TECHNICAL REPORT

IST-MBT-2012-01

The Argos Manual

for Argos version: 0.39

Stefan Tiran

stiran@ist.tugraz.at

March 2012

Institute for Software Technology (IST)
Graz University of Technology
A-8010 Graz, Austria

Contents

1 Introduction **3**

2 How To Run Argos **4**
- 2.1 Program Invocation . . . 4
- 2.2 Options and Arguments . . . 4
- 2.3 Examples . . . 5

3 OOAS Language **5**
- 3.1 Base structure . . . 5
- 3.2 The Type Definition Block . . . 6
  - 3.2.1 Defining Integer Types . . . 6
  - 3.2.2 Defining Enum Types . . . 7
  - 3.2.3 Defining Complex Types (Lists, Tuples) . . . 7
  - 3.2.4 Defining Classes . . . 7
- 3.3 Structure of Classes . . . 8
  - 3.3.1 Inheritance . . . 8
  - 3.3.2 Defining Variables (Attributes) . . . 8
  - 3.3.3 Creating Objects . . . 8
  - 3.3.4 Defining Methods . . . 8
  - 3.3.5 Defining Actions . . . 9
- 3.4 Composition . . . 10
  - 3.4.1 Sequential Composition . . . 10
  - 3.4.2 Non-deterministic Choice . . . 10
  - 3.4.3 Prioritizing Composition . . . 10
- 3.5 Expressions . . . 11
  - 3.5.1 Conditional Assignment . . . 11
  - 3.5.2 Fold . . . 11
  - 3.5.3 Quantifier Expressions . . . 11
  - 3.5.4 List Comprehension . . . 11
  - 3.5.5 Tuple Expressions . . . 12
  - 3.5.6 Access Expressions . . . 12
  - 3.5.7 Cast Expressions . . . 13
- 3.6 Operators . . . 13
  - 3.6.1 Numeric Operators . . . 13
  - 3.6.2 Logical Operators . . . 13
  - 3.6.3 Comparison Operators . . . 13
  - 3.6.4 List Operators (Head, Tail, Length, Concatenation) . . . 14
- 3.7 The Dood Block . . . 15
- 3.8 The System Assembling Block . . . 16
- 3.9 Comments . . . 17
- 3.10 Formal Syntax . . . 17

4 Incomplete Features 19
4.1 Sequential Block Expression . . . 19
4.2 String . . . 19

5 CADP Target 19
5.1 The Argos CADP-Target . . . 19
5.2 Generating BCG . . . 20
5.3 Animator . . . 20

6 Alarm System 20
6.1 Overview . . . 20
6.2 Requirements . . . 21
6.3 State Chart . . . 21
6.4 Object-Oriented Action System . . . 22
6.4.1 Interface . . . 22
6.4.2 Time Handling . . . 22
6.4.3 Protocol Layer . . . 22
6.4.4 Non-determinism . . . 22
6.4.5 Code . . . 23
6.5 Labeled Transition System . . . 26

List of Figures

1 HelloWorld . . . 6
2 Creating an object . . . 9
3 Defining sum of all list elements using the fold meta function . . . 12
4 Prime number calculation . . . 13
5 Tuple Example . . . 14
6 Cast Example . . . 15
7 Stack . . . 16
8 State Chart of Car Alarm System . . . 21
9 Labeled Transition System . . . 27

1 Introduction

Argos is a compiler for Object-Oriented Action Systems (OOAS). It has been developed by Willibald Krenn from Graz University of Technology within the MOGENTES project.

Its main target is producing Action Systems in the proprietary format that *Ulysses* can process. *Ulysses* is an ioco-checker written by Harald Brandl also within the MOGENTES project.

2 How To Run Argos

Argos is a .NET based command line program. To run it you need either the .NET Framework or mono (`http://www.mono-project.com`).

2.1 Program Invocation

Since Argos is a command line tool you need to open a command line prompt (using Windows) resp. a shell (using a UNIX-like system).

When using Windows change to the Argos directory and simply run argos.exe:

```
c:\Argos \> argos
```

When using a UNIX-like system run mono with argos.exe as parameter:

```
$ mono argos.exe
```

In both cases one should now see this welcome screen:

```
          >> Mogentes OO-Action System Parser <<

          Version: 0.39 (16471)
  Grammar-Version: 0.06

Usage:    argos [options] <oo-actionsystem-file>
Options:
     -o<name>   ... write output to file <name> (overwrites!)
     -n<name>   ... use namespace <name> (default: as)
     -d<depth>  ... set max. search depth (default: 35)
     -q         ... "quiet" mode: doesn't print Warnings to console

     -a         ... create pseudo-action system code
     -p         ... create prolog code
     -c         ... create CADP implicit LTS
```

2.2 Options and Arguments

As can be seen in the welcome screen for translating Action Systems the last parameter has to be the OOAS-File you want to convert, every other options has to come before.

If no other option is given, Argos will translate the given OOAS-File to C#-Pseudo-Code and print it out on the screen. Since this most probably won't be something you will find extremely useful, you should at least specify an output-file and the desired language.

For specifying the output-file use option `-o` followed by the file name resp. the relative path of the file you want to create. There should be NO space between `-o` and the file name!

For specifying the language include option `-c` for creating C-Code for the CADP-Toolbox resp. option `-p` for creating Prolog-Code for use with the Ulysses tool.

When creating Prolog-Code the following options are important:

-n With this option one can specify the namespace within the Prolog-File; the only meaningful use of this option is `-nasm` for letting the Ulysses tool know, that this file represents the mutant rather then the original file.

-d With this option one can specify the max. depth, to which the Ulysses tool will search for differences. Please note that this option has to be followed by a number WITHOUT a space before! The default value is 35, so leaving out this option is equivalent to writing `-d35`.

2.3 Examples

To illustrate the use of the options, we hereby provide the following examples:

- Translate an original OOAS specification `original.ooas` to a Prolog file `original.pl` for the Ulysses tool with max. search depth 42:

  ```
  argos.exe -p -d42 -ooriginal.pl original.ooas
  ```

- Translate a mutated OOAS specification `mutant.ooas` to a Prolog file `mutant.pl` for the Ulysses tool with max. search depth 42:

  ```
  argos.exe -p -d42 -nasm -omutant.pl mutant.ooas
  ```

- Translate OOAS specification `original.ooas` to a C file `original.c` for the use with the CADP toolbox:

  ```
  argos.exe -c -ooriginal.c original.ooas
  ```

3 OOAS Language

The Argos tool will only work if the given file is well formed according to the language that is described in this section. The language is based on the language proposed in [2] but only a subset of features is actually implemented.

