Abstract

UML is an acronym for Unified Modelling Language. It has become de facto the standard for the object-oriented software analysis and design stages in software development. UML is a visual modelling language, and it consists of a set of diagrams. Static diagrams are used to depict static structure of a program, whereis dynamic diagrams specify how the control flow(s) of the program should behave. The examples of behavioral diagrams are a State diagram, which describes the behavior of objects of a given class, and a Sequence diagram, which describes inter-object interactions in a given scenario.

A consistency problem may arise due to the fact that some aspects of the model may be described by more than one diagram. Hence, the consistency of the system description should be checked before implementing the system.

This thesis describes an algorithmic approach to a consistency check between UML Sequence and State diagrams. The algorithm we provide handles complex state diagrams, e.g. diagrams that include forks, joins, and concurrent composite states. We also describe BVUML, a tool we have implemented in order to automate the validation process.

Contents

1 Introduction **4**

2 Problem Description **6**

3 A Simple Example **8**

3.1 BVUML Top Level Results 9

3.2 The Detailed Output 11

4 Solution Description **14**

4.1 Inputs 14

4.2 The main idea of the solution 14

4.3 The algorithm 16

5 A Run Example **20**

5.1 The Sequence Diagram 21

5.2 The State Diagram 23

6 Run Results **25**

6.1 Part One 25

6.2 Part Two 26

6.3 Third Part 30

7 Design decisions **31**

7.1 Saturateable states 31

7.2 Forks implementation 32

7.3 Forks and joins as states 33

7.4 Non-determinism and per-step concurrency 34

7.5 Single threaded solution 34

7.6 Fine granularity step 34

7.7 Composite states 34

7.8 Some implementation notes 34

8 Related Work **37**

8.1 HUGO project 37

8.2 IIOSS Model Debugging Facility (MDF) 37

8.3 Prosasim UML simulator 38
8.4 PecSee project 38
8.5 LSC: Live Sequence Charts 39

9 Conclusions and Future Work 40
9.1 Partial legs ordering in a Sequence diagram 40
9.2 Compound initial transition 40
9.3 Several actions for a leg 40
9.4 Summary 41

10 Appendix A: Full output example 42

11 Appendix B: Definitions 48
11.1 Basic Behavior definitions 48
11.2 Sequence Diagram definitions. 49
11.3 State Diagram definitions. 49

1 Introduction

UML is an acronym for Unified Modelling Language. It has become the de facto standard for the object-oriented software analysis and design stages in software development. UML is a visual modelling language, and it consists of a set of diagrams. Static diagrams are used to depict static structure of a program, whereis dynamic diagrams specify how the control flow(s) of the program should behave. While describing all the standard UML diagrams, we emphasize Sequence and State diagrams, which are important for this thesis.

Static diagrams are Class diagram, Object diagram, Component diagram and Deployment diagram.

A Class diagram shows classes and types, their internal structure, and their interrelationship to other classes or types. Examples of such relationships are inheritance, association, aggregation, and (template class) instantiation relationships.

An Object diagram shows a snapshot of the detailed state of a system at a given point of time. This is an instance of a class diagram. Its use is fairly limited, mainly to exemplify data structures.

A Component diagram shows the dependencies among software components. This usually means showing compilation dependencies between different binary code files, and the mapping between the source code files and the binary code files.

A Deployment diagram shows the configuration of run-time processing elements and the software components, processors, and objects that execute on them. Software component instances represent run-time manifestation of software code units.

Dynamic diagrams are Use case diagram, Activity diagram, State diagram and Interaction diagrams.

A Use case diagram represents the functionality of a system or a subsystem, as manifested to external interactors. The external interactors may be other systems or human users.

An Activity diagram is a variation of a state machine in which the states represent the performance of activities and the transitions are triggered by the completion of activities.

A State diagram describes the behavior of objects of a given class. It includes inner states of the object and transitions between these states upon a specific event, such as a function call or a signal receipt.

For example, the inner states of a toggle button would be `Pressed` and `Unpressed`, and the transitions between the states will happen on the `Button Pressed` or `Button Released` events. The transitions in the imaginary State diagram of the toggle button example above might also invoke an action (function) that will write the event to a log or play a sound.

A State diagram also provides composite states, i.e. states that compose other states; concurrent composite states, control choice, fork and join states, and history states. We distinguish between two kinds of states: transient states and persistent states. Transient states are states in which the control flow must stay infinitely small amount of time. Persistent states are states in which the control must stay at least for a considerable amount of time. A thorough description of State diagrams is given in chapter 11.

Interaction diagrams describe how a group of objects interact in some behavior. An interaction is the communication pattern performed by the objects to accomplish a specific purpose, such as performing complex operation. There are two kinds of Interaction diagrams: Collaboration diagrams and Sequence diagrams. The difference between the diagrams is that the Sequence diagram emphasizes *time ordering* between inter-object messages (such as signals or function calls), whereas the Collaboration diagram emphasizes the messages themselves.

The Sequence diagram for a `Reboot` button press scenario would include 3 objects: the actor (a human user), the GUI object which receives the event, and the operation system access object. Another actor, the operation system, might be added as well. The user will send `ButtonPressed()` message to the reboot button, the button will send a `Reboot()` message to the operation system access object. One can develop this scenario by requiring a confirmation dialog to pop up upon reboot request. This way the reboot button, or some application business object, will forward `Confirm()` message to yet another GUI object in the diagram.

This thesis describes an algorithmic approach to consistency checking between UML Sequence and State diagrams. The algorithm we provide also handles complex state diagrams, e.g. diagrams that include forks, joins, and concurrent composite states. We also describe BVUML (an acronym of Behavioral Validator of UML), a tool we have implemented in order to automate the validation process. BVUML implements the consistency check algorithm. Hybrid Sequence-State diagrams are introduced to visualize the process; in these diagrams states are associated with the Sequence diagrams.

