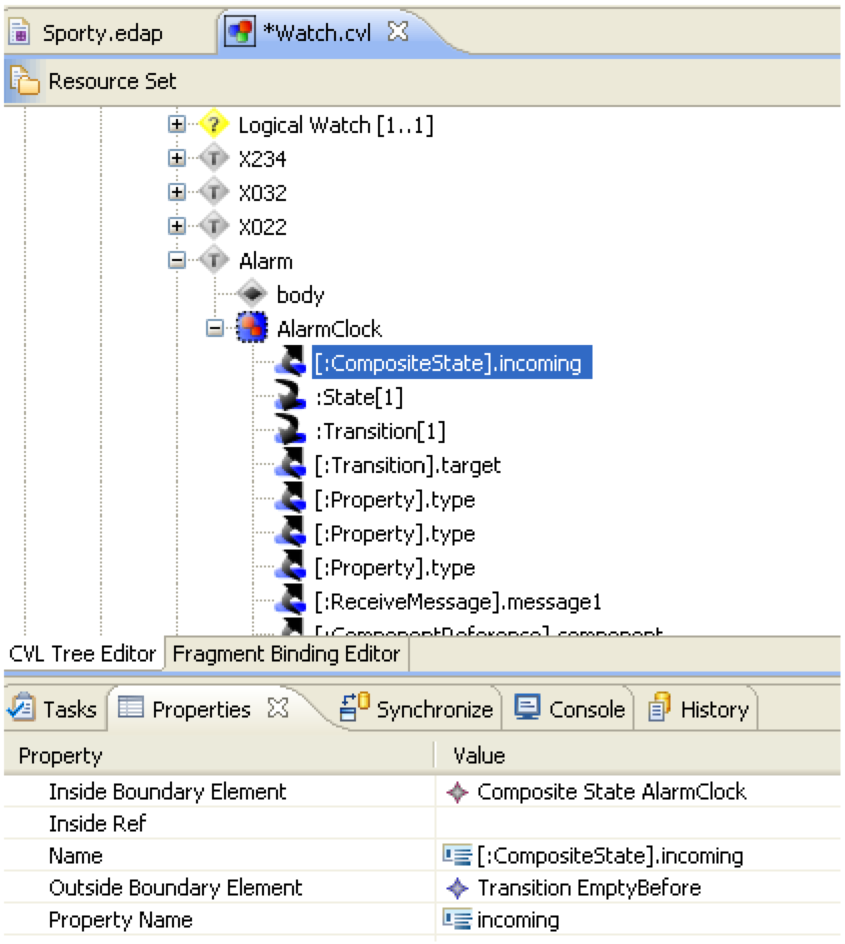
- "FromBinding" is used to bind "FromPlacement" and "FromReplacement". As illustrated in Figure 1, a "FromPlacement" can only be bound to a "FromReplacement" as long as they are pointed to objects of the same type outside the fragments regardless of from which types of objects they are pointed inside the fragments. Figure 12 and Figure 13 show one "ToPlacement" and one "ToReplacement" that are bound in the fragment substitution.



```
Watch
  Display [1..1]
  Logical Watch [1..1]
    Logical Watch
      OR [1..-1]
        AlarmClock
          AlarmClock ref
          AlarmClock ref := Alarm()
          Empty placement
            :Transition[1]
            [:StateMachine].substate
            [:Transition].target
            [:CompositeState].incoming
            :CompositeState[1]
```

Property	Value
Name	:Transition[1]
Outside Boundary Element	Transition nextApp
Replacement Boundary Element	

Figure 12 FromPlacement


Sporty.edap | *Watch.cvl

Resource Set

- Logical Watch [1..1]
- X234
- X032
- X022
- Alarm
  - body
  - AlarmClock
    - [:CompositeState].incoming
    - :State[1]
    - :Transition[1]
    - [:Transition].target
    - [:Property].type
    - [:Property].type
    - [:Property].type
    - [:ReceiveMessage].message1
    - [:ComponentReference].component

