C-SAW User's Manual

Version 1.0

Released January 2004

Yuehua Lin
Jeff Gray
Jing Zhang

September 2004

http://www.gray-area.org/Research/C-SAW

Software Composition & Modeling Laboratory

SOFTCOM

Department of Computer and Information Sciences

University of Alabama at Birmingham

This project was previously funded by the DARPA Program Composition for Embedded Systems (PCES) program, and currently supported by the National Science Foundation under CSR-SMA-0509342.

Contents

1 Introduction

1.1 Overview of C-SAW

The Constraint-Specification Aspect Weaver (C-SAW) is a model-to-model transformation engine that weaves crosscutting constraints or changes into domain specific models automatically for rapid model construction and evolution. It is implemented as a Plug-in component of Generic Modeling Environment (GME), which is a meta-environment for constructing system and software models, developed by Vanderbilt University.

C-SAW combines the ideas of Aspect-Oriented Software Development (AOSD) and Model-Integrated Computing (MIC). MIC has been refined at Vanderbilt University over the past decade to assist with creation and synthesis of computer-based systems. Aspect-Oriented Software Development (AOSD) is a new model for software composition that contributes to better separation of concerns and improved modularity of software. Compared to other model transformation tools, C-SAW offers the ability to explore numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model. This permits a modeler to make changes more easily to the base model without manually visiting multiple locations in the model.

GME is a toolkit for realizing the principles of MIC. It is a UML-based meta-modeling environment that can be configured and adapted from meta-level specifications (called the modeling paradigm) that describe the domain. When using the GME, a modeling paradigm is loaded into the tool to define an environment containing all the modeling elements and valid relationships that can be constructed in a specific domain. GME also provides mechanisms for developing model interpreters, adds-on components and plug-in components for various purposes. For example, a model interpreter can supply an ability to generate other software artifacts (e.g., code or simulation scripts) from the models.

C-SAW is implemented as a Plug-in component of GME. An overview of model transformation using C-SAW is depicted in Figure 1-1. As shown in the figure, model transformation translates between source and target model(s), which can be instances of the same meta-models. Transformation rules and application strategies are written in a special language, called the Embedded Constraint Language (ECL), which is described in details in Chapter 4. Typically, a transformation specification includes an aspect specification and one or more strategy specifications. The aspect specification defines a set of locations in the domain models; and strategy specifications describe the transformation behaviors. The source model(s) and the transformation specification(s) are interpreted by the transformation engine to generate the target model(s). All the metamodels and models need to be built within GME.

Within C-SAW there includes an ECL parser and an interpreter for ECL to perform the transformation by interacting with modeling APIs provided by GME. A C-SAW user doesn’t need to know the internal parser and the interpreter, but need to provide the these inputs of C-SAW: source models and their metamodel built with GME and the transformation specifications.

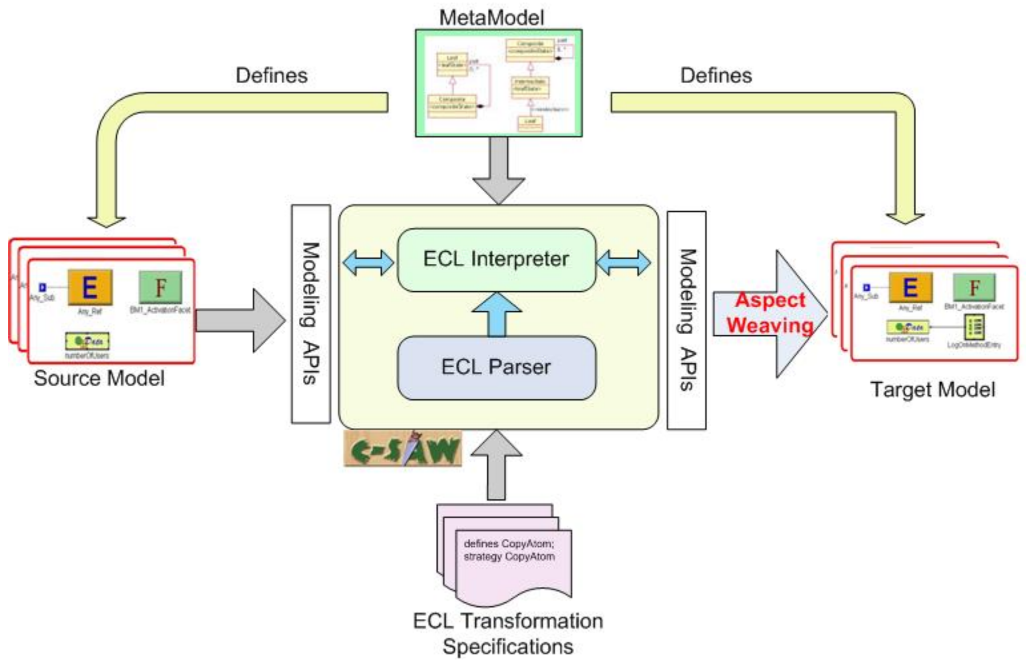


Figure 1-1

Additional information about C-SAW, including software downloads and video demos, is available at: http://www.gray-area.org/Research/C-SAW.

2 Installation

2.1 Installing GME

To use C-SAW, GME first needs to be installed on your computer. GME is available at http://www.isis.vanderbilt.edu/Projects/gme/. Instructions on how to install and run GME can be found on this website. C-SAW users should first become familiar with GME and the basic GME GUI commands and interfaces. The sources models need to be transformed by C-SAW should be built within GME in advance.

Different C-SAW versions are provided to support GME3-r4.11.10 or the above GME versions. You need to choose the appropriate C-SAW version that supports the GME version, with which you develop your models.

2.2 Installing C-SAW

The new weaver can be tied to any paradigm. Suppose you need to work on a GME project, you install the C-SAW as follows.