The rest of the thesis is organized as follows: Chapter 2 describes the problem in detail. In Chapter 3 we present the proposed solution by means of an example. Chapter 4 elaborates on the solution by describing the algorithm. Chaper 5 and Chapter 6 describe a complex example and the BVUML output on it. Some design decisions concerning BVUML are provided in Chapter 7. Finally, a short reference to related work and concluding remarks are given in Chapter 8 and Chapter 9.

2 Problem Description

All the UML diagrams used in a software project must form a consistent view of the project. One diagram must not contradict another. This thesis deals with inconsistencies between the Sequence diagram and the state diagrams for the classes of the objects that appear in the Sequence diagram.

An example of a relatively simple inconsistency is to include a method name for an object in a Sequence diagram, which does not appear in the method names list of the object's class. Inconsistencies of such type are already handled by many UML tools.

Inconsistencies can be more complex than this example. Let us introduce some formal definitions to make the explanation easier.

1. *Leg:* A basic component of a Sequence diagram or a State diagram, depicted in UML as an arrow. In the UML, a *Leg* is called a *transition* in a State diagram, whereas in a Sequence diagram it is called a *message*.

2. *Transition:* Consists of a positive number of legs.

3. *Sequence (diagram) transition:* A transition in a Sequence diagram that consists of an incoming leg (message) and all the outgoing legs (messages), if any, triggered by it.

4. *State (diagram) transition:* A transition in a State diagram that includes the trigger and all the legs on the path between two persistent states, with no persistent states in between.

5. *Current object state:* The conjunction of all the active states in a State diagram, that defines the behavior of the object for any incoming event. At the beginning of the object lifetime, its *current object state* includes only all the initial State diagram states. The *current object state* is updated with each state transition. It can include multiple simple states, if the diagram contains concurrent composite or fork states.

Let's describe more complex inconsistencies. Take, for example, some object in a Sequence diagram. The outgoing legs usually occur due to some incoming leg, such as a function call or a constructor call. The State diagram for the object's class describes the object behavior. The incoming leg to the object in the Sequence diagram serves as a trigger of the associated State diagram transition. If there is no transition for such a trigger in the State diagram, obviously either the object behavior definition in the State diagram is wrong or incomplete, or the Sequence diagram is incorrect. Either way, it is an inconsistency between the diagrams. But this is not the whole story.

Let `Objx` be the object where the control currently resides. In the beginning of the object lifetime, its *current object state* includes all the initial State diagram states. The *current object state* is updated with each state transition. It may happen that after an incoming Sequence diagram leg, the actions that are described in the State diagram are different from these described (as outgoing messages) in the Sequence diagram. That is, at the *current object state*, the Sequence of function calls or signal emissions specified in the Sequence diagram is illegal, according to the State diagram of that object. Note that the messages in the Sequence diagram might be consistent with the State diagram, from a different *current object state*. For example, in a imaginary design of an ice-cream eating robot, `eat ice-cream` does make sense only in the `ice-cream was bought` state, and not prior to it. Therefore the lookup for suitable State diagram transitions must start from the *current object state*.

The *Sequence transition* must match a *State transition* that emanates from a *current object state*. Otherwise the diagrams are inconsistent.

The problem of finding such inconsistencies becomes very complex in many cases. For example, the *current object state* may consist of many simple states due to transitions into concurrent composite, fork, and join states. These simple (non-composite persistent) states may be in different levels of a State diagram hierarchy, as will be explained later. States that change the control behavior, such as choice states and history states, make it harder to locate the *current object state* after a given transition. Many additional elements, such as transition guards (conditions), on entry/ on exit actions, transitions that pass hierarchical levels in the State diagram, etc, were not mentioned in this introduction. This is elaborated on in Chapter 7.

3 A Simple Example

The example describes the interaction between the components of a cellular phone, and the phone user. The example is taken from [Mar].

One use case of the cellular system is dialing a number. In one of the scenarios of this use case the user has entered a phone number. Just before he presses the send button, there is an incoming call. The cellular phone allows the user to answer an incoming call even after he has entered several digits. When the user presses the send button, this means that the `Dialer` object either has to make a call, or to receive an incoming call. The decision is made according to the `Dialer` state, whether there was an incoming call or not. If there was an incoming call, it is answered. A new call is attempted otherwise.

We want to present how our tool, BVUML, finds the possible inconsistencies. To illustrate the inconsistency search process, we use a Sequence diagram that is inconsistent with respect to the State diagram. The Sequence and state diagrams for the `Dialer` are specified in Figures 1 and 2, respectively. The Sequence diagram designer intended to make a call, even though the pressing of the `Send` button should– according to the State diagram of the Dialer– answer an incoming call. The run of the `Dialer` will detect the inconsistency at the first `send` message (message #6 in the Sequence diagram).