CVL Tree Editor | Fragment Binding Editor

Tasks | Properties | Synchronize | Console | History

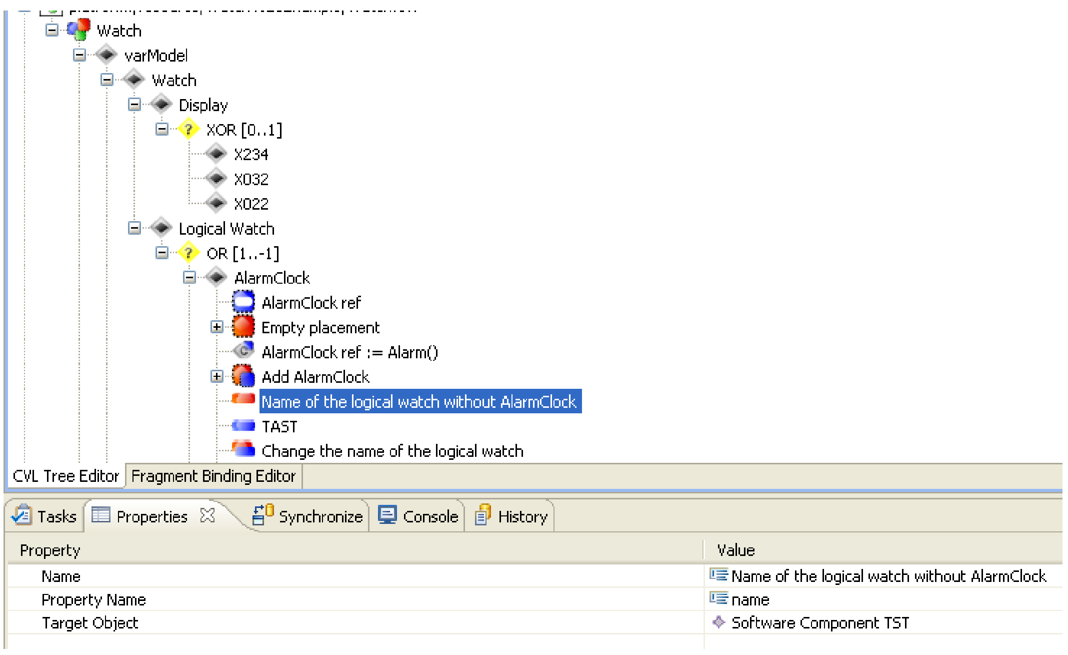
Property	Value
Inside Boundary Element	Composite State AlarmClock
Inside Ref	
Name	[:CompositeState].incoming
Outside Boundary Element	Transition EmptyBefore
Property Name	incoming

Figure 13 FromReplacement

3.1.2.4 Creating Value Substitution

You need to follow the following to create a value substitution:

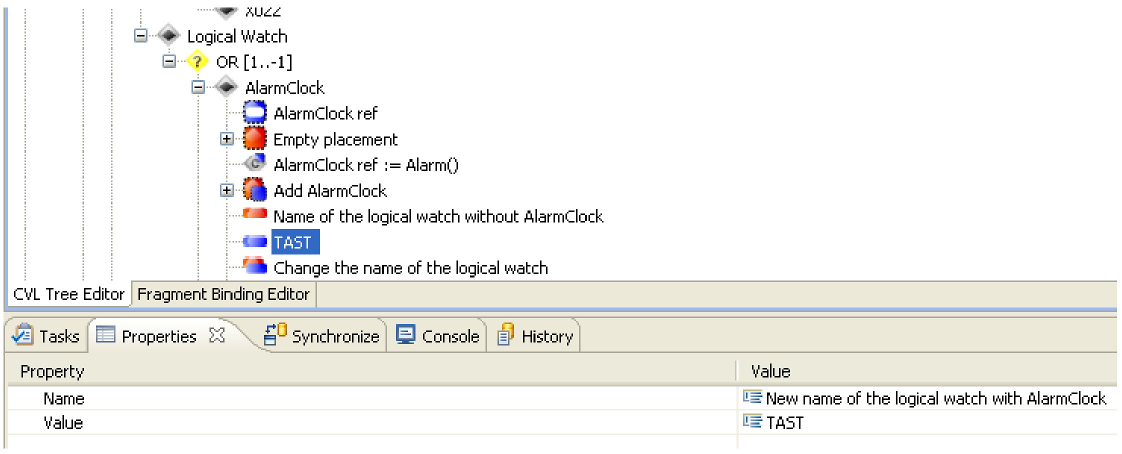
- Create a “Placement Value” and define the property name of the target object that you want to change value on as shown in Figure 14. Note that it is possible to define a “Placement Value” inside a “Replacement Fragment” in case of the need for further customization of the “Replacement Fragment”.