3.1 Base structure

To illustrate the base structure of an OOAS file let us consider the simple Hello World program in Figure 1.

Every OOAS consists of two different regions: The type definition block and the system assembling block. In the HelloWorld system the type definition block consists of Lines 1 - 13 and the system assembling consists of Lines 14 and 15.

Let us now discuss the system Line by Line: Line 1 just consists of the keyword `types` stating that here begins the type definition block. In Line 2 a class called `Greeter` is defined. The keyword `autocons` states that one instance of the class will automatically be created at system start. This instance is called the "root object". The symbols in Line 3 and 13 state the begin and the end of the class definition. In Lines 4 and 5 there is

```
 1 types
 2   Greeter = autocons system
 3   |[
 4   var
 5         done : bool = false
 6   actions
 7         obs HelloWorld = requires done = false :
 8                 done := true
 9                 end
10   do
11         HelloWorld
12   od
13   ]|
14 system
15         Greeter
```

Figure 1: HelloWorld

a variable definition block. Even though this block is not formally necessary for Argos to compile a file, Argos will not create a valid prolog file if this block is missing. In Lines 6 - 9 there is the action block consisting of the action `HelloWorld`. The keyword `obs` states that this action is *observable* rather then *controllable* (keyword: `ctr`) or internal (no keyword). The keyword `requires` states the beginning of the guard, which enables the action. The body of this action consists of Lines 8 and 9. Line 8 contains an assignment, the keyword `end` in Line 9 marks the end of the action. Lines 10 - 12 contains the so-called **do od**-block . The **do od**-block is described in Section 3.7.

As already mentioned, Lines 14 + 15 contain the system assembling block. In this block the system is composed of its classes. Since this example system only contains of one class, the block is quite simple and needs no further explanation.

3.2 The Type Definition Block

The type definition block has two purposes: As we have already seen it can be used for defining the classes of which the system consists of. However it can also be used for creating named types which is similar to the use of `type-def` in C.

The latter use is quite important as OOAS has a very strict typing system. For example if you want to define a variable as integer, you also have to define the range of the values. This is due to the fact that too wide ranges can lead to enormous efficiency problems when simulating the model.

3.2.1 Defining Integer Types

Integer types can be defined by the lower and upper range.

Consider you want to define a type `MyInt` that can contain values between 0 and 31, then you can write

```
MyInt = int [0..31];
```

Please note that the semicolon has to be placed if and only if this is not the last definition.

3.2.2 Defining Enum Types

Enums can be defined by enumerating the possible values and optionally assigning them integer values. If integer values are explicitly assigned, enum types can be casted to integer types.

Consider you want to define an enum type `Color` that can contain the values `red`, `green`, `blue`, `yellow` and `black`, then you can write:

```
Color = {red, green, blue, yellow, black};
```

The possibility to explicitly assign integer values is especially useful when dealing with equivalent classes of integers. Consider you need an integer type that can only contain small number of different values but these values define a range with "holes" rather then a complete one.

So in case you need a type that can hold the values 1, 42 and 1337 you can write

```
  CoolValueSet = {CoolValue1 = 1, CoolValue2 = 42, CoolValue3 = 1337};
```

3.2.3 Defining Complex Types (Lists, Tuples)

Additional to the former so-called simple types the OOAS language also supports so-called complex types. These are: lists and tuples.

A list is defined by the number and the type of its elements.

An example for the former variant would be:

```
MyList = list [42] of char;
```

A tuple is defined by the types of its elements. Unlike a list the types can differ among the different elements.

An example would be:

```
MyTuple = (MyInt, SmallInt);
```

Even though maps and qualitative reasoning types would be specified by the OOAS language, Argos does not support them yet.

3.2.4 Defining Classes

In the OOAS language the definition of a class (ooActionSystem) does not differ from the definition of any other type.

3.3 Structure of Classes

3.3.1 Inheritance

As common in Object-Oriented languages, classes can be derived from other classes and inherit their behavior. A sample class `Desc` inheriting from a class `base` could look like this:

```
Desc = system (base)
|[
]|
```

3.3.2 Defining Variables (Attributes)

Variables store the information of the state of an object. An alternative name of variables often used in error messages is "attributes". Variables can be static and have to be initialized with a valid value. For objects there is a predefined object `nil`. There is also a self-reference called `self`.

The definition of a variable can look like this:

```
      done : bool = false;
```

Please note that the semicolon has to be placed if and only if this is not the last definition.

3.3.3 Creating Objects

The only place where an object can be created is in the variable definition block. Line 5 in Figure 2 demonstrates how a variable with the name `myOtherClassObject` can be initialized with an object with the name `OtherClassObject` of a class `OtherClass`.

For details on the System Assembling Block (Lines 23 and 24) see Section 3.8.

3.3.4 Defining Methods

Methods can be used for defining subroutines. Methods can have parameters, variables and a return value. Methods can be called from actions and from other methods, but not recursively - neither directly nor indirectly.

The body of a method isn't any different from the body of an action and therefore described in Section 3.3.5.

A sample method, that returns the bigger of two values could look like this:

```
maxValue(a : MyInt, b : MyInt) : MyInt =
  result := if (a > b) then a else b end
end;
```

```
types
  RootClass = autocons system
  |[
  var
        myOtherClassObject : OtherClass = new(OtherClass,"OtherClassObject") ;
        initialized : bool = false
  actions
        obs init = requires initialized = false : initialized := true end
  do
        init ()
  od
  ]| ;
  OtherClass = system
  |[
  var
        initialized : bool = false
  actions
        obs init = requires initialized = false : initialized := true end
  do
        init ()
  od
  ]|
system
        RootClass [] OtherClass
```

Figure 2: Creating an object

Returning a value is done by assigning it to the variable result. For a list of possible expressions we refer to Section 3.5.

One important aspect of methods is their pureness. A method is pure iff there is no assignment to a variable other than `result`. This property is checked on compile time. Only if a method is pure it can be used in a guard or in a list comprehension.