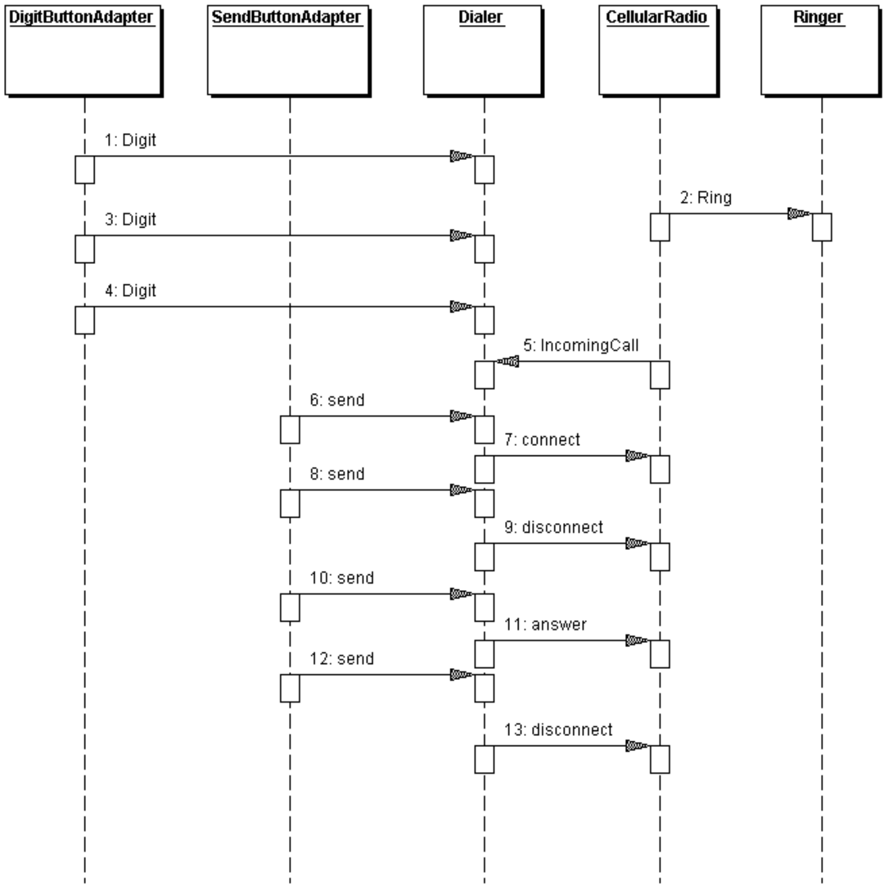

Figure 1: Dialer Sequence Diagram

3.1 BVUML Top Level Results

In this section we briefly describe the output of our tool on the State and Sequence diagrams shown above.

BVUML provides a way to view the results of a run as a hybrid Sequence-State diagram, which is basically a Sequence diagram with added state marks. This view is shown next. It includes all the Sequence diagram transitions, and for each transition - the states that are reached by running the transition are shown. They are described here in text mode, and can be presented also in HTML format. One can see, for example, that at the beginning, the *current object state* included only the `InitialState` state, and upon pressing a `Digit` the `Idle` state is reached. This is also the state in the beginning of the next transition.

```
        B V U M L   R u n   D i a g r a m
```

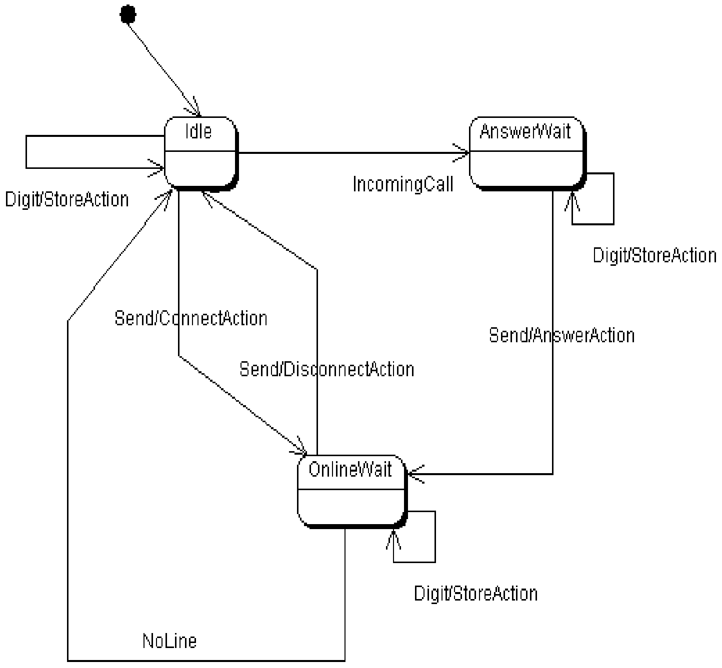

Figure 2: Dialer State Diagram

```
[cellular]
||
|| /____________________________________________________[(001) Digit]______
|| \
||   [ (InitialState) ]  // This is the current object state
||   [ (Idle) ]
||          |
||          |----------> (Idle) // goto next state in the state diagram
||
|| /____________________________________________________[(002) Digit]______
|| \
||   [ (Idle) ]
||          |
||          |----------> (Idle)
||
|| /____________________________________________________[(003) Digit]______
```

```
|| \
||   [ (Idle) ]
||         |
||         |----------> (Idle)
||
|| /_______________________________________[(005) IncomingCall]______
|| \
||   [ (Idle) ]
||         |
||         |----------> (AnswerWait)
||
|| /______________________________________________[(006) Send]______
|| \
||   [ (AnswerWait) ]
||         |
```

The numbers in this output describe the order of the messages which reach the `Dialer` object in the Sequence diagram. The numbers do not have to be subsequent, but they have to form ascending series. The reason they appear is that the UML Sequence diagram, when saved to UML standard model file, does not keep the order of the links. This UML standard model file has XML Model Interchange (XMI) type and will be described later.

3.2 The Detailed Output

The results of a BVUML run are composed of three parts. The first one describes the input data, the second part depicts the execution of the transitions, and the third sums up the results.