Property	Value
Name	Name of the logical watch without AlarmClock
Property Name	name
Target Object	Software Component TST

Figure 14 Placement Value

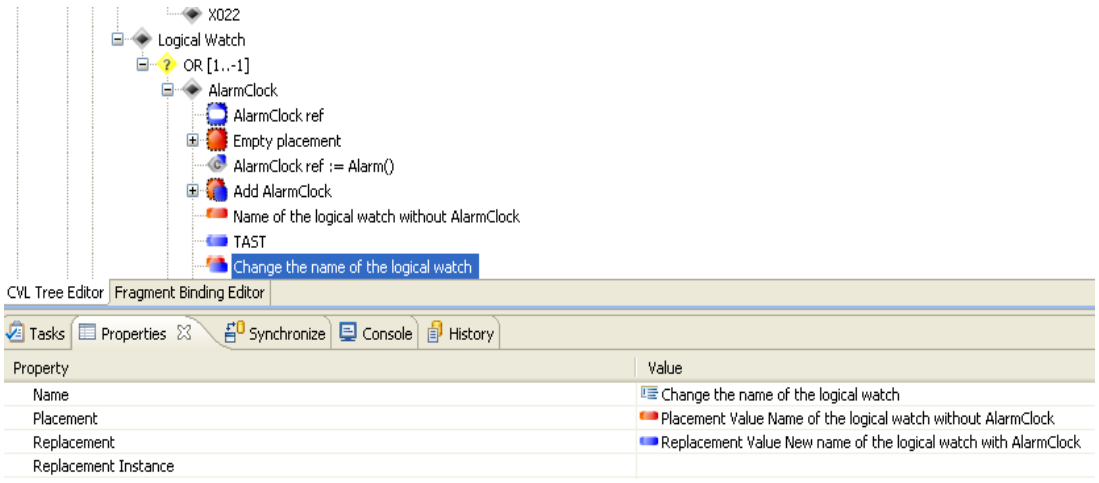
- Create a “Replacement Value” that you want to assign to the target property of the target object as shown in Figure 15.



Property	Value
Name	New name of the logical watch with AlarmClock
Value	TAST

Figure 15 Replacement Value

- Create a “Value Substitution” as shown in Figure 16.



```
X022
Logical Watch
  OR [1..-1]
    AlarmClock
      AlarmClock ref
      Empty placement
      AlarmClock ref := Alarm()
      Add AlarmClock
      Name of the logical watch without AlarmClock
      TAST
      Change the name of the logical watch
```

Property	Value
Name	Change the name of the logical watch
Placement	Placement Value Name of the logical watch without AlarmClock
Replacement	Replacement Value New name of the logical watch with AlarmClock
Replacement Instance	

Figure 16 Value Substitution

3.1.2.5 Creating Reference Substitution

Creating a reference substitution is very similar to creating a value substitution. The only two differences are:

- In the properties of a “Placement Object”, there is one called “ReferredObject” instead of “Value” in the “Placement Value”. “ReferredObject” should be the object the “old” reference points to.
- In the properties of a “Replacement Object”, there is one called “Target Object” instead of “Value” in the “Replacement Value”. “Target Object” here should be the object the “new” reference is supposed to point to.

Note that it is possible to define a “Placement Object” inside a “Replacement Fragment” in case of the need for further customization of the “Replacement Fragment”.

