

CS 578 – Software Architectures Spring 2010

Homework Assignment #2

Due: *Tuesday, March 9, 2010*

– *see course websites for submission details* –

In this assignment, you will be introduced to architectural description languages, specifically to support a dynamic robotic application that is described in the next section. You will first determine and model the system requirements for this application in a semantic model called *Domain Model*, which is described in Addendum 1. Secondly, you should design an architecture that supports your Domain Model in the C2 style. You should include full specifications of all component and connector types, as well as their instantiations and interconnections in the architecture. Then you are to use the C2SADEL architecture description language (ADL), which is described in addendum 2, and which was overviewed in class,¹ to specify this architecture.

Application Description

The application you are to design consists of 3 *follower robots* that form a convoy and follow a *leader robot*. The *leader robot* follows landmarks in the environment and never fails. To make the robots observable to each other, each robot has a unique color and emits a unique code using an infrared (IR) LED. Furthermore, each *follower robot* is equipped with an IR sensor and a camera. Consequently, each *follower robot* can distinguish the color of the robot in front of it and follow it using its camera. Moreover, each *follower robot* can distinguish the IR code of the robot in front of it and follow it using its IR sensor.

The aim of the application is to keep the convoy running in the face of system failures including camera failures and depleted batteries. Cameras are the primary tools for following. However when a robot's camera fails, the robot can utilize its infrared sensor to continue following the robot in front of it (but with lower quality of service). Moreover, a robot's battery may die during the operation. In such a case the robot behind the failed robot in the convoy should switch to following the robot in front of the failed robot.

¹ You can find detailed information on C2SADEL at <http://sunset.usc.edu/~nenod/dissertation/dissertation2.pdf>

Deliverables

You should include the following items in your submission in a single file:

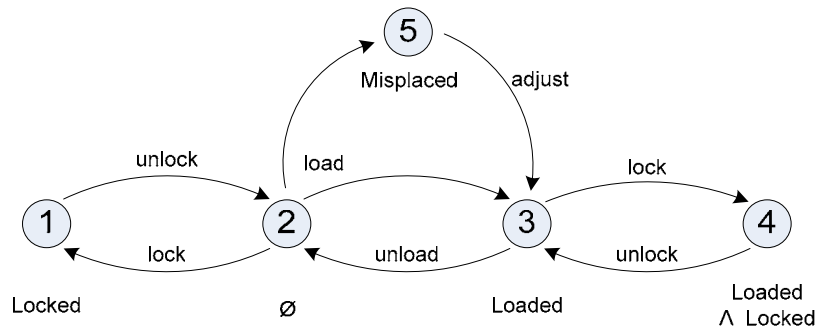
- 1) The Domain Model. We prefer to get the graphical representation of your domain model (See Addendum 1).
- 2) Graphical representation of your suggested architecture. You should show the components, connectors, and their interconnections in your architecture.
- 3) The C2SADEL description of your architecture. You should include full specifications of all component and connector types, as well as their instantiations and interconnections in the architecture.

Hints

- 1- To design the components you should consider the separation of concerns and abstraction rules. Consider the example Domain Model in the Addendum 1. To support this domain model one may design 4 component types: A “Loader Component” which implements the load and unload operations, a “Locker Component” which implements the lock and unlock operations, a “Adjuster Component” which performs the adjust operation, and a “Sensor Component” which performs the sensing of the environment and providing the values for Loaded, Locked, and Misplaced variables.
- 2- You do not need to model low-level actions in the system (i.e. you do not need to model a color following algorithm). You may consider actions in the granularity level of ColorFollowing, IRFollowing and so on.
- 3- Some possible variables of the domain model can be IsCameraWorking, IsBatteryWorking and so on.

Addendum 1

The system requirements for a domain can be described formally in a semantic model called **Domain Model**. This model defines *States* of the domain, available *Actions* in the domain, and the *State Transitions* caused by those actions. A domain model is formally defined as a 4-tuple $\{V, S, A, R\}$ in which, V is a finite set of variables, $S \subseteq 2^V$ is a finite set of States, A is a finite set of Actions and $R: S \times A \mapsto S$ is a Transition Function. Consider a domain in which an item can be loaded/unloaded to/from a container which can be locked/unlocked. Moreover, the item might get misplaced in the container in the process of loading. Figure 1 is a graphical presentation of this domain in which Loaded, Locked, and Misplaced are the variables of the domain, circles represent the states and arrows represent the actions of the domain.