3.3.5 Defining Actions

Named actions consist of an optional parameter list, a variable definition block and a body but they lack a return value. Named actions can only be called in the so-called **do od**-block .

A named action can be controllable (keyword `ctr`), observable (keyword `obs`) or internal (no keyword).

The definition of local variables does not differ from any other variable definition so we refer to Section 3.3.2.

The body of a named action can consist of one or more basic actions. Every action has a guard which "enables" it. The general form of a guard is `requires p :` S_1 `end` where p is the condition and S_1 is the action. The condition can be any expression that doesn't involve the calling of a non-pure method or a non-pure assignment.

For further details about expressions see Section 3.5!

Basic actions can be the built-in action `skip`, single or multiple assignment or a sequential composition of two other actions. For details on the composition we refer to Section 3.4.

The built-in action `skip` doesn't do anything at all. Sometimes this is called *identity assignment.*

Single assignment has the simple form `v := e` where v is a variable and e is an expression.

However you can also assign multiple expressions to multiple variables by separating them with commas.

So if `done` and `multi` are both variables of type `bool`, then you can write:

```
done, multi := true, true
```

3.4 Composition

3.4.1 Sequential Composition

The easiest way of composing two actions is the sequential composition. It has the general form `A ; B` where A and B are both actions and has the meaning that iff A and B are *enabled*, first action A is executed and then action B is executed. Note that it is not necessary for B to be enabled before A. It is also possible that B becomes enabled because of A.

Please note that despite of the apparent resemblance with the semicolon as marker of the end of a line in many imperative programming languages there are two very important differences:

The first difference is that since it is a composition operator, there must not be a semicolon after the last action. This difference seems to be more on a syntactical level, but should also warn about the other very important difference:

The second difference is that since actions have guards, no action of a sequence block is executed, if a single action is not enabled. This also means that an action can hinder actions to be executed, which were called *before*!

3.4.2 Non-deterministic Choice

The counterpart of the "sequential composition" is the so called "non-deterministic choice".

The general form of a non-deterministic choice is `A [] B` where A and B are both actions and the meaning is that either A or B is executed. However an action can only be chosen if it is enabled. According to the theory of action systems an action has to be chosen if it is aborting. However there is no abort feature implemented yet.

3.4.3 Prioritizing Composition

The "prioritizing composition" is rather an abbreviation than an actual new type of composition.

The general form is `A // B` and it means that if A is enabled, it is chosen and otherwise B is chosen.

3.5 Expressions

Expressions can be evaluated and their values can be assigned to variables.

3.5.1 Conditional Assignment

Conditional Assignments are expressions. They can not be used for branching the control sequence. Branching the control sequence has to be done by using guards and non-deterministic choice.

A conditional assignment can look like this:

```
c := if (a > b) then a else b end
```

3.5.2 Fold

For the purpose of iterating through a list in OOAS *fold* is used. *Fold* is a higher-order function well known from the functional programming paradigm.

In the OOAS language both *right fold* and *left fold* are defined. The symbol for the former is `:>:` whereas the symbol for the latter is `:<:`. However in Argos only *right fold* is implemented.

The general form of a fold expression is `method :: (init_expression) :>: (list_expression)` where `method` is the name of the method, `init_expression` is the value for the last parameter of the method for the first call. For any other call the result of the former call is used instead. Finally `list_expression` is the expression that evaluates to the list to be iterated.

Figure 3 shows how to define a method `Sum` using a method `Add` and the *fold* operator.

3.5.3 Quantifier Expressions

The OOAS language allows to use the quantifiers `forall` and `exists`. The resulting expressions will evaluate to Boolean values. The general form is `(quantifier : Type : (logical sentence))`.

For example an existential quantification can look like this: `(exists i : LengthInt : (primes[i] = a))`.

3.5.4 List Comprehension

The OOAS language allows to construct new list expressions based on existing lists by using the set-builder notation. These expressions can be assigned to list variables at any time after the initialization, but not in the declaration.

The general form of a list comprehension is:

`list := [x | var x : Type & (condition)]`, where `Type` is the type of the list elements and `condition` is a logical sentence which determines whether a value should be inserted to the list.

The example in Figure 4 illustrates both *Quantifier Expressions* and *List Comprehension* by computing prime numbers.

```
types
  MyInt = int [0..42];
  SmallInt = int [0..3];
  MyTuple = (MyInt, SmallInt);
  Sum = autocons system
  |[
  var
        my_list : list [4] of SmallInt = [5,5,5,5]
  methods
    add(a: MyInt, b: SmallInt) : MyInt =
        requires true :
          result := a + b
        end
    end ;
    sum(Xs : list [4] of SmallInt) : MyInt =
        requires true  :
          result := add :: (0) :>: (Xs)
        end
    end
  actions
        obs Sum(a: MyInt) =
                requires a = sum(my_list) :
                        skip
                end
  do
        var A : MyInt : Sum(A)
  od
  ]|
system
        Sum
```

Figure 3: Defining sum of all list elements using the fold meta function

3.5.5 Tuple Expressions

The general form of an expression that evaluates to a tuple is:

`TupleType(v_1,...,v_n)`

where TupleType is the type of the tuple and v_1,...,v_n are the values.

An example for using Tuples can be found in Figure 5

3.5.6 Access Expressions

Access Expression is the general term for the access of elements in a list or tuple, call of method or access to an attribute of an object.

Accessing elements of a list is done by using the index operator `[]` well known from accessing arrays in imperative languages like C. The index starts with 0.

The index operator `[]` can also be used to access a specific element of a tuple. This feature is illustrated in the guard of action `change2` in Figure 5. However this feature is not implemented in the left hand side of an assignment, e. g. it is not possible to write `theTuple[0] := 1`.

```
types
  MyInt = int [2..100];
  LengthInt = int [0..40];
  Primes = autocons system
  |[
  var
        initialized : bool = false ;
        primes : list [ 40 ] of MyInt = [3]
  actions
    obs init = requires initialized = false :
      primes := [ x | var x: MyInt & (forall tmp : MyInt :
            (tmp < x => (x mod tmp <> 0 ))) ] ;
      initialized := true end;

    obs Prime(a: MyInt) =
      requires (exists i : LengthInt : (primes[i] = a)) :
        skip
      end

  do
        init() // var A : MyInt : Prime(A)
  od
   ]|
system
        Primes
```

Figure 4: Prime number calculation

3.5.7 Cast Expressions

When dealing with a class hierarchy it is possibly to use an object of a sub class instead of an object of its super class. This is called *upcast* and done implicitly in most contexts. However, when concatenating two lists it has to be done explicitly using the operator `as` as illustrated in Line 10 in Figure 6.