Step 1:

- Download the C-SAW zip file from http://www.gray-area.org/Research/C-SAW/
- Unzip the C-SAW zip file to get the C-SAW component (e.g., called Component.dll here)

Step 2:

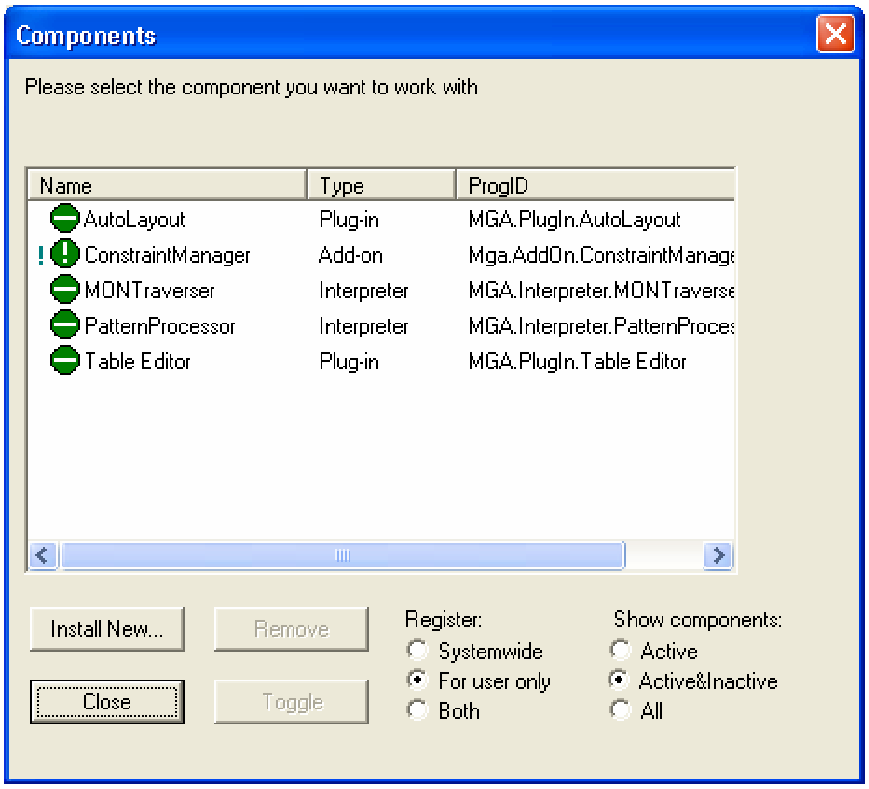
- Launch GME and Open the GME project (mga file extension) using the "Open Project…" option under the File menu
- Select the File Menu and then click on the register component command

Step 3:

- This brings up the Components dialog as shown in Figure 2-1.
- Click the installNew button; Go to the folder you stored the component.dll, select component.dll
- Click "Open"

Step 4:

- You will go back to the Components dialog and the NewWeaver Plugin is already on the list.
- Select it and click the Toggle button. See Figure 2-2.



Components

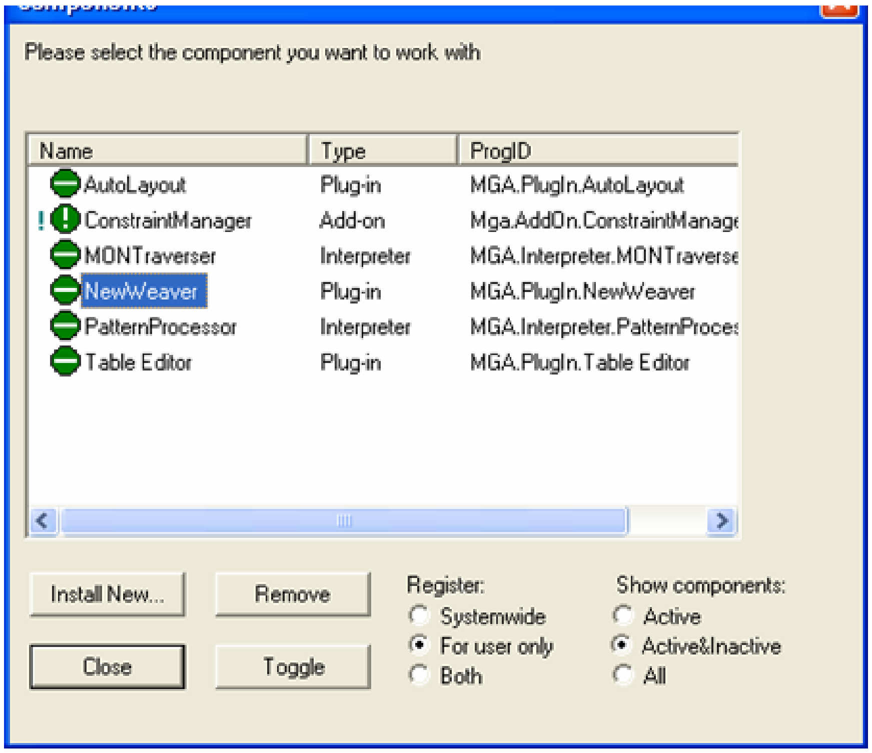
Please select the component you want to work with

Name	Type	ProgID
AutoLayout	Plug-in	MGA.PlugIn.AutoLayout
! ConstraintManager	Add-on	Mga.AddOn.ConstraintManage
MONTraverser	Interpreter	MGA.Interpreter.MONTraverse
PatternProcessor	Interpreter	MGA.Interpreter.PatternProces
Table Editor	Plug-in	MGA.PlugIn.Table Editor

Register: Systemwide [✓ For user only] Both

Show components: Active [✓ Active&Inactive] All

Figure 2-1



Please select the component you want to work with

Name	Type	ProgID
AutoLayout	Plug-in	MGA.PlugIn.AutoLayout
! ConstraintManager	Add-on	Mga.AddOn.ConstraintManage
MONTraverser	Interpreter	MGA.Interpreter.MONTraverse
NewWeaver	Plug-in	MGA.PlugIn.NewWeaver
PatternProcessor	Interpreter	MGA.Interpreter.PatternProces
Table Editor	Plug-in	MGA.PlugIn.Table Editor

Register: Systemwide [✓ For user only] Both

Show components: Active [✓ Active&Inactive] All

Figure 2-2

Step 5:

- The C-SAW button on the GME tool bar, installation is done. See Figure 2-3.