The first part expresses the input data extracted by BVUML from its input file. We used a UML standard compliant modelling tool, called ArgoUML. Guards and actions do not appear in the figures generated by ArgoUML, but they are shown in this part of the output.

```
========Input data=========
top[], parent: =null;
top.InitialState[Leg: {} to top.Idle],
    parent: top
```

```
top.Idle[
    Trigger=IncomingCall Leg: {} to top.AnswerWait,
    Trigger=Send Leg: {action=ConnectAction} to top.OnLineWait,
    Trigger=Digit Leg: {} to top.Idle],
    parent: top
top.AnswerWait[
    Trigger=Send Leg: {action=AnswerAction} to top.OnLineWait,
    Trigger=Digit Leg: {} to top.AnswerWait],
    parent: top
top.OnLineWait[
    Trigger=NoLine Leg: {action=DisconnectAction} to top.Idle,
    Trigger=Send Leg: {action=DisconnectAction} to top.Idle,
    Trigger=Digit Leg: {} to top.OnLineWait],
    parent: top
Sequence Transitions: [
 (001) Trigger=Digit Legs=[],(003)Trigger=Digit Legs=[],
 (004) Trigger=Digit Legs=[], (005) Trigger=IncomingCall Legs=[],
 (006) Trigger=Send Legs=[Leg(007): {action=ConnectAction}],
 (008) Trigger=Send Legs=[Leg(009): {action=DisconnectAction}],
 (010) Trigger=Send Legs=[Leg(011): {action=AnswerAction}],
 (012) Trigger=Send Legs=[Leg(013): {action=DisconnectAction}]] with index 0
========End of input data=========
```

The second part includes the trace of the run. The output is constructed in the following way:

```
Running <The Sequence diagram transition that is being run>
   Running <the states that are run in a hierarchical order>
      Running <State diagram leg which is tried from a persistent state>
         Trying <State diagram leg which is tried from a transient state>
```

A line which starts with "Trying < State diagram leg ", ends with "... failure" or "... success". This specifies the status of a particular transition search, i.e. given a Sequence transition and the current states, the status of whether it is legal to traverse this leg or not, is displayed. There can be many failure of this type in a successful run, as shown next.

```
Running (001) Trigger=Digit Legs=[] :
        Running top with currentStates[top.InitialState] :
        Running top.Idle (externally) :
                Running  Trigger=IncomingCall Leg: {} to top.AnswerWait.  ... failure
                Running  Trigger=Send Leg: {action=ConnectAction} to top.OnLineWait.  ... failure
                Running  Trigger=Digit Leg: {} to top.Idle.  ... success
Running (003) Trigger=Digit Legs=[] :
        Running top with currentStates[top.Idle] :
        Running top.Idle (externally) :
```

```
        Running  Trigger=IncomingCall Leg: {} to top.AnswerWait.  ... failure
        Running  Trigger=Send Leg: {action=ConnectAction} to top.OnLineWait.  ... failure
        Running  Trigger=Digit Leg: {} to top.Idle.  ... success
Running (004) Trigger=Digit Legs=[] :
    Running top with currentStates[top.Idle] :
    Running top.Idle (externally) :
        Running  Trigger=IncomingCall Leg: {} to top.AnswerWait.  ... failure
        Running  Trigger=Send Leg: {action=ConnectAction} to top.OnLineWait.  ... failure
        Running  Trigger=Digit Leg: {} to top.Idle.  ... success
Running (005) Trigger=IncomingCall Legs=[] :
    Running top with currentStates[top.Idle] :
    Running top.Idle (externally) :
        Running  Trigger=IncomingCall Leg: {} to top.AnswerWait.  ... success
        Running  Trigger=Send Leg: {action=ConnectAction} to top.OnLineWait.  ... failure
        Running  Trigger=Digit Leg: {} to top.Idle.  ... failure
Running (006) Trigger=Send Legs=[Leg(007):{action=ConnectAction}] :
    Running top with currentStates[top.AnswerWait] :
    Running top.AnswerWait (externally) :
        Running  Trigger=Send Leg: {action=AnswerAction} to top.OnLineWait.  ... failure
        Running  Trigger=Digit Leg: {} to top.AnswerWait.  ... failure
```

Let us look again at the last transition (number 006) of the Sequence diagram. The `ConnectAction` appears in the Sequence diagram, but the only transition for the trigger `Send` in the State diagram includes `AnswerAction` as its action. Thus the Sequence diagram is not consistent with the State diagram.

In the third part the results are shown, such as whether the Sequence diagram traversal has finished and whether the run has finished correctly or not.

```
Run Failed - Sequence Diagram is not suitable to the State Diagram.
```

4 Solution Description

This chapter describes the inputs, outputs, and the way of operation of our method below.

4.1 Inputs

This section describes the algorithm that we use in BVUML. The inputs to the algorithm are:

- The software project model description file. This file contains all the information about the UML model in the XMI format.
- The name of the Sequence diagram to be checked.
- The name of the object in the Sequence diagram one would like to check.

Each run of BVUML validates the consistency on a specific object in a Sequence diagram. BVUML should be activated several times for different objects of the same Sequence diagram. This way one can ensure that some interactions between objects in the Sequence diagram are possible, according to the corresponding state diagrams of the objects, whereas other interactions are not. It is especially vital when one wants to ensure that some action always happens before another.

4.2 The main idea of the solution

One can visualize the BVUML method as a hybrid Sequence-state diagram. From the input we know which State diagram we want to check and which object in the Sequence diagram we are about to check. Given a message in the Sequence diagram, it is interpreted in one of two ways in the State diagram:

- It should be a leg leaving persistent state, if the message is an incoming message to the checked object in the Sequence diagram.
- It is a leg leaving any state, if the message is an outgoing message from the checked object in the Sequence diagram. Usually this leg will emanate from a transient state, unless it is the case of UML completion transition, which will be described further.

A message can change the *current object state* of the state diagram. Notice that the *current object state* changes according to the previous transitions. A simple example of a coffee machine, where only a trigger changes the *current object state*, is shown in Figure 3.