3.6 Operators

3.6.1 Numeric Operators

The numeric operators are `+` for adding two numbers of an integer or float type, `-` for subtracting, `*` for multiplying. The operator for the division of two numbers of a float type is `/`. For the division of two numbers of an integer type there are operators `div` for the quotient and `mod` for the remainder.

3.6.2 Logical Operators

The logical operators are inspired by VDM and written `and` for logical and, `or` for logical or, `not` for negation, `=>` for implication, and `<=>` for bi-implication.

3.6.3 Comparison Operators

For comparing two values of the same type there are `=` for equality

```
types
  MyInt = int [0..42] ;
  MyTuple = (MyInt, MyInt) ;
  TupleDemo = autocons system
  |[
  var
        theTuple : MyTuple = MyTuple(1,1)
  actions
        ctr change1 = requires theTuple = MyTuple(1,1) :
                theTuple := MyTuple(1,2)
                end ;
        ctr change2 = requires theTuple[1] = 2 :
                theTuple := MyTuple(1,3)
                end
  do
        change1 [] change2
  od
  ]|
system
        TupleDemo
```

Figure 5: Tuple Example

and `<>` for inequality.

Values of integer and float types can also be compared with `>` for greater, `>=` for greater or equal, `<` for less, and `<=` for less or equal.

3.6.4 List Operators (Head, Tail, Length, Concatenation)

The OOAS language provides several basic list operators that make it easy to use lists.

The first operator is `hd` which is short for *head*. This operator returns the first element of a list. Therefore if `my_list` is a list, then `hd my_list` is equivalent to `my_list[0]`.

The counterpart to *head* is *tail*, which returns the list without the first element. In OOAS the operator token is `tl`. So if you want to remove the first element from a list `my_stack` you can write `my_stack := tl my_stack`.

The next important operation is determining how many elements are currently in a list, which is the so-called *length* of a list. In OOAS this operator is written `len`. For example if an action should only be enabled if there are less than 10 elements in the list `my_stack` you can write `requires len my_stack < 10`.

The last operation is creating a list that contains all the elements of two given lists, which is called *concatenation*. The operator is written `^`. If for instance you want to append an element `a` to a list `my_stack`, you can write

```
my_stack := [a] ^ my_stack
```

The example in Figure 7 illustrates how these operators can be used to build a stack.

```
types
  Class1 = autocons system
  |[
  var
        O2 : Class2 = new (Class2);
        O3 : Class3 = new (Class3);
        OL : list[2] of Class2 = [nil]
  actions
        ctr fill = requires true :
                OL := [O2]^[O3 as Class2]
                end
  do
        fill
  od
  ]| ;
  Class2 = system
  |[
  ]| ;
  Class3 = system (Class2)
  |[
  ]|
system
        Class1
```

Figure 6: Cast Example

3.7 The Dood Block

The actual behavior of an action system is defined in the so-called **do od**-block . The **do od**-block was inspired by Dijkstra's guarded iteration statement. The canonical form of a **do od**-block is to list all named actions of the class and connect them using the `[]` operator. In this case in each iteration an action is available iff its guard evaluates to `true` and the system terminates iff no action is enabled.

However, in OOAS actions are neither limited to named actions nor is the non-deterministic choice the only possible operator.

Instead, all operators described in Section 3.4 are available as well as the built-in action `skip` and so-called anonymous actions. An anonymous action is a guard followed by assignments, methods or built-in action `skip`. In literature anonymous actions are often referred to as *guarded commands.*

If (named) actions are parametrized, enumeration can be used as generalization of the non-deterministic choice over the values of a data type.

For instance given a data type `SmallInt = int[0..3]` and a named action `doSomething ( a : SmallInt )` it is possible to write

```
do
var : A : SmallInt : doSomething(A)
od
```

as abbreviation for

```
do
```

```
types
  SmallInt = int [0..3];
  Stack = autocons system
  |[
  var
        my_stack : list [10] of SmallInt = [0]
  actions
        obs top (a : SmallInt) = requires (a = hd my_stack) :
                skip
                end ;
        obs pop = requires (len my_stack > 0) :
                my_stack := tl my_stack
                end;
        obs push (a : SmallInt) = requires len my_stack < 10 :
                my_stack := [a] ^ my_stack
                end

  do
        var A : SmallInt : push(A) []  pop() [] var B : SmallInt : top(B)
  od
  ]|
system
  Stack
```

Figure 7: Stack

```
doSomething(0) [] doSomething(1) [] doSomething(2) [] doSomething(3)
od
```

3.8 The System Assembling Block

In the System Assembling Block the system is assembled by its classes. All the classes used by the system are connected either by the non-deterministic choice operator `[]` or by the prioritized composition operator `//`. This feature allows the user to manually set the priority among the classes.

3.9 Comments

The OOAS language supports single line comments. Comments start with a `#` symbol.

3.10 Formal Syntax

The following EBNF grammar defines the syntax of an Object-Oriented action system (OOAS). Features that are not implemented at all are left out. However there are some features that are not completely implemented and do produce code that doesn't work. These features are stroked out.