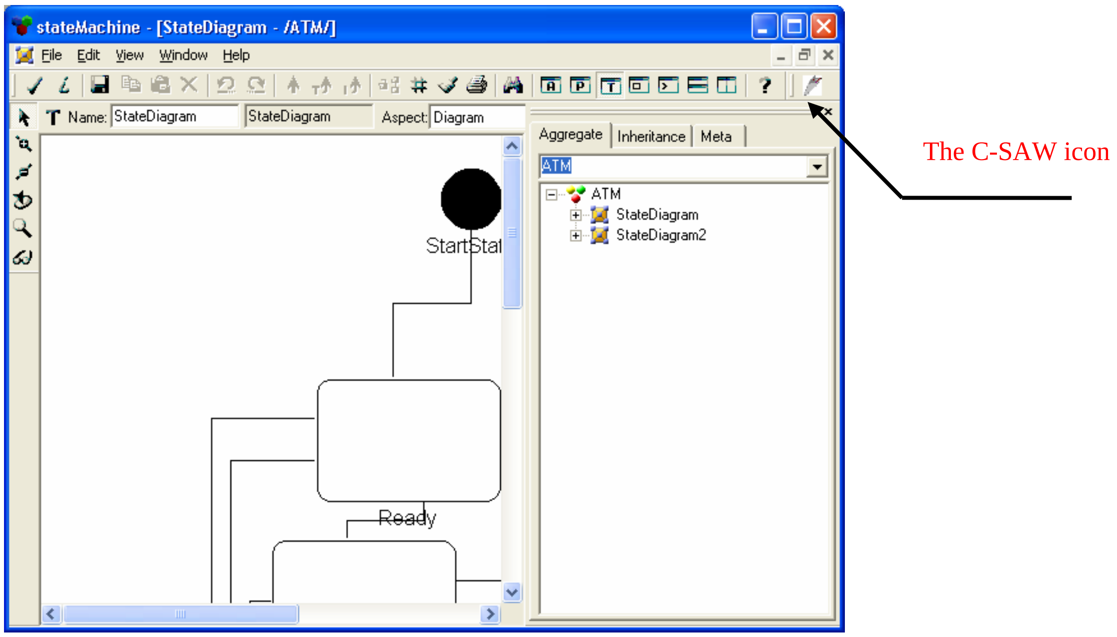


Figure 2-3

3 Using C-SAW

3.1 Introduction

To use C-SAW to perform model transformations, first of all, you need to develop source models in a GME project and write ECL transformation specifications that describe the transformation task(s). After the source models and model transformation specifications are developed, the following steps need to be taken when using C-SAW.

- Open your GME project that contains your source models.

If the C-SAW icon is not visible on the GME toolbar, you need to register it within your project (See Chapter 2 for installation details). If you open a model in GME, which is called the focused model, is the default entry point of a model transformation, i.e., the initial context where transformation starts. However, you can change the entry point by specifying any target object explicitly in the aspect specification. For example, you can specify the root folder or a collection of models as the initial transformation context.

- Click the C-SAW icon on the GME toolbar, and follow the prompt to open your specification file and then the specification is executed automatically by C-SAW.

After you finish executing a specification file, and you want to transform another one, you need to close the project and then open it again before opening another specification file.

3.2 Step-by-step Guide via a State Machine Example

This example shows how to modify an attribute of an ATM model, which is an instance model of the state machine metamodel. In this scenario, the source model (as input to C-SAW) is the ATM model; and the target model (as output of C-SAW) is the modified ATM model. A simple ECL specification is developed to describe such a transformation. C-SAW executes this specification to modify the attribute of the ATM model.

1. Open the project in GME that contains the ATM model.

If you also open the ATM model after opening the project, the open ATM model will be the default entry point of the transformation unless you specify any other model(s) or atoms explicitly by model querying. In this example, the open ATM model is implicitly as the default entry point, from which the CardInserted atom is selected as the application context of the ModifyAttr strategy. Originally, the text attribute of the CardInserted atom is “Please Input Pin” as shown in Figure 3-1.

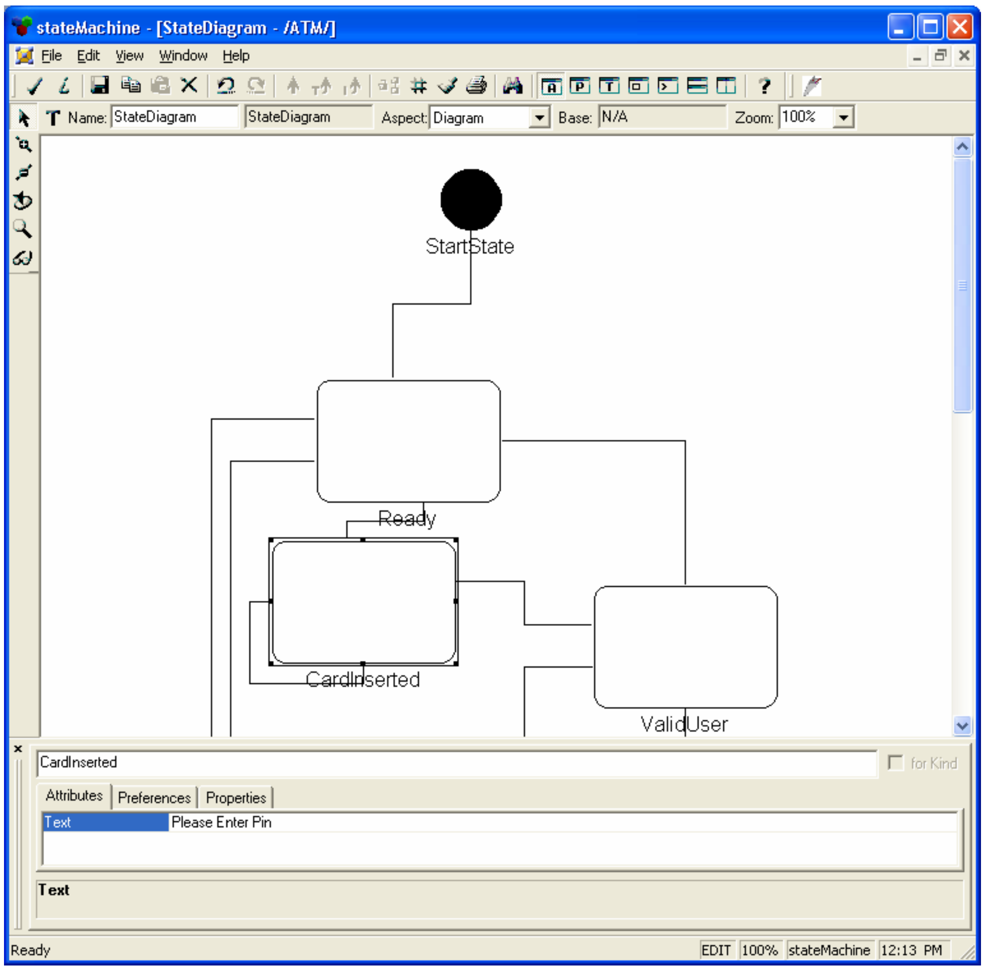
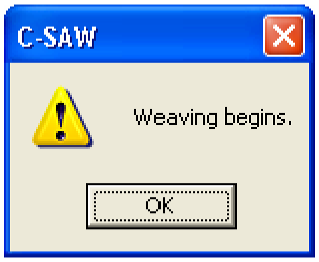