The formal description of this domain is as follows:

$$V = \{Loaded, Locked, Misplaced\}$$

$$S = \{\emptyset, Loaded, Locked, Misplaced, Loaded \wedge Misplaced\}$$

$$A = \{load, unload, lock, unlock, adjust\}$$

$$R = \{\{Locked, unlock, \emptyset\}, \{\emptyset, lock, Locked\}, \{\emptyset, load, Misplaced \vee Loaded\},$$

$$\{Loaded, unload, \emptyset\}, \{Misplaced, adjust, Loaded\}, \{Loaded, lock, Loaded \wedge Locked\},$$

$$\{Loaded \wedge Locked, unlock, Loaded\}\}$$

Addendum 2

C2SADEL: An Architecture Description and Evolution Language

This addendum introduces C2SADEL, a Software Architecture Description and Evolution Language for C2-style architectures. The complete specification of C2SADEL's syntax is given below. C2SADEL supports component evolution via heterogeneous subtyping and facilitates architectural descriptions that allow establishment of type-theoretic notions of architectural soundness. It also supports modeling of connectors with context-reflective interfaces and different data filtering capabilities, as well as configurations that adhere to the topological rules of the C2 style.

We encountered a tension between formality and practicality in designing C2SADEL. Our goal was a language that was simple enough to be usable in practice, yet formal enough to adequately support analysis and evolution. For this reason, we kept the syntax simple and reduced formalism to a minimum.

A C2SADEL specification consists of either a set of component types or of an architecture. An architecture contains a specification of component types, connector types, and topology. To properly specify an architecture's topology, component and connector types are instantiated and connected.

1. Component Types

A component specification is a type that can be defined in-line or externally (using the keyword *extern*). The specification of an external component type is given in a file different from the file in which the rest of the architecture is specified. For example,

```
component WellADT is extern {WellADT.c2;}
```

specifies that the *WellADT* component used in the KLAX architecture (Figure 1-1) is specified in the file *WellADT.c2*. This feature allows for components to be treated as reusable design elements, independent of an architecture. A component type consists of the following:

- state variables,
- component invariant,
- interface,
- behavior, and
- the map from interface elements to the operations of the behavior. This map is a surjective function.

A component type may be a *subtype* of another type. The exact subtyping relationship must be specified. Keywords *nam*, *int*, *beh*, and *imp* are used to denote name, interface, behavior, and implementation conformance, respectively. Different combinations of these relationships are specified using the keywords *and* and *not*. For example,

```
component WellADT is subtype Matrix (beh)
```

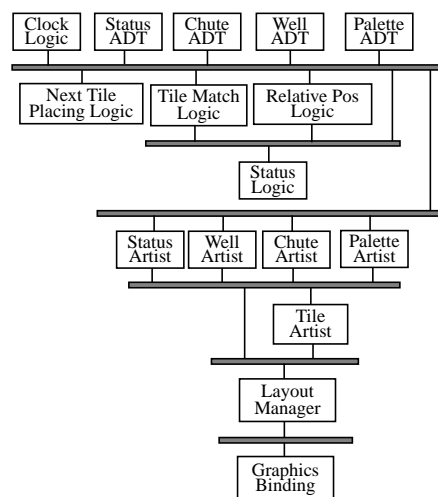


Figure 1-1. KLAX architecture.

specifies that the KLAX component *WellADT* preserves (and possibly extends) the behavior of a component *Matrix*, but may change its interface. This relationship can be made stricter by specifying that *WellADT* must alter *Matrix*'s interface as follows:

```
component WellADT is subtype Matrix (beh \and \not int)
```

As in a programming language, variables are specified as <name, type> pairs, as in

```
capacity : Integer;
```

Additionally, a component's state variable may also be specified as a function:

```
well_at : Integer -> Color;
```

The *well_at* function maps a set of *Integer* locations in the well to a set of *Color* tiles at each location.

Variable types in C2SADEL, such as *Integer* or *Color*, are *basic* types and are distinguished from components, which are *architectural* types. We do not explicitly model the semantics of basic types; however, C2SADEL does allow the architect to specify that one basic type is a subtype of another:

```
Natural is basic_subtype Integer;
```

A component's invariant is a conjunction of predicates specified in first-order logic. The invariant defines a set of conditions that must be satisfied throughout the component's execution. It is specified with component state variables as operands and logical operators (*\and*, *\or*, *\not*, *\implies*, and *\equivalent*), comparison operators (*\greater*, *\less*, *\eqgreater*, *\eqless*, *=*, and *<>*), set operators (*\union*, *\intersection*, *\in*, *\not_in*, and *#*), and arithmetic operators (*+*, *-*, ***, */*, and *^*).¹ Operator precedence in C2SADEL is defined as shown in Table 1-1.