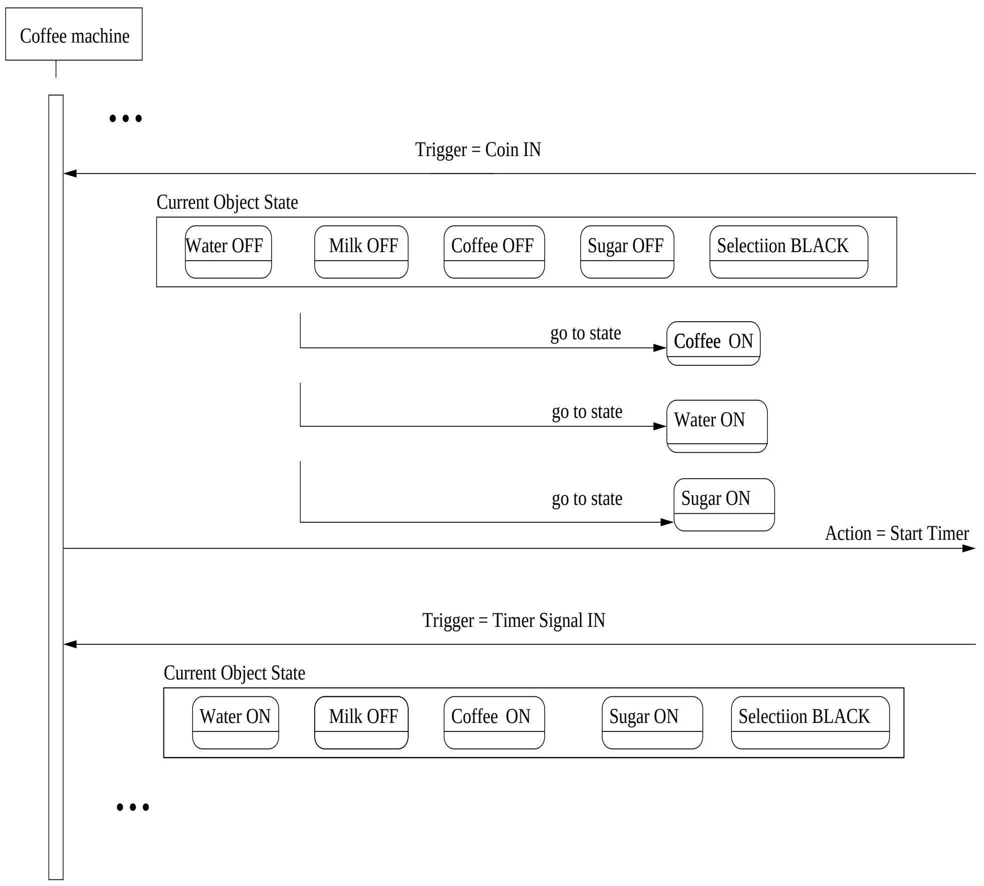

Figure 3: Hybrid Sequence-State diagram fragment.

Our algorithm iterates over the Sequence diagram transitions of a given object.

Each Sequence transition is compared with the set of state transitions that start from the *current object state*.

In order to find whether a Sequence diagram and a State diagram of an object obj_i, which is included in the Sequence diagram, are consistent, the following steps are taken.

The Sequence diagram transitions are traversed by the algorithm. For each transition, it checks whether there is any suitable state diagram transition, i.e. a transition that has the same trigger, same actions with identical guards, same action ordering as in the Sequence diagram transition. The suitable state transitions are searched from the *current object state*. If there is a suitable state transition, the iteration continues; otherwise an error message is issued. This process of iteration and comparison is called a *run* of Sequence diagram over the State diagram. This run can be done for each object in a Sequence diagram. The State diagram that is a part of the run is the State diagram associated with the class of the object.

4.3 The algorithm

For a given object in the Sequence diagram, the algorithm iterates over the Sequence transitions. Each Sequence transition is checked for a suitable State transition, which is looked for from the *current object state*. The *current object state* is updated upon successful lookup.

The lookup uses a breadth first search over the State diagram. The lookup starts by finding all *legs* emanating from a persistent state, that are suitable for the incoming *leg* of the Sequence diagram. If the leg comparison succeeds, the algorithm looks for a suitable state *transition* that starts with this edge. Following is a description of the lookup algorithm:

```
Input: a persistent state and a Sequence transition being checked.
Output: a suitable State transition or a failure status.
```

We chose to implement a BVUML step in a hierarchical way. State_machine_top (abbreviated in our examples to top), which is the uppermost state in every UML State diagram, is the first state in a run. For each composite (or concurrent composite) state we must know its current configuration, i.e, its *active (sub)states* – the inner states in which the thread of control is currently located. Each state in the current configuration might itself be a composite state and holds a configuration of its own. Thus, a hierarchy of states is created. The conjunction of all the leaf states in this hierarchy is the /it current object state. Each composite state recursively activates the step on its active substates (see CompositeState::step(Sequence transition)).

One of the main reasons we chose this method of implementation was to support State diagram *exceptions*. An exception is a leg emanating from some state located inside a concurrent state, to some other state outside that concurrent state. E.g, when an exception is received, the algorithm has to cause every thread of control inside of the concurrent state and its descendants to exit the state. (If the composite state was entered via a fork state, other rules, which are described in the next subsections, apply.)