Figure 3-1

2. Invoke the C-SAW engine.

Click on the C-SAW icon on the toolbar to invoke the C-SAW engine. If the icon is not visible, you need to register it. After you click on the C-SAW icon, you can see the prompt as shown in Figure 3-2 and then click ok.


C-SAW

Weaving begins.

Figure 3-2

3. Open the specification file.

Go to the folder you store your specifications and select the specification file as shown on Figure 3-3. The specification file is called modifyAttribute.spc, which is shown in Listing 3-1.

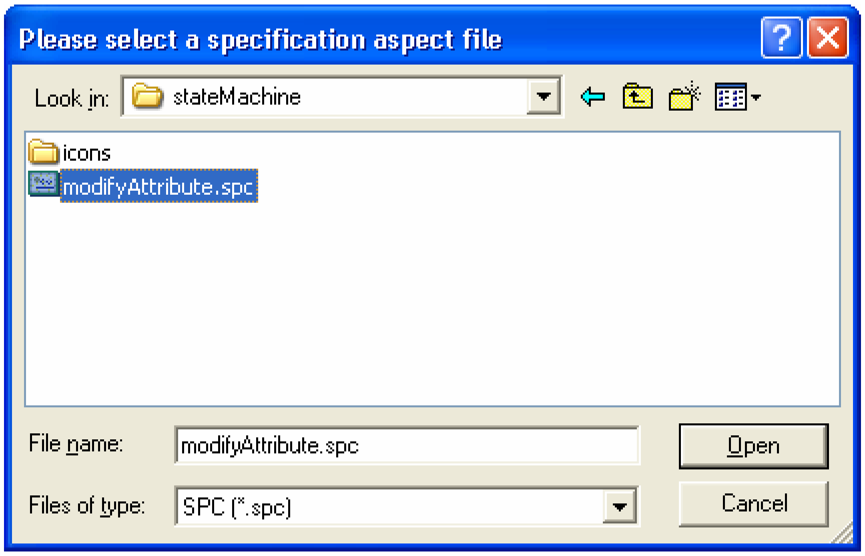

Please select a specification aspect file
Look in: stateMachine
icons
modifyAttribute.spc
File name: modifyAttribute.spc
Files of type: SPC (*.spc)
Open
Cancel

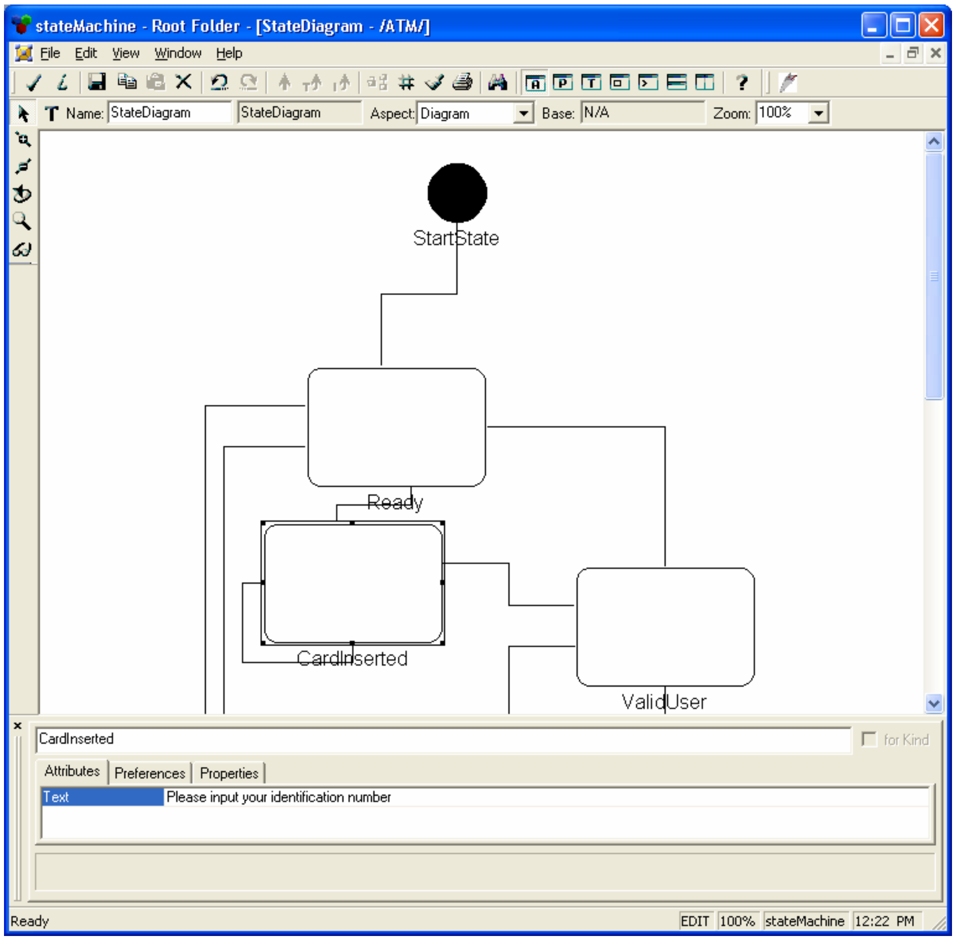
Figure 3-3

```
defines ModifyAttr, PickAState;

strategy ModifyAttr ( )
{
   declare newText : string;

   newText := "Please input your identification number";
   setAttribute("Text", newText);
}

aspect PickAState()
{
   atoms("State")->select(m|m.name() == "CardInserted")->ModifyAttr ( );
}
```

Listing 3-1

4. C-SAW executes the open specification file.

The original ATM model is modified by C-SAW by executing the specification. As shown in Figure 3-4, the text attribute of the CardInsterted atom is changed to “Please input your identification number”.