3.2 Creating Resolution Models and Generate Resolved Models

We need resolution models to derive different products from the variability model of watch product line. Since the three types of substitutions are executable primitives, the user is only supposed to make choices upon iterators while making resolution models. As shown in Figure 17, the user is able to select the feature he/she wants to include in the product and then a resolution model can be generated automatically and also highlighting support for the resolution choices is provided. Press the button shown in Figure 17 to generate resolution model from the selected choices and remember to save the diagram right after the resolution model is generated. Press the button shown in Figure 17 to run CVL transformation to generate resolved models.

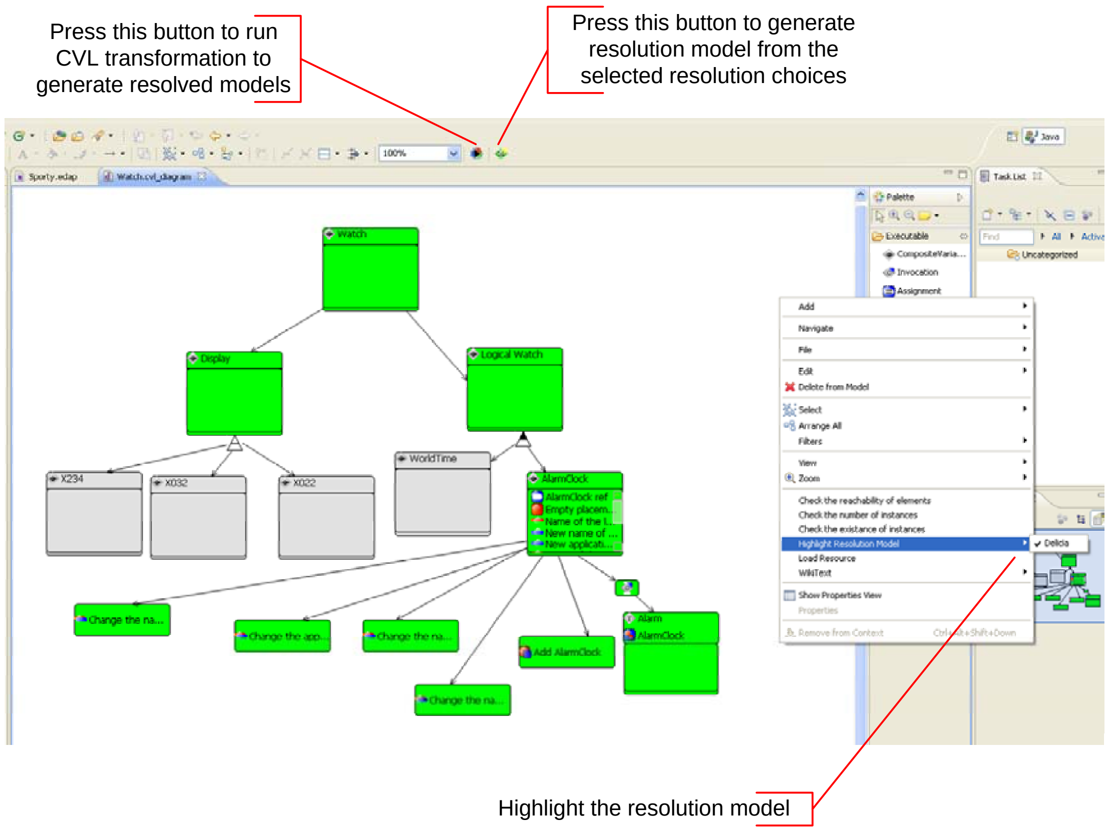

Press this button to run CVL transformation to generate resolved models
Press this button to generate resolution model from the selected resolution choices
Highlight the resolution model
Watch
Display
Logical Watch
X234
X032
X022
WorldTime
AlarmClock
Alarm
Change the na...
Change the app...
Add AlarmClock
Add
Navigate
File
Edit
Delete from Model
Select
Arrange All
Filters
View
Zoom
Check the reachability of elements
Check the number of instances
Check the existance of instances
Highlight Resolution Model
Load Resource
WikiText
Show Properties View
Properties
Remove from Context
Delicia

Figure 17 Resolved Product Highlighted