OOAS	:=	['consts' ConstList] 'types' TypeList 'system' ASTypeComp
ConstList	:=	NamedConst { ';' NamedConst }
NamedConst	:=	Identifier '=' Exp
TypeList	:=	NamedType { ';' NamedType }
NamedType	:=	Identifier '=' (ComplexType | OOActionSystem)
ComplexType	:=	('list' '[' (Num | Identifier) ']' 'of' ComplexType)
		| ('[' Identifier { ',' Identifier } ']')
		| ('(' ComplexType { ',' ComplexType } ')')
		| SimpleType
SimpleType	:=	'bool'
		| ('int' '[' (Num | Identifier) '..' (Num | Identifier) ']')
		| ('float' '[' (FloatNum | Identifier) '..' (FloatNum | Identifier) ']')
		| 'char'
		| '{' Identifier ['=' Num] { ', Identifier ['=' Num] } '}'
		| Identifier
OOActionSystem	:=	['autocons'] 'system' [Identifier]
		'|[' ['var' AttrList] ['methods' MethodList]
		['actions' NamedActionList] ['do' [ActionBlock] 'od'] ']|'
AttrList	:=	Attr { ';' Attr }
Attr	:=	['static'] ['obs' | 'ctr'] Identifier ':' ComplexType ['=' Exp]
MethodList	:=	Method { ';' Method }
Method	:=	Identifier ['(' MethodParamList ')'] [':' ComplexType] '='
		['var' LocalActionVars 'begin'] ActionBody 'end'
MethodParamList	:=	Identifier ':' ComplexType { ',' Identifier ':' ComplexType }
NamedActionList	:=	NamedAction { ';' NamedAction }
NamedAction	:=	['obs' | 'ctr'] Identifier ['(' MethodParamList ')']
		[':' ComplexType] '=' ['var' LocalActionVars] DiscreteActionBody
LocalActionVars	:=	Identifier ':' ComplexType { ';' Identifier ':' ComplexType}
DiscreteActionBody	:=	'requires' Exp ':' ActionBody 'end'
Exp	:=	AtomExpression BinOperator AtomExpression
AtomExpression	:=	([OpUn] (Reference | Constant | InitComplexType | QuantExp
		| '(' Exp ')' { '.' Identifier } [AccessExp])
		['as' Identifier])

		| ('if' Exp 'then' Exp 'else' Exp 'end')
OpUn	:=	'-' | 'not' | 'hd' | 'tl' | 'len'
Constant	:=	('true') | ('false') | ('nil') | ('self') | (FloatNum) | (Num) |
		(~~StringLiteral~~)
InitComplexType	:=	InitListType | ('new' '(' Identifier [',' StringLiteral] ')')
InitListType	:=	'[' Exp [ListComp | { ', Exp'}] ']'
ListComp	:=	'|' 'var' Identifier ':' ComplexType { ; Identifier ComplexType } '&' Exp
QuantExp	:=	('forall' | 'exists') Identifier ':' SimpleType
		{ ',' Identifier ': SimpleType' } ':' '(' Exp ')'
ActionBody	:=	ActionBodyParallel '//' ActionBodyParallel
ActionBodyParallel	:=	ActionBodySeq '[]' ActionBodySeq
ActionBodySeq	:=	ActionBodyParen ';' ActionBodyParen
ActionBodyParen	:=	('(' ActionBody ')') | (Statement)
Statement	:=	('skip') | (Reference { ',' Reference } ':=' Exp { ',' Exp})
Reference	:=	QualId [AccessExp] [['::' '(' Exp ')'] ':>:' '(' Exp ')']
AccessExp	:=	(('[' Exp ']') | ('(' MethodCallParam ')'))
		{(('[' Exp ']') — ('(' MethodCallParam ')')) }
		['.' Identifier { '.' Identifier} AccessExp]
QualId	:=	['self' '.'] Identifier { '. Identifier'}
MethodCallParam	:=	[Exp { ',' Exp}]
ActionBlock	:=	ActionBlockPar { '//' ActionBlockPar}
ActionBlockPar	:=	ActionBlockSeq { '[]' ActionBlockSeq}
ActionBlockSeq	:=	['var' BlockVarList [~~'&' Exp~~] ':']
		ActionBlockParen { ';' ActionBlockParen}
ActionBlockParen	:=	('(' ActionBlock ')') | (AnonOrNamedAct)
AnonOrNamedAct	:=	DiscreteActionBody | 'skip' |
		Identifier ['(' MethodCallParam ')'] [':>:' '(' Exp ')']
BlockVarList	:=	BlockVar { '; BlockVar'}
BlockVar	:=	Identifier ':' ComplexType
ASTypeComp	:=	ASTypeCompPar { '//' ASTypeCompPar}
ASTypeCompPar	:=	ASTypeCompSeq { '[]' ASTypeCompSeq}
ASTypeCompSeq	:=	ASTypeCompBP
ASTypeCompBP	:=	('(' ASTypeComp ')')
BinOperator	:=	('<=>') | ('>')| ('>=')| ('<')| ('<=)')| ('=')| ('<>')| ('=>') | ('-')| ('+')
		| ('or')| ('and')| ('/') | ('div') | ('*') | ('mod') | ('^')
Identifier	:=	Letter {Letter | Digit}
Num	:=	['+' | '-'] Digit { Digit }
FloatNum	:=	['+' | '-'] Digit { Digit } '.' Digit {Digit} ['e' | 'E' ['+' | '-'] Digit { Digit }]
Letter	:=	'$' | '_' | 'A'..'Z' | 'a'..'z'
Digit	:=	'0'..'9'

4 Incomplete Features

The following features are supported by the grammar, but do not produce executable or useful code.

4.1 Sequential Block Expression

In the **do od**-block when calling an action with a variable it is possible to add an additional guard. However if the guard is true, the produced Prolog-Code will not work.

Example:

```
do
var A : SmallInt & A < 3 : push(A)
od
```

4.2 String

It is possible to assign a string to a variable of any list type of char.

Example:

```
  var
my_string : list[20] of char = [0]
  actions
obs fill = requires true :
my_string := "Hello World!"
end
```

However, in the produced Prolog-Code the characters stored in the list are not of type char, but instead Prolog atoms. This may lead to strange behaviour.

5 CADP Target

5.1 The Argos CADP-Target

The CADP-Target is an Add-On for Argos. It produces C files for the use with the CADP toolbox available at `http://www.inrialpes.fr/vasy/cadp/`. It is deployed as dynamic library with the file name `ArgosCadpTarget.dll`. If this file is present an additional Argos option `-c` is available.

The invocation of Argos to produce C files for CADP is very similar to the invocation of Argos to produce prolog files for Ulysses. Instead of using the option `-p` one has to use the option `-c` and the options `-n` and `-d` should be omitted as they are meaningless in this context.

5.2 Generating BCG

The generated C file can be used to produce a graph in CADP's proprietary BCG format. This is done by compiling the file against CADP's tools using the `cadp_cc` compiler.