CardInserted

Text	Please input your identification number

Figure 3-4

4 The Embedded Constraint Language (ECL)

4.1 Overview

The transformation language provided by C-SAW is a textual script-like language, called the Embedded Constraint Language (ECL). The ECL is an extension of Object Constraint Language (OCL), and provides many of the common features of OCL, such as arithmetic operators, logical operators, and numerous operators on collections (e.g., *size* and *select*). ECL also provides special operators to support model aggregates (e.g., *models* and *atoms*), connections (e.g., *source* and *destination*) and transformations (e.g., *addModel* and *setAttribute*). In general, ECL supports an imperative transformation procedural style. In addition, by providing *strategy* and *aspect* constructs, ECL offers the ability to explore numerous modeling scenarios by considering crosscutting modeling concerns as aspects that can be rapidly inserted and removed from a model. This permits a model engineer to make changes more easily to the base model without manually visiting multiple locations.

4.2 Types and Operators

The data types in ECL include the basic data types (e.g., boolean, integer, real and string) and the GME object types (e.g., atom, model, object and folder) as well as the GME collection types (e.g., atomList, modelList, and objectList). This type set can be extended according to further need. The data types are explicitly used in parameter definition and local variable declaration of a strategy. The narrow type conversion is supported in assignment statements. There are there GME types (i.e., set, reference and connection), which are not explicitly supported in ECL. However, they can be considered as object type. Table 4-1 is an overview of the ECL data types.

Table 4-1 An overview of data types.

Type Keyword	Description
boolean	A boolean can have only one of two values: true or false.
integer	The integer type represents the mathematical natural numbers.
real	The real type represents the mathematical concept of real values.
string	A string is a sequence of characters, written with enclosing double quotes, e.g., “NewGateWay”.
atom	Equivalent to the atom type in GME
model	Equivalent to the model type in GME
atomList	A list of atoms.
modelList	A list of models.
object	Equivalent to the object type in GME
objectList	Equivalent to the object type in GME
folder	Equivalent to the folder type in GME

For detailed information about these types, please reference the GME manual.

4.3 Operators on Basic Data Types

There are standard operators for the boolean, integer and real as well as string types, which are described in the following tables.

Table 4-2 Standard operators for the boolean type.

Operator	**Notation**	**Result type**
or	a or b	boolean
and	a and b	boolean
exclusive or	a xor b	boolean
implies	a implies b	boolean

Table 4-3 Standard operators for the integer and real type.

Operator	**Notation**	**Result type**
equals	a == b	boolean
not equals	a <> b	boolean
less	a < b	boolean
more	a > b	boolean
less or equal	a <= b	boolean
more or equal	a >= b	boolean
plus	a + b	integer or real
minus	a – b	integer or real
multiplication	a * b	integer or real
division	a / b	real
not	-a or not a	integer or real

4.4 Operators on Object and Collection Types

The operators on object and collection types include two groups: one group is used to querying objects and collections (including aggregation and selection operators), the other group is used to alter the models and atoms (called transformation operators).

4.4.1 Model Aggregation and Selection

One common activity during model transformation is to find elements in a model. There are two different approaches to locating model elements. The first approach - querying - evaluates an expression over a model, returning those elements of the model for which the expression holds. The other common approach uses pattern matching where a term or a graph pattern containing free variables is matched against the model. Currently, ECL supports model queries primarily by providing the *select* operator. Other operators include model aggregation operators to select a collection of objects, and a set of operators to find a single object.

The *select(expression)* operator is frequently used in ECL to specify a selection from a source collection and the result is always a subset of the original collection. In addition, model aggregation operators can also be used to perform model querying. For example, *models(expression)* is used to select all the submodels that satisfy the constraint specified by the expression. Other aggregation operators include *atoms(expression)*, *connections(expression)*, *attributes(expression)*, *source()* and *destination()*. Specifically, *source()* and *destination()* are used to return the source object and the destination object in a connection. Another set of operators are used to obtain a single object (e.g., *findAtom* and *findModel*). The following ECL code shows a number of operators just mentioned:

```
rootFolder().findFolder("ComponentTypes").models() ->
                    select(m|m.name().endWith("Impl")) ->AddConcurrency();
```

First, *rootFolder* returns the root folder of a modeling project. Next, *findFolder* is used to return a folder named “ComponentTypes” under the root folder. Then, *models()* is used to find all the models in the “ComponentTypes” folder. Finally, the *select* operator is used to select all the models that match the predicate expression (i.e., those models whose names end with “Impl”). The “AddConcurrency” strategy is then applied to the resulting collection.

4.4.2 Transformation Operators

ECL provides basic transformation operations to add model elements remove model elements and change properties of model elements. Standard OCL does not provide such capabilities because it does not allow side effects on a model. However, a transformation language should be able to alter the state of a model. ECL extends the standard OCL by providing a series of operators for changing the structure or constraints of a model. To add new elements (e.g., a model, atom or connection) to a model, ECL provides such operators as *addModel***,** *addAtom* and *addConnection*. Similarly, to remove a model, atom or connection, there are operators like *removeModel*, *removeAtom* and *removeConnection*. To change the value of any attribute of a model element, *setAttribute* can be used.

4.5 Function Call Expressions

Operators on all the object and collection types define various behaviors to manipulate GME models, which are also called ECL functions and used via function calls. There are two kinds of function calls depending on the type of the caller: 1) caller.functionName(parameters), and 2) caller->functionName(parameters). The former using a dot means the caller is a single object (e.g., an atom, a model or an object); the latter using an arrow means the caller is a set of objects(e.g., an atomList, a modelList or an objectList). Parameters can be none, one or more. For example, the function call "*aModelInstance.atoms()*" returns all the atoms of the model instance; and “*aModelList->size()*” return the number of the models in this model list. Thus, caller constitutes the context of a function call.