Had we chosen a flat (non-hierarchical) approach, we would have been forced to iterate over a non-hierachical configuration of currently active states and to remove those inner states that are in the concurrent state from this configuration. Keep in mind that there might be some composite inner states inside the concurrent state. Control must leave them as well. In short, with the hierarchical approach the implementation of State diagram exception support is much easier.

```
/* Note that:
 - 'current states' include only persistent states.
 - CompositeState is a PersistentState.  */
CompositeState::step(SequenceTransition transition) {
// suitable for concurrent composite state as well
  foreach state in the 'current object state' of this composite state
    results.add(state.step(transition)) // PersistentState:step(SequenceTransition)
  if results have no successful entry then
    return 'none_found'
  elseif results have 2 or more successful entries, then
    return 'ambiguity'
  else
    return the transition stored in the successful entry.
}
```

Another consideration of similar kind was to use PersistentState::run(SequenceTransitions) (that does not exist now) instead of PersistentState::step(Transition). The former means that a composite state is given a set of Sequence transitions, and executes them till their completion. This means that there is no global "clock", which calls PersistentState::step() every "tick". We chose the latter, which is easier to understand and to debug. It is like a tick in languages like Esterel [Hal93].
In the former solution the caller of PersistentState::run() will have to know to which subset of the inner states the execution has reached.
/* The step function from a single persistent state S is described next. See a discussion on fine granularity step in Section 7.6. */

```
PersistentState::step(SequenceTransition transition) {
  PersistentState::gatherResults()
  if any state answer encountered means that multiple options (possibilities)
    were encountered during the transition search, return it.
  count the legal results count, and if more than one,
    return multiple options.
  if no correct results were encountered, return
    PersistentState::step() for the parent state.
  else return the single correct result.
}
```

We discuss (non)concurrent implementation in Section 7.4. Completion transition is elaborated on in Section 7.8.

```
Collection PersistentState::gatherResults(
    SequenceTransition transition)
{ // All legs emanating from a persistent state should have a trigger,
  // apart from a leg that starts a completion transition.
  for each emanating leg that has a trigger
    PersistentStateLeg::step(transition, this_state);
  collect the results into a collection and return them.
}

PersistentStateLeg::step(SequenceTransition seqTrans,
    PerstistentState S)
{ compare the trigger, guard of this leg
  for conformance to the incoming leg of the seqTrans;
  3 possible actions - the exit action of the original
  state, the leg action, and the entry action are also
  checked.
  if OK then {
    return call BFSSearch(next state of this leg,
            outgoing legs of the Sequence transition).
  }
}
```

Design decisions concerning history state can be found in Section 7.8, forks and joins are elaborated on in Section 7.3, and saturateable state and count are discussed in Section 7.1

```
PersistentStateLeg::BFSSearch(State fromState, OrderedCollection
legs)
// legs that form the Sequence diagram transition.
{
  // initially, includes only fromState parameter.
  bfs_states = { fromState };

  foreach state S in bfs_states
  {
      if (state S is a HistoryState) then
        add all remembered states to bfs_states.
      remove S from bfs_states.
      continue.
    }

    if (state S is a JoinState) then {
      increment saturateable count of S.
      if S is not saturated then
        continue.
    }
    if (state S is a FinalState) then {
      increment the saturateable count of the parent
```

```
        state of the S.
      continue.
    }
    // traverse legs.
    if (S is a fork state) then {
      compare all legs emanating from S with the
      upcoming legs in the ‘legs‘ parameter.
      if (not all equal) {
        continue.
      }
      else {
        remove all the compared legs from ’legs’.
        insert the target state
          of each leg L into bfs_states.
      }
    }
    else {
      foreach outgoing leg L that has no trigger {
        compare L with the current
        Sequence transition leg L’.
        if L == L’ then {
          remove L’ from ’legs’
          //the function parameter.
          insert the target state of L into bfs_states.
        }
        else
          continue.
      }
      if some leg was removed, remove the S from
        bfs_states.
    }

    if ((bfs_states include only persistent states) and
        (‘legs‘ are empty)) then
      return bfs_states with status ’executed’.
    else if ((there no leg was removed) and
        (S was not a history state)) then
        // a history state just
        // delegates execution till the next iteration
      return status ’no_options’.
  }
}
```

What values should the algorithm return? We found out that `step executed`, `no options`, and `multiple options` suffice.

5 A Run Example

In this chapter we elaborate on a more detailed, and complex, example. The stages in BVUML input gathering, BVUML usage, and a BVUML output are shown below.

1. Compose a project using ArgoUML: Draw Sequence and State diagrams.

2. Extract an XMI file from the project file using unzip. This file contains all the data about the constructed UML model, including all the Sequence diagrams and all the state diagrams.

3. Supply this XMI file as a parameter to BVUML, together with the Sequence diagram name and an object in the Sequence diagram one wants to check.

4. Invoke BVUML on the input.

5. BVUML iterates over the Sequence diagram transitions of a given object. Each Sequence transition is compared with the set of state transitions that emanate from the *current object state*. If there is a suitable state transition, whose function calls, conditions, triggers, etc are compatible with the Sequence diagram transition, the iteration continues. Otherwise, an error message is issued.

6. The tool outputs the run discussed above and prints the final result - whether run finished correctly or not.

We use a Navy Command and Control (C&C) system example to demonstrate how BVUML works. The purpose of the system is to track all objects that appear in a warfare arena. The data update scenario of a given ship in the C&C is shownṪhis data describes all available knowledge of ships, aircrafts, and submarines in the area, such as their hostility, country of origin, course, speed, location, model, etc. The data update is performed via sensors, such as radars and radar detectors (radets), and via intelligence services, using satellite connection. This update is performed every specified period of time. We are interested only in part of the functionality of the ship – the object that handles target update, i.e., update of all the ships, aircrafts, and submarines data.

There are two types of updates: a local update and a global update. A local update is a data update that does not involve any communication with objects external to the ship. A global update is a data update that involves communication with external objects, such as satellites. The full results are shown in appendix A. Please note that ArgoUML displays the stimulus names (i.e. Sequence diagram message names, triggers), but it does not show guards or actions on either the Sequence or State diagram. Therefore guards and triggers are also not shown in the Sequence diagram of the system. They are, however, saved in the XMI file that is given as an input to the BVUML. The tool outputs them for user convenience in the first part of the run.