Given a shell with the environment variables `$CADP` set to the CADP directory and `$ARGOS` set to the Argos directory one can compile an Argos generated C file `model.c` into an object file `model.o` using the following command:

```
$CADP/src/com/cadp_cc -g3 -O0 -I. -I$CADP/incl -I$CADP/src/open_caesar \
-I $ARGOS/trunk/src/cadp/cadpruntime/ -c model.c -o model.o
```

Now the generator from the CADP toolbox can be compiled using the command:

```
$CADP/src/com/cadp_cc  -g3 -O0 -I. -I$CADP/incl -I$CADP/src/open_caesar \
-c $CADP/src/open_caesar/generator.c -o generator.o
```

Finally the two object files can be linked into an executable using the command:

```
$CADP/src/com/cadp_cc generator.o model.o -o generator \
-L${BCG:-$CADP}/bin.‘$CADP/com/arch‘ -lcaesar -lBCG_IO -lBCG -lm
```

5.3 Animator

For debugging purposes Argos provides a so-called Animator that enables one to walk step wise through a model. The models can be either encoded as graphs in `aut` files or as dynamic link library compiled from an Argos generated C file. The compilation can be done using any C compiler that is capable of generating dynamic link libraries for 32 bit Windows platforms.

However, compiling the model to a dynamic link library can be a little bit tricky, so a solution file for Microsoft Visual Studio 2008 can be found in the argos source directory or to be exact in the sub-directory `cadp/model`. Unfortunately some paths in this file are absolute, so they have to be changed to match the paths that are actually used.

This can be done by a right-click on *model* within the *Solution Explorer*. Now a *model Property Pages* window pops up, providing an element *Configuration Properties*. This element can be expanded, and *C/C++* has to be chosen. The first entry of this page, called *Additional Include Directories* has to be changed to fit.

6 Alarm System

6.1 Overview

In this section a model of a Car Alarm System is presented. The purpose is to show how Object-Oriented Action Systems can be used to model real world system in contrast to the examples presented so far. The Car Alarm System has been a demonstrator in the MOGENTES project. The specifications were provided by Ford. This demonstrator was also used in [1].

6.2 Requirements

In [1] the specification was condensed to the following three requirements:

R1: Arming. The system is armed 20 seconds after the vehicle is locked and the bonnet, luggage compartment, and all doors are closed.

R2: Alarm. The alarm sounds for 30 seconds if an unauthorized person opens the door, the luggage compartment, or the bonnet. The hazard flasher lights will flash for five minutes.

R3: Deactivation. The CAS can be deactivated at any time, even when the alarm is sounding, by unlocking the vehicle from outside.

6.3 State Chart

In [1] a UML transformation tool was used to generate an Object-Oriented action system. The paper contains a state chart which is cited in Figure 8. In contrast the action system presented in this manual was written by hand but it is somehow based on this State Chart regarding the interpretation of the requirements.

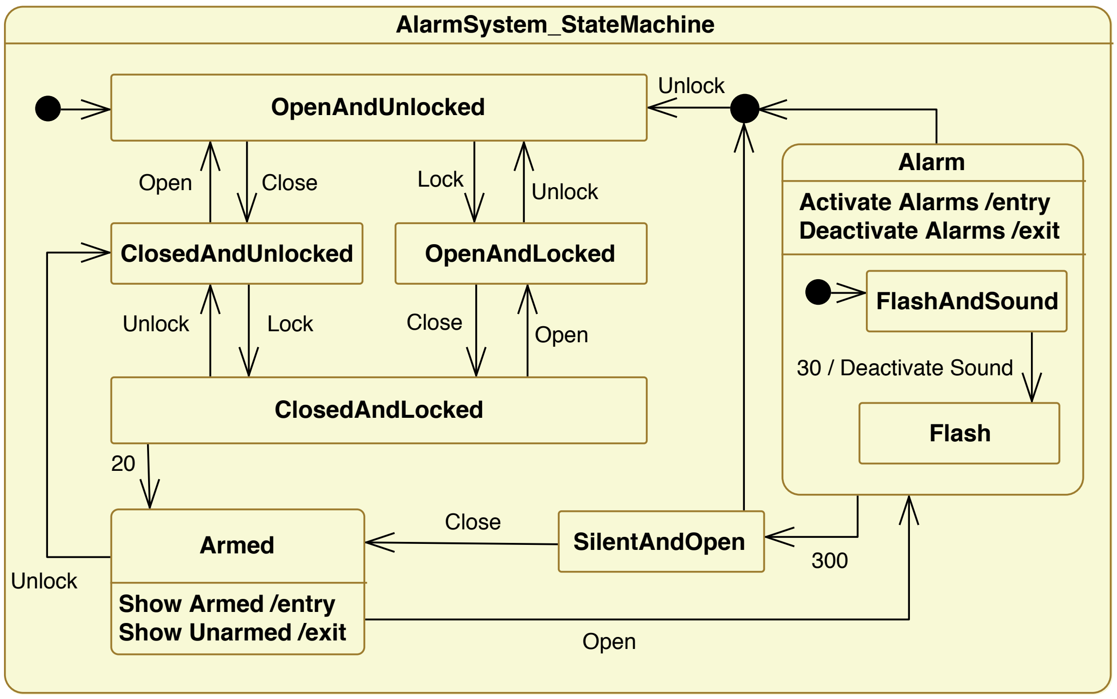

Figure 8: State Chart of Car Alarm System

6.4 Object-Oriented Action System

6.4.1 Interface

In the context of generating labeled transition systems for *Ulysses* the interface of an Object-Oriented action system consists of named actions. According to the ioco theory, labels have to be marked as either input or output. In Object-Oriented action systems the corresponding keywords are `ctr` resp. `obs`.

Therefore the interface of the Car Alarm System consists of 4 controllable actions `Close`, `Open`, `Lock` and `Unlock`. In the context of the implementation these actions correspond to the input coming from various sensors. In the context of a generated test case these actions correspond to outputs to the System Under Test.

Also there are 6 observable actions `ArmedOn`, `ArmedOff`, `SoundOn`, `SoundOff`, `FlashOn`, and `FlashOff`. In the context of the implementation these action correspond to output that triggers the resp. signals to the environment. In the context of the generated test cases these actions correspond to the expected inputs coming from the System Under Test.