When a caller is not specified explicitly, there still exists a default caller, which is the rootModel of the currently focused paradigm; for example, a single statement "*atoms();*"

will return all the atoms of the root model of currently open paradigm although there is not any explicitly specified caller for it, thus it is equivalent to "*self.atoms();*" where self is a keyword that represents the current focused model or atom.

According to all the functions currently supported, the valid caller type can be either an atom/model/object or a list of them or a folder, depending on the definition of each function. The usages and purpose of all the functions are describe in Appendix 2 where some other predefined functions (e.g., *show*) and operators on string type (e.g., *endWith*) are also included.

4.6 Aspect and Strategy

There are two kinds of modular constructs in ECL: *strategy* and *aspect*, which are designed to provide aspect-oriented capabilities in specifying crosscutting modeling concerns. A strategy is used to specify elements of computation and the application of specific properties to the model entities (e.g., adding model elements). A modeling aspect is used to specify a crosscutting concern across a model hierarchy (e.g., a collection of model elements that across a model).

In general, an ECL specification may consist of one or more strategies, and a strategy can be called by other strategies. A strategy call implements the binding and parameterization of the strategy to specific model entities. The context of a strategy call can be an entire project; a specific model, atom, or connection; or a collection of assembled modeling elements that satisfy a predicate. The aspect construct in ECL is used to specify such a context.

In a procedural style transformation, an aspect is the start point of a transformation process. Currently, it doesn't support parameter binding, local variables definition and assignment statements as a strategy does. Inside an aspect body, it is typically a statement that calls a strategy. By default, this strategy is applied to the default entry point (e.g., the currently open model in GME). If there is a querying statement preceded with the strategy call, the strategy call will be applied to the querying result. For example, "*atoms()->select(m|m.name() == "LogOnWrite").DoSomething();*" the DoSomething() strategy will be applied to the atom named "LogOnWrite".

Usually, a strategy is called within an aspect. Additionally, a strategy can be called within another strategy. Inside an aspect or a strategy, there should be at least one statement (e.g., you can put a show statement to print out something). In other words, the body of an aspect or a strategy can't be blank, otherwise a syntactical error occurs.

4.7 Learn to Write ECL Specifications via an Example

Embedded System Modeling Language (ESML) is used to model real-time mission computing embedded avionics applications. ESML provides the following modeling categories to allow representation of an embedded system: 1) Components, 2) Component Interactions, and 3) Component Configurations.

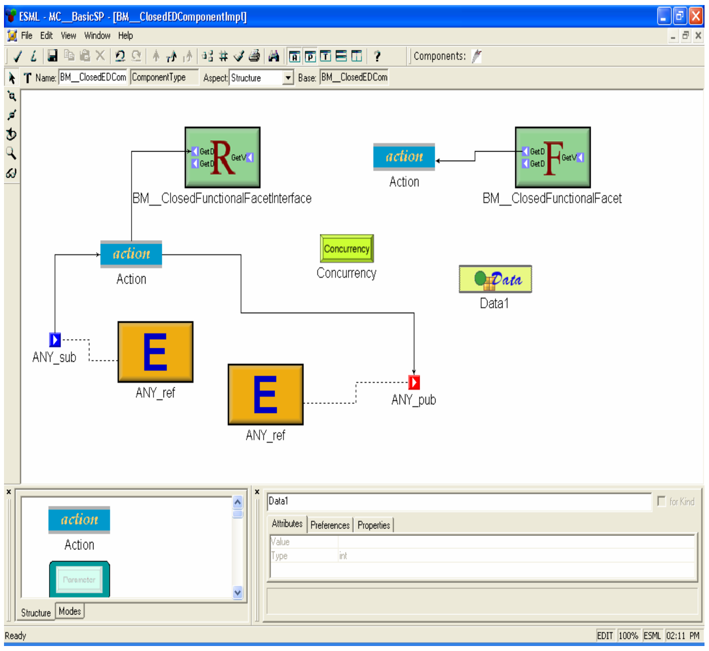
Bold Stroke is a product-line architecture written in several million lines of C++ that was developed by Boeing in to support families of mission computing avionics applications for a variety of military aircraft. There are over 20 representative ESML models for all of the Bold Stroke usage scenarios that have been defined by Boeing. For each specific scenario within Bold Stroke, the components and their interactions via an event channel are captured by ESML models. For example, the internal configuration of a component is specified with data elements and event types as well as communication ports (e.g., the facet/receptacle descriptors for communicating events). An example of transforming Bold Stoke ESML models to support a black box data recorder is illustrated as below.

In avionics systems, an essential diagnostic tool for failure analysis is a "black box" that records important flight information. This device can be recovered during a failure, and can reveal valuable information even in the event of a total system loss. Currently, the Bold Stroke ESML component models do not represent the black box flight recorder concern. To add this concern to corresponding models, a log atom can be used to specify different logging policy such as "Record the values upon <entry/exit> of <a set of named methods>" or "Record the value upon every update to the <data variable>." A model transformation is needed to weave black box data recorder representations (i.e., the log atoms in ESML) into these ESML component models. The requirement of this transformation task can be briefly described as follows:

```
For all models which type is component,
    Find all the data atoms,
   For each data atom,
       add a log atom and proper connection
```

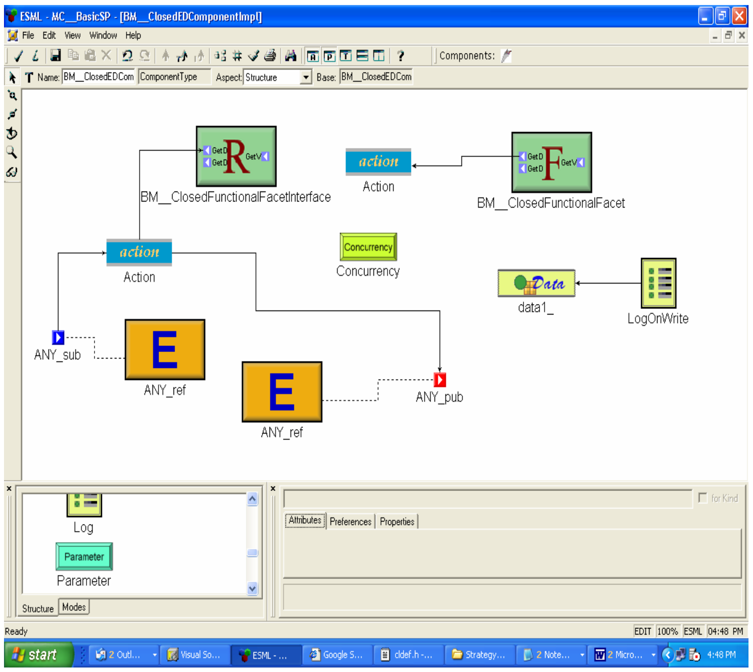
Figure 4-1 shows an ESML component model without any log atom. To manually add log atoms into over 20 existing component models will be time consuming and susceptible to errors. C-SAW automates this transformation based on the ECL transformation specification shown in Listing 4-1.