Table 1-1: C2SADEL operator precedence (in descending order)

#, \not
^
*, /
+, -
\union, \intersection
\greater, \eqgreater, \less, \eqless, =, <>, \in, \not_in
\and, \or
\implies, \equivalent

For example, the invariant for the *WellADT* component can be specified as follows.

```
invariant {
  (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
}
```

A component's interface consists of a set of interface elements. An interface element is declared with a direction indicator (*prov* or *req*), name, set of parameters, and possibly a result type. The parameter specification syntax is identical to that used in variable specification. Since interface elements may have identical names, a unique label may be assigned to each as a notational convenience. For example, in

```
prov gt1: GetTile (location : Integer) : Color;
prov gt2: GetTile (i : Natural) : GSColor;
```

both interface elements are intended to be used with operations that remove and return a tile at the given location in the KLAX *well*. The first interface element accesses a color tile at the *Integer* location *location*; the second accesses a gray-scale tile at the *Natural* location *i*. The labels, *gt1* and *gt2*, uniquely identify the two.

1. *<>* denotes inequality; *\in* and *\not_in* denote set membership; *#* denotes set cardinality; *^* denotes exponentiation.

A component's behavior consists of a set of operations. Each operation is declared as either *provided* or *required* and with a unique label, used to refer to the operation. Additionally, each operation may define a set of preconditions that must be true *prior* to the operation's execution, and a set of postconditions that must be true *after* its execution. Since operations are separated from the interface elements through which they are accessed, operations also define local variables, which, along with component state variables, are used in specifying the pre- and postcondition predicates. The pre- and postconditions are specified in the same manner as component invariants. An operation's postcondition may contain the keyword `\result`, to denote the operation's return value. Additionally, a postcondition may specify the value of a variable after the operation has executed, denoted with a `~`, followed by the variable name.

An example operation can be specified as follows.

```
prov tileget: {
  let pos : Integer;
  pre (pos \greater 0) \and (pos \eqless num_tiles);
  post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
}
```

The local variable `pos` denotes the position in the well. `num_tiles` and `well_at` are component state variables. Recall that `well_at` is a function that returns the color value of the well at the given position. The postcondition specifies that the number of tiles in the well decreases after the tile is removed.

The `tileget` operation can export multiple interfaces. For example, both `GetTile` interface elements can be mapped to the operation, provided that `GSColor` is a basic subtype of `Color`:

```
map {
  gt1 -> tileget (location -> pos);
  gt2 -> tileget (i -> pos);
}
```

These elements are composed into a complete component specification as follows:²

```
component WellADT is subtype Matrix (beh) {
  state {
    capacity : Integer;
    num_tiles : Integer;
    well_at : Integer -> GSColor;
  }
  invariant {
    (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
  }
  interface {
    prov gt1: GetTile (location : Integer) : Color;
    prov gt2: GetTile (i : Natural) : GSColor;
  }
  operations {
    prov tileget: {
      let pos : Integer;
      pre (pos \greater 0) \and (pos \eqless num_tiles);
      post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
    }
  }
  map {
    gt1 -> tileget (location -> pos);
    gt2 -> tileget (i -> pos);
  }
}
```

2. For illustration, the specification of `WellADT` only includes the aspects of this component previously discussed.

Finally, a component type may be specified as a *virtual* type: it can be used in the definition of the topology, but it does not have a specification and does not affect type checking of the architecture; furthermore, a virtual type cannot be evolved via subtyping. The concept of virtual types is useful in the case of components for which implementations are known to already exist, but which are not specified in C2SADEL.

2. Connector Types

Since the connectors in this dissertation do not export a particular interface, but are context-reflective, the only aspect of connector types modeled in C2SADEL is their filtering mechanism, denoted with the *message_filter* keyword. The different filtering mechanisms are *no_filtering*, *notification_filtering*, *message_filtering*, *prioritized*, or *message_sink*. An example broadcast connector is specified as follows.

```
connector BroadcastConn is {
  message_filter no_filtering;
}
```

3. Topology

To model the topology of an architecture, component and connector types are instantiated and interconnected. Each type may be instantiated multiple times. C2SADEL requires that a component be attached to at most one connector on its top and one on its bottom; it allows multiple components and connectors to be attached to the top and bottom sides of a connector. The part of the KLAX topology that concerns the well is specified as follows.

```
architectural_topology {
  component_instances {
    Well : WellADT;
    WellArt : WellArtist;
    MatchLogic : TileMatchLogic;
  }
  connector_instances {
    ADTConn : BroadcastConn;
    ArtConn : BroadcastConn;
  }
  connections {
    connector ADTConn {
      top Well;
      bottom MatchLogic, ArtConn;
    }
    connector ArtConn {
      top ADTConn;
      bottom WellArt;
    }
  }
}
```