6.4.2 Time Handling

One important aspect of modeling such a system is time handling. In former models within the MOGENTES project passing of time had been modeled by introducing an observable action *after*. Later in the project this caused some problems as some test cases were not generated.

To solve these problems the after action in this model is removed. Instead, elapsed time is now encoded as first parameter of every remaining action.

6.4.3 Protocol Layer

In the context of model-ling an embedded system the **do od**-block can be thought as protocol layer. To keep the model simple, the most canonical form of the **do od**-block has been used. That is, all named actions are connected using the non-deterministic choice. Since the named actions are parametrized, enumeration is used. Unfortunately the size of the state space grows exponentially by number of enumerated values, so it is very important to keep the resp. ranges as small as possible. In the context of this model, where the meaning of each action's parameter is the elapsed time, this means that only "interesting" time periods are considered. Actually, "interesting" events only happen immediately, 20, 30 or 270 seconds after the previous event. Exactly these values are contained in the Enum Type `TimeSteps`.

6.4.4 Non-determinism

This model makes use of non-determinism. That means that there are situations in which the system can choose how it responses to a certain input. More technically that means, that there sequences of controllable and observable actions, after which there is more than one observable action. In the context of using the model to specify a system that means underspecification. A system can choose among some possible options how to react.

In this particular model, non-determinism is used so that the implementation has some freedom with regards to the exact sequence of observable actions. For example when an intrusion is detected by the alarm system, then three events happen immediately: The arming is set off, the sound alarm is set on and the flash alarm is set on. A deterministic system would specify the exact order, in which these actions must be triggered. In the context of test case generation this could lead to some false positives, as the requirements do not specify a fixed order.

6.4.5 Code

```
types
  TimeSteps = {Int0 = 0 , Int20 = 20, Int30 = 30, Int270 = 270} ;
  Int = int[0..270] ;
  SmallInt = int[0..3] ;
  AlarmSystem = autocons system
  |[
    var
      open : bool = true ;
      locked : bool = false ;
      armed : bool = false ;
      soundAlarm : bool = false ;
      flashAlarm : bool = false ;
      blockingLevel : SmallInt = 0 ;
      armedOnNow : bool = false ;
      armedOnLater : bool = false ;
      armedOff : bool = false ;
      soundOn : bool = false ;
      soundOffNow : bool = false ;
      soundOffLater : bool = false ;
      flashOn : bool = false ;
      flashOffNow : bool = false ;
      flashOffLater : bool = false  ;
      silent : bool = false # true after alarms turned off
    actions
      ctr Close ( waittime : Int) =
        requires not flashAlarm and not soundAlarm :
          requires open and waittime = 0 and blockingLevel = 0 and locked and
              silent:
            open := false ;
            blockingLevel := 1 ;
            armedOnNow := true ;
            silent := false
          end []
          requires open and waittime = 0 and blockingLevel = 0 and not locked and
              silent:
            silent := false ;
            open := false
          end []
          requires open and waittime = 0 and blockingLevel = 0 and locked and not
              silent:
            open := false ;
            blockingLevel := 1 ;
            armedOnLater := true
          end []
          requires open and waittime = 0 and blockingLevel = 0 and not locked and
```

```
              not silent:
            open := false
          end
        end ;
      ctr Open ( waittime : Int) =
        requires true :
          requires not open and waittime = 0 and blockingLevel = 0 and not armed :
            open := true
          end []
          requires not open and waittime = 0 and blockingLevel = 0 and armed :
            open := true ;
            armed := false ;
            blockingLevel := 3 ;
            armedOff := true ;
            soundOn := true ;
            flashOn := true
          end
        end ;
      ctr Unlock ( waittime : Int) =
        requires true :
          requires locked and waittime = 0 and blockingLevel = 0 and not
              flashAlarm and soundAlarm:
            locked := false ;
            silent := false ;
            soundOffLater := false ;
            soundOffNow := true ;
            blockingLevel := 1
          end []
          requires locked and waittime = 0 and blockingLevel = 0 and flashAlarm
              and soundAlarm :
            locked := false ;
            flashOffNow := true ;
            blockingLevel := 2 ;
            silent := false ;
            soundOffLater := false ;
            soundOffNow := true
          end []
          requires locked and waittime = 0 and blockingLevel = 0 and not
              flashAlarm and not soundAlarm:
            locked := false ;
            silent := false ;
            armedOff := true ;
            blockingLevel := 1
          end []
          requires locked and waittime = 0 and blockingLevel = 0 and flashAlarm
              and not soundAlarm:
            locked := false ;
            flashOffNow := true ;
            flashOffLater := false ;
            blockingLevel := 1 ;
            silent := false
          end
        end;
      ctr Lock ( waittime : Int) =
        requires true :
          requires not locked and waittime = 0 and blockingLevel = 0 and open :
            locked := true
          end []
          requires not locked and waittime = 0 and blockingLevel = 0 and not open
              :
            locked := true ;
            blockingLevel := 1 ;
            armedOnLater := true
          end
        end ;
      obs ArmedOn (waittime : Int) =
        requires (armedOnNow or armedOnLater):
          requires (waittime = 20 and armedOnLater) :
            armedOnLater := false ;
            blockingLevel := blockingLevel − 1 ;
            armed := true
          end
          [] requires (waittime = 0 and armedOnNow) :
            armedOnNow := false ;
            blockingLevel := blockingLevel − 1 ;
            armed := true
          end
        end ;
      obs ArmedOff (waittime : Int) =
        requires (waittime = 0 and armedOff) :
          armedOff := false ;
          blockingLevel := blockingLevel − 1 ;
          armed := false
        end ;
      obs SoundOn (waittime : Int) =
        requires true :
          requires (waittime = 0 and soundOn and blockingLevel = 1) :
            soundOn := false ;
            blockingLevel := blockingLevel − 1 ;
            soundAlarm := true ;
            soundOffLater := true
          end []
          requires (waittime = 0 and soundOn and blockingLevel <> 1) :
            soundOn := false ;
            blockingLevel := blockingLevel − 1 ;
            soundAlarm := true
          end
        end ;
      obs SoundOff (waittime : Int) =
        requires (soundOffNow or soundOffLater) :
          requires (waittime = 30 and soundOffLater) :
            soundOffLater := false ;
            flashOffLater := true ;
            soundAlarm := false
          end
          [] requires (waittime = 0 and soundOffNow) :
            soundOffNow := false ;
            blockingLevel := blockingLevel − 1 ;
            soundAlarm := false
          end
        end ;
      obs FlashOn (waittime : Int) =
        requires true :
          requires (waittime = 0 and flashOn and blockingLevel = 1) :
            flashOn := false ;
            blockingLevel := blockingLevel − 1 ;
            flashAlarm := true ;
            soundOffLater := true
          end []
          requires (waittime = 0 and flashOn and blockingLevel <> 1) :
            flashOn := false ;
            blockingLevel := blockingLevel − 1 ;
            flashAlarm := true
        end
```

```
          end ;
        obs FlashOff (waittime : Int) =
          requires (flashOffNow or flashOffLater) :
            requires (waittime = 270 and flashOffLater) :
              flashOffLater := false ;
              flashAlarm := false ;
              silent := true
            end
            [] requires (waittime = 0 and flashOffNow) :
              flashOffNow := false ;
              blockingLevel := blockingLevel - 1 ;
              flashAlarm := false
            end
          end
  do
    var A : TimeSteps : Close (A) []
    var B : TimeSteps : Open (B) []
    var C : TimeSteps : Lock (C) []
    var D : TimeSteps : Unlock (D) []
    var E : TimeSteps : ArmedOn (E) []
    var F : TimeSteps : ArmedOff (F) []
    var G : TimeSteps : SoundOn (G) []
    var H : TimeSteps : SoundOff (H) []
    var I : TimeSteps : FlashOn (I) []
    var J : TimeSteps : FlashOff (J)
  od
  ]|
system
  AlarmSystem
```