The above specification written in ECL supports the requirement "Log every time and in every component that an update occurs on the data1 field". The Start aspect selects all the component models whose name ends with "Impl" in the ComponentTypes folder. The FindData strategy finds all of the "Data" atoms in every component and then applies the AddLog strategy on each of them. Inside the AddLog strategy, parentModel is the model to which the data atom belongs. A new log atom is created in this model and set "OnWrite" to its Kind attribute and the "MethodList" attribute as "Update". Finally, it connects this new "Log" atom to its corresponding "Data" atom. As a result, after using C-SAW to apply this ECL specification, "LogOnWrite" atoms will be inserted into each component that has a "Data" atom as shown in Figure 4-2.



ESML - MC__BasicSP - [BM__ClosedEDComponentImpl]
File Edit View Window Help
Components:
Name: BM__ClosedEDCom ComponentType Aspect: Structure Base: BM__ClosedEDCom
BM__ClosedFunctionalFacetInterface
Action
BM__ClosedFunctionalFacet
Action
Concurrency
Data1
ANY_sub
ANY_ref
ANY_ref
ANY_pub
Action
Parameter
Structure Modes
Data1
for Kind
Attributes Preferences Properties
Value
Type int
Ready
EDIT 100% ESML 02:11 PM

Figure 4-1



ESML - MC__BasicSP - [BM__ClosedEDComponentImpl]
File Edit View Window Help
Components:
Name: BM__ClosedEDCom ComponentType Aspect: Structure Base: BM__ClosedEDCom
BM__ClosedFunctionalFacetInterface
Action
BM__ClosedFunctionalFacet
Action
Concurrency
data1_
LogOnWrite
ANY_sub
ANY_ref
ANY_ref
ANY_pub
Log
Parameter
Structure Modes
for Kind
Attributes Preferences Properties
Ready
EDIT 100% ESML 04:48 PM
start
4:48 PM

Figure 4-2

```
strategy FindData1( ) {
            atoms()->select(a | a.name() == "Data1")->AddLog();
}
strategy AddLog( ) {
    declare parentModel : model;
    declare dataAtom, logAtom : atom;
    dataAtom := self();
    parentModel := parent();
    logAtom := parentModel.addAtom("Log", "LogOnWrite");
    logAtom.setAttribute("Kind", "OnWrite");
    logAtom.connectedTo(dataAtom);
}
aspect Start( ) {
  rootFolder( ).findFolder("ComponentTypes").models()->FindData1();
}
```

Listing 4-1

5 Appendix 1: ECL BNF Specification

```
name : id:IDENT;

defines : { DEFINES defs  SEMI };

defs :   def ( COMMA def )*;

def : name ;

cpars : cpar ( SEMI cpar )* ;

cpar : name ( COMMA name )* ":" name;

cdef : { INN foldername { DOT modelname { DOT aspectname }}} (STRATEGY name
| ASPECT name )
            LPAR  { cpars } RPAR
            priority
            description
            LBRACE cexprs  { action  } RBRACE
        | FUNCTION name LPAR  { cpars } RPAR cexprs  { action  } ;

priority : PRIORITY "=" id:INTEGER | ;

description : id:STR | ;

foldername : name ;

modelname :  name ;

aspectname : name ;

lval : name;

astring  : str:ACTION ;

action  : astring;

cexprs : cexpr SEMI ( cexpr SEMI )* ;

cexpr : { action } ( assign | DECLARE STATIC cpar
                              | DECLARE cpar
                           | lexpr) ;

assign : lval ASSIGN lexpr ;
```

```
ifexpr : IF cexpr
            THEN cexprs { action }
            { ELSE cexprs  { action } }
            ENDIF;

lexpr :  relexpr lexpr_r ;

lexpr_r: { lop  relexpr  lexpr_r }        ;

relexpr: addexpr  { rop addexpr }      ;

addexpr           : mulexpr  addexpr_r   ;

addexpr_r :{   addop  mulexpr  addexpr_r }   ;

mulexpr           : unaexpr  mulexpr_r   ;

mulexpr_r :{  mulop unaexpr  mulexpr_r      }          ;

unaexpr           : ( unaryop  postfixexpr  ) | postfixexpr          ;

postfixexpr:     primexpr  postfixcall   ;

postfixcall  :    { (  (   DOT | ARROW | CARET )  call )  postfixcall }         ;

primexpr : litcoll   |  lit |          objname callpars | LPAR cexpr RPAR | ifexpr ;

callpars : {  LPAR  ( ( decl ) ? decl  { actparlist  }  |  { actparlist  }  )   RPAR } ;

lit        :          string   | number ;

tname  : name ;

litcoll    : coll  LBRACE { cexprlrange } RBRACE   ;

cexprlrange :   cexpr   {   (   (COMMA cexpr )+   )  |  ( ".." cexpr ) }           ;

call  :   name  callpars ;

decl     :    name     ( COMMA name  )*     { ":" tname } "\|";

objname           :    name | SELF ;

actparlist :    cexpr     ( COMMA cexpr  )*    ;
```

```
lop       :    AND | OR | XOR | IMPLIES         ;

coll  :    SET    | BAG  | SEQUENCE | COLLECTION          ;

rop        :    "==" | "<" | ">" | ">=" | "<=" |       "<>"    ;

addop > :    PLUS | MINUS    ;

mulop >  :    MUL | DIV ;

unaryop :    MINUS | NOT     ;

string :     str:STR   ;

number :    r:REAL | n:INTEGER      ;

}

#token STR                          "\"~[\"]*\""
#token INTEGER                      "[0-9]+"
#token  REAL            "([0-9]+.[0-9]* | [0-9]*.[0-9]+) {[eE]{[\-\+]}[0-9]+}"
#token IDENT            "[a-zA-Z][a-zA-Z0-9_]*"
```