Since we wanted to emphasize selected parts of the output, only a subset of these results is shown below.

5.1 The Sequence Diagram

In the following scenario example we are interested in the messages that the `TargetUpdateInstance`, the object that handles the data update, receives. The objects that interact with `TargetUpdateInstance` are called `The World` in the Sequence diagram. One might think of them as an environment in which the ship, our main object of interest, resides. In reality, `The World` might be split to operator, commander, and daemon objects, for example. We describe the incoming stimuli (triggers) below.

Message 1.0 sets the update option to global update.
Message 1.1 actually triggers the update process.
Messages 1.2 to 1.6 receive specific update requests from various target detectors, such as radar detectors and radars.
Messages 1.7 terminates radar and radar detector updates.
Message 1.7.1 specifies that intelligence source is contacted.
Message 1.7.2 sends confirmation to the intelligence source.
Message 1.7.3 writes the event from 1.7.1 to a system log.
Message 1.7.4 shows that the update process with the intelligence source has finished.

When all the updates have ended, we turn to notify other ships and to write all the data to a database.
Message 1.7.6 is a request for opening new transaction. This is sent to a transaction manager, for example.
Message 1.9 is the response to this request.
Message 1.9.0 describes data write to some database.
Message 1.9.1 a request to transmit the collected data to other ships. The decision whether to transmit is done by an outer object.
Message 1.9.2 is another log request.

Message 1.9.3 establishes FM connection for the forthcoming transition.

Message 2.0 specifies an asynchronous write completion and transaction closing event, from the database access proxy class.

Message 2.0.1 specifies the FM connection establishment event.

Message 2.0.1.0 sends data over the FM connection.

Message 2.0.2 specifies the end of data transfer via the connection.

Message 2.0.3 specifies the resume action – time allocated for the entire operation has passed.

Message 2.1 marks the end of execution.

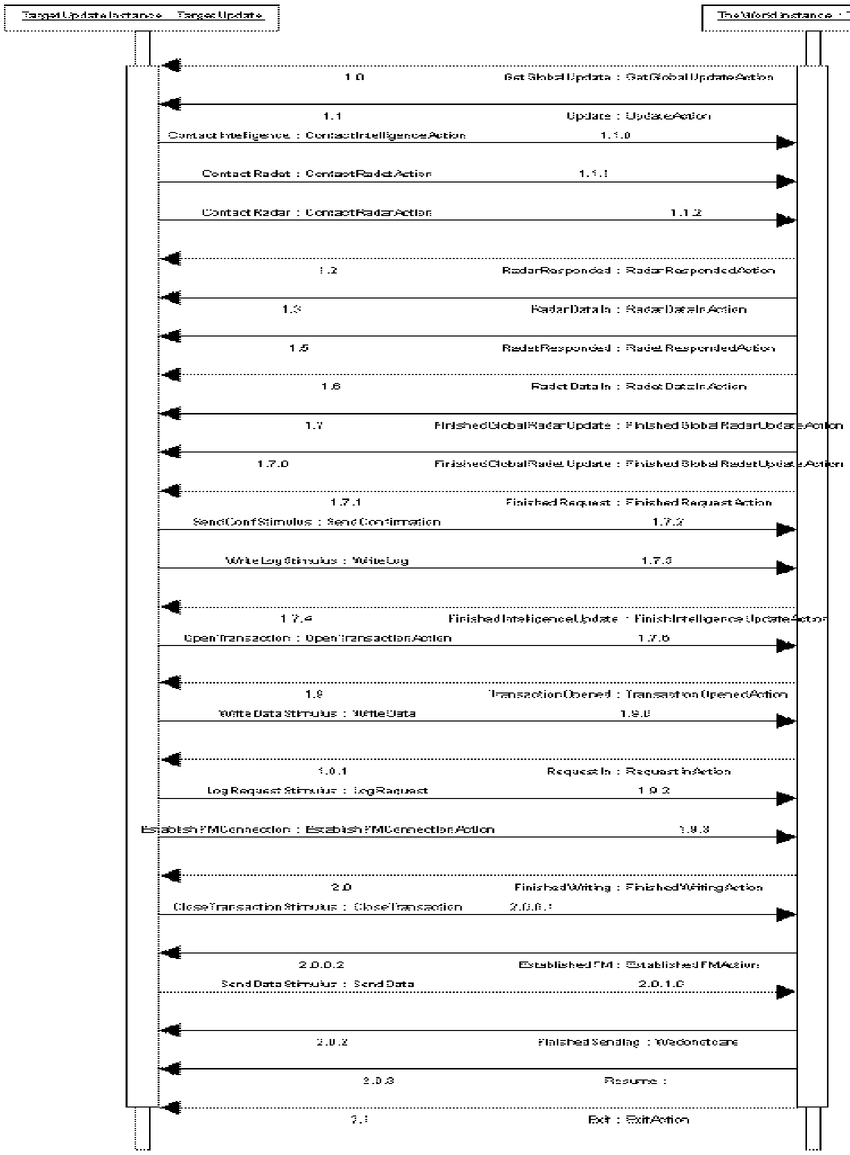

Figure 4: Target Update Sequence Diagraam

5.2 The State Diagram

The State diagram of the example in Figure 5 has 3 main stages.

In the first stage the user chooses the kind of data update. In the second stage the update is executed. This includes update from on-ship sensors, and possibly an intelligence data update. The third stage includes the logging of the results and then sending to other C&C centers. Note that the `resume` trigger is shown only with states in stage 3 in order to keep the diagram simple. Ideally the `resume` should emanate from all states (or their parent states). The resume indicates that the time allocated for the entire operation has elapsed (i.e., a timeout event). If this happens, the target update is aborted.