C2SADEL Syntax Summary

This section contains the complete BNF specification of C2SADEL. For simplicity, all literals, including single-character literals (e.g., ‘}’ or ‘;’) are displayed in bold type. Single-character literals are displayed without quotation marks. Unless bolded, curly braces (‘{’ and ‘}’) represent repetition of the enclosed expression. “{...}*” represents zero or more occurrences, while “{...}+” denotes one or more occurrences.

```
arch_component_set ::=
  (arch_component_type)*

arch_component_type ::=
  component identifier is arch_component_type_decl

arch_component_type_decl ::=
  component_type_decl | virtual_comp_type

arch_component_types ::=
  component_types { arch_component_set }

arch_connector_type ::=
  connector identifier is
  {
    message_filter filtering_policy ;
  }

arch_connector_types ::=
  connector_types { (arch_connector_type)* }

arch_topology ::=
  architectural_topology
  {
    component_inst
    connector_inst
    attachments
  }

attachments ::=
  connections { (connection_decl)* }

basic_subtype ::=
  identifier is basic_subtype identifier ;

basic_subtype_decl ::=
  basic_types { (basic_subtype)* }

behavior_decl ::=
  operations { (operation_decl)* }

binary_operator ::=
  = | <> | + | - | * | / | ^ |
  \implies | \equivalent |
  \and | \or |
  \union | \intersection | \in | \not_in
  \greater | \less | \eqgreater | \eqless
```



```

C2_architecture ::=
  architecture identifier is
  {
    [basic_subtype_decl]
    arch_component_types
    arch_connector_types
    arch_topology
  }

C2_component_set ::=
  [basic_subtype_decl]
  (component_type)+

C2_SADEL_spec ::=
  C2_architecture | C2_component_set

component_inst ::=
  component_instances { (instance_decl)* }

component_type ::=
  component identifier is component_type_decl

component_type_decl ::=
  extern_comp_type | local_comp_type

connection_decl ::=
  [ component | connector ] identifier
  {
    top [ connection_list ] ;
    bottom [ connection_list ] ;
  }

connection_list ::=
  identifier , connection_list | identifier

connector_inst ::=
  connector_instances { (instance_decl)* }

digit ::=
  0 | 1 | ... | 9

dir_indicator ::=
  prov | req

extern_comp_type ::=
  extern { filename ; }

filtering_policy ::=
  no_filtering |
  notification_filtering |
  message_filtering |
  prioritized |
  message_sink

function_decl ::=
  identifier : identifier -> identifier ;

identifier ::=
  letter { _ | letter | digit } *

instance_decl ::=
  identifier : identifier ;

```

```

integer ::=
  (digit)+

interface_decl ::=
  interface { (interface_element_decl)* }

interface_element_decl ::=
  dir_indicator identifier :
    identifier ( param_decl ) [: [\set] identifier] ;

invariant_decl ::=
  invariant { [logic_expr ;] }

let_decl ::=
  let {var_decl ;}* [pre_decl | post_decl]

letter ::=
  A | B | ... | Z | a | b | ... | z

local_comp_type ::=
  [subtype_decl]
  {
    state_decl
    invariant_decl
    interface_decl
    behavior_decl
    map_decl
  }

logic_expr ::=
  subexpr [\and subexpr]

map_decl ::=
  map { (single_map)* }

numeric_literal ::=
  [-] integer [. integer] [^ integer]

operand ::=
  [\not | #] identifier |
  numeric_literal      |
  subexpr              |
  ( subexpr )

operation_decl ::=
  dir_indicator identifier :
  { let_decl | pre_decl | post_decl }

param_decl ::=
  var_decl ; param_decl | var_decl

param_to_var ::=
  identifier -> identifier , param_to_var |
  identifier -> identifier

post_decl ::=
  post [post_logic_expr] ;

post_logic_expr ::=
  post_subexpr [\and post_subexpr]

```

```

post_operand ::=
  [\not | #] [~] identifier |
  numeric_literal          |
  post_subexpr             |
  ( post_subexpr )

post_subexpr ::=
  post_operand binary_operator post_operand |
  \result = post_operand

pre_decl ::=
  pre [logic_expr] ; [post_decl]

single_map ::=
  identifier -> identifier ( param_to_var ) ;

state_decl ::=
  state { (var_decl ; | function_decl ;)* }

subexpr ::=
  operand binary_operator operand

subtype_decl ::=
  subtype identifier ( subtype_rel_expr )

subtype_rel ::=
  nam | int | beh | imp

subtype_rel_expr ::=
  [\not] subtype_rel {\and [\not] subtype_rel}*

var_decl ::=
  identifier : [\set] identifier

virtual_comp_type ::=
  virtual { }

```