6.5 Labeled Transition System

In Figure 9 you can find a graphical representation of labeled transition system that corresponds to the Object-Oriented action system. It is a ioco product graph produced by *Ulysses*. So there is an additional observable action *delta* that denotes quiescence. Iff in a given state the system has no specified output, there is a self loop labeled with *delta*.

Acknowledgements

Funded by the European Artemis Joint Undertaking project MBAT, Combined Model-based Analysis and Testing of Embedded Systems and by the Austrian Research Promotion Agency (FFG), program line "Trust in IT Systems", project number 829583, TRUst via Failed FALsification of Complex Dependable Systems Using Automated Test Case Generation through Model Mutation (TRUFAL).

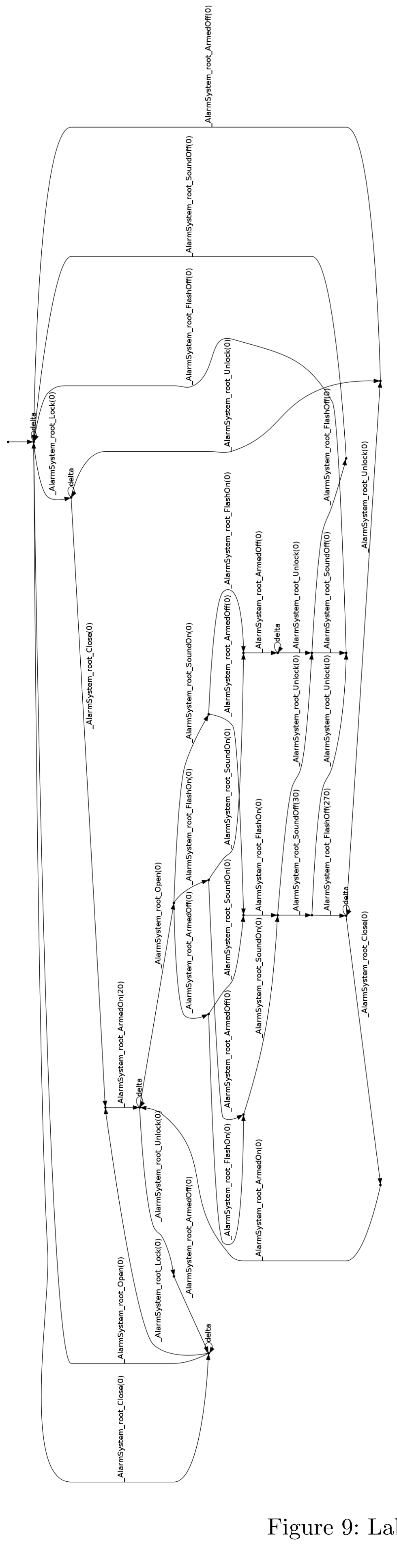
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_SoundOff(0)
_AlarmSystem_root_FlashOff(0)
_AlarmSystem_root_Unlock(0)
delta
_AlarmSystem_root_Lock(0)
delta
_AlarmSystem_root_Unlock(0)
_AlarmSystem_root_FlashOff(0)
_AlarmSystem_root_FlashOn(0)
_AlarmSystem_root_Unlock(0)
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_Unlock(0)
delta
_AlarmSystem_root_SoundOff(0)
_AlarmSystem_root_SoundOn(0)
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_Close(0)
_AlarmSystem_root_Unlock(0)
_AlarmSystem_root_Unlock(0)
_AlarmSystem_root_SoundOn(0)
_AlarmSystem_root_SoundOff(30)
_AlarmSystem_root_FlashOn(0)
_AlarmSystem_root_FlashOff(270)
_AlarmSystem_root_FlashOn(0)
_AlarmSystem_root_Open(0)
_AlarmSystem_root_SoundOn(0)
delta
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_SoundOn(0)
_AlarmSystem_root_Close(0)
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_ArmedOn(20)
delta
_AlarmSystem_root_Unlock(0)
_AlarmSystem_root_FlashOn(0)
_AlarmSystem_root_ArmedOn(0)
_AlarmSystem_root_ArmedOff(0)
_AlarmSystem_root_Lock(0)
_AlarmSystem_root_Open(0)
delta
_AlarmSystem_root_Close(0)

Figure 9: Labeled Transition System

References

[1] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. UML in action: A two-layered interpretation for testing. In *UML&FM*, ACM Software Engineering Notes (SEN), 2010. in press.

[2] Marcello M. Bonsangue, Joost N. Kok, and Kaisa Sere. An approach to object-orientation in action systems. In *Mathematics of Program Construction, LNCS 1422*, pages 68–95. Springer, 1998.