6 Appendix 2: ECL Function Interfaces

name
purpose: return a caller's name
caller: an atom, a model or an object
usage:caller.name().
result type: string

kindOf
purpose: return an caller's kindname.
caller: an atom, a model or an object
usage:caller.kindOf().
result type: string

size
purpose: return the size of the caller list.
caller: atomList, modelList or an objectList.
usage:caller.size().
result type: int

isNull
purpose: determine if the caller is null.
caller: an atom, a model or an object
usage:caller.isNull().
result type: boolean

models
purpose: return all the models or the models with specific kindName within a caller.
caller : a model or a folder.
usage:caller.models() or caller.models(string kindName).
result type: modelList

modelRefs
purpose: return all the models within a caller that are referred by the model references with the specific kindName.
caller: a model or an object that represents a model.
usage: caller.modelRefs(string kindName).
result type: modelList

atoms
purpose: return all the atoms or the atoms with specific kindName within a caller.
caller: a model or an object that represents a model.
usage:caller.atoms() or caller.atoms(string kindName).
result type: atomList.

select
purpose: select the atoms or the models within a caller according to the specified condition.
caller: atomList, modelList or an objectList
usage:caller.select(logic expression);
for example, atoms("")->select(m|m.kindOf() == "State").
result type: atomList or modelList.

parent
purpose: return the parent model of a caller.
caller: a model, an atom or an object that represents a model/an atom.
usage: caller.parent()
result type: model

refersTo
purpose: return a model/an atom/an object that the caller refers to.
caller: a modelRef or an object that represents a model reference.
usage: caller.referesTo( )
result type: model or atom or object

connections
purpose: return all the connections with a specific connection kindname within a model.
caller: a model or an object that represents a model.
usage: caller.connections(string connName)
result type: objectList that represents a list of connections.

destination
purpose: return the destination point of a connection.
caller: a connection.
usage: caller.destination().
result type: model/atom/reference/object

source
purpose: return the source point of a connection.
caller: a connection.
usage: caller.source().
result type: model/atom/reference/object

addConnection
purpose: add a connection with a specific kindName from a source object to a destination object within a caller..
caller: a model or an object that represents a model.
usage: caller.addConnection(string kindName, object source, object destination)
result type: connection

findConnection
purpose: find a connection with a specific kindName from a source object to a destination object within a caller..
caller: a model or an object that represents a model.
usage: caller.findConnection(string kindName, object source, object destination)
result type: connection

removeConnection
purpose: remove a connection with a specific kindName from a source object to a destination object within a caller..
caller: a model or an object that represents a model.
usage: caller.removeConnection(string kindName, object source, object destination)
result type: void

rootFolder
purpose: return the root folder of an open GME project.
caller: none.
usage: rootFolder()
result type: folder

findFolder
purpose: return a folder with a specific name.
caller: a folder.
usage: caller.findFolder(string folderName)
result type: folder

addFolder
purpose: add a folder based on its kindName and assign a new name to it.
caller: a folder
usage: caller.addFolder(string kindName, string newName)
result type: folder

addSet
purpose: add a set based on its kindName and assign a new name to it within a caller or a list of callers.
caller: a model or a list of models.
usage: caller.addSet(string kindName, string newName) or caller->addSet(string kindName, string newName)
result type: set

addMember
purpose: add an object as a member of a set.
caller: an object that represents a set.
usage: caller.addMember(object anObj)
result type: void

addReference
purpose: add a reference with a specific kindName that refers to an object and assign a new name to it within a caller.
caller: a model.
usage: caller.addReference(string kindName, object refTo)
result type: reference

findAtom
purpose: return an atom based on its name within a caller.
caller: a model or an object that represents a model.
usage: caller.findAtom(string atomName)
result type: atom

addAtom
purpose: add an atom based on its partName(i.e., kindName) that belongs to a model and assign a new name to it.
caller: a model or an object that represents a model.
usage: caller.addAtom(string partName, string newName)
result type: atom

removeAtom
purpose: remove an atom based on its name.
caller: a model or an object that represents a model, which is the parent model of the to-be-removed atom.
usage: caller.removeAtom(string atomName)
result type: void

addModel
purpose: add a model based on its partName(i.e., kindName) that belongs to a model and assign a new name to it.
caller: a model, a folder, an object that represents a model/folder or a list of models.
usage:
1) if caller is a single object, caller.addModel(string partName, string newName);
2) if caller is a list, caller-> addModel(string partName, string newName)
result type: model

findModel
purpose: return a model based on its name.
caller: a folder or a model or an object that represents a model.

usage: caller.findModel(string modelName)
result type: model

removeModel
purpose: remove a model based on its name.
caller: a model or an object that represents a model, which is the parent model of the to-be-removed model.
usage: caller.removeModel(string modelName)
result type: void

findObject
purpose: return an object based on its name within a caller.
caller: a model or a folder, or an object that represents a model/folder.
usage: caller.findObject(string objName)
result type: object

getAttribute
purpose: return the value of an attribute of a caller which type is int, double or string.
caller: an atom, a model or an object.
usage: caller.getAttribute(attrName)
result type: int, double or string

intToString
purpose: convert an integer value to string
caller: none.
usage: intToString(int val)
result type: string

setAttribute
purpose: set a new value to an attribute of a caller
caller: an atom, a model or an object.
usage: caller.setAttribute(string attrName, anyType value)
result type: void

show
purpose: display string message
caller: none
usage: show(any expression that returns a string)
result type: void

endWith
purpose: check if a string ends with a substring
caller: a string
usage: caller.endWith(string aSubString)
result type: boolean.

Notes:

1) Any find operation (e.g., findAtom, findModel, findConnection) will return null if it can not find the atom or model or connection.
2) Combination of a find operation and isNull operation can be used to determine whether a model/atom/connection exists or not.