Stage 1 is a composite state, whereas stages 2 and 3 are concurrent composite states. ArgoUML does not yet support concurrent composite states. BVUML distinguishes a concurrent composite state from a non-concurrent composite state by the number of initial states in it. The non-concurrent composite state includes only a single initial state, whereas a concurrent composite state includes several initial states.

Note also that UML 1.4 includes fork and join $s$tates in the State diagram.

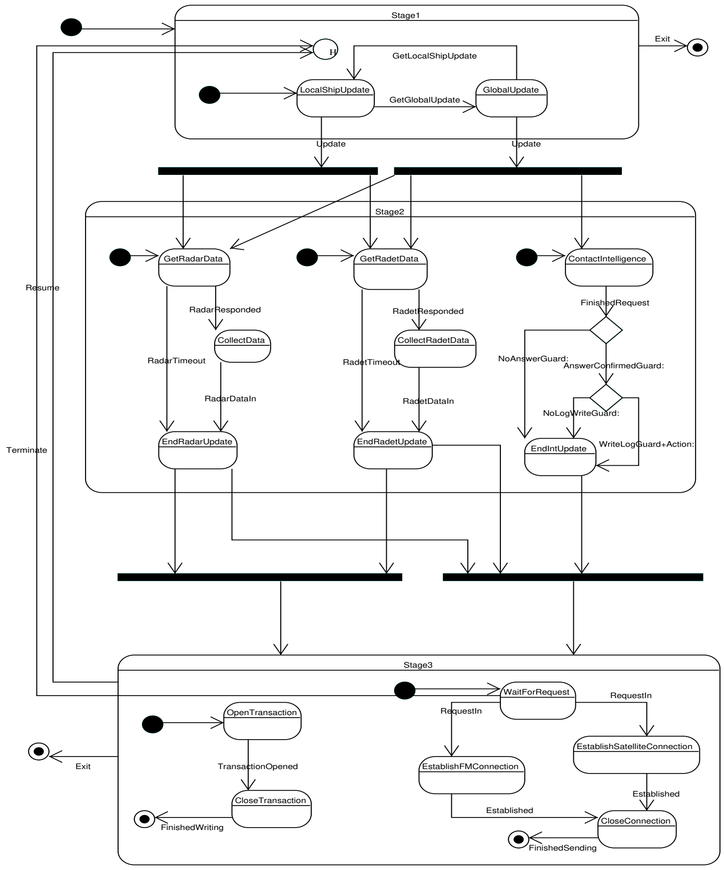

Figure 5: Target Update State Diagram

6 Run Results

Run results are composed of three parts. The first part describes the input data, the second part depicts execution of the transitions, and the third part sums up the results.

6.1 Part One

As mentioned, ArgoUML, or any other UML standard compliant modelling tool, saves all the data about the UML model in a .xmi file. In particular, the data about the state and Sequence diagrams resides in this file.

The first part of BVUML output expresses the input data extracted from the .xmi file, which is one of the parameters to the program. This part is used mostly for making sure the XMI parsing worked well. It includes the states and their legs, and the Sequence transitions. One can also see if something diagram designer has entered is missing in the State diagram, or vice versa. This part is mostly used to see whether all the data one has entered in the diagrams appears in XMI and was extracted from it. Recall that guards and actions do not appear in the figures. Following are some lines from this part.

The following output example shows the Sequence transition that starts with FinishedRequest trigger, and has two legs. See legs 1.7.1-1.7.3 in the Sequence diagram.

```
Trigger=(1.7.1)FinishedRequest Legs=[Leg (1.7.2):
{action=SendConfirmation, guard=AnswerConfirmed}, Leg (1.7.3):
{action=WriteLog, guard=WriteLog}]
```

The following output example shows the legs of state_machine_top.Stage2.Answered state, which is the first state below state_machine_top.Stage2.ContactIntelligence.

```
state_machine_top.Stage2.Answered
[Leg: {action=SendConfirmation, guard=AnswerConfirmed} to
state_machine_top.Stage2.WillingToAnswer,
// the second state below state_machine_top.Stage2.ContactIntelligence.
Leg: {guard=NoAnswer} to state_machine_top.Stage2.EndIntUpdate],
parent: state_machine_top.Stage2 // the composing state.
```

6.2 Part Two

The second part of the output includes the trace of the consistency validation between the given State and System diagrams. A subset of it is shown below. The output is constructed in the following way:

```
Running <The Sequence diagram transition that is being run>
    Running <the states that are run in a hierarchical order>
        Running <State diagram leg which is tried from a persistent state>
            Trying <State diagram leg which is tried from a transient state>
```

The output is indented for clarity only. This way one can easily spot a start of a new step, for instance. One can see that BVUML starts with the first trigger, and continues from there.

Comments are marked by "//". Is the following description, a transition is identified by the id of its trigger, as shown in the output of the run. The full output can be found immediately after the following comments.

There are many points of interest in the BVUML output.

- Notice the transition that starts with leg 1.0. This is the first line of the output.
- In the same transition, I introduce the concept of running the state externally. This is very important for composite states. It means running the state as a simple state, i.e, not looking inside.
- The transition that starts with leg 1.1 begins its run from top.Stage1.GlobalUpdate – the state that was reached at the previous transition.
- In the same transition, after a fork state is reached, the flow of control continues within this transition. This has several implications on the BVUML algorithm.
- Note also in the same transition, i.e, the one that starts with leg 1.1, all the actions in the State diagram successfully match the actions in the Sequence diagram.
- The *current object state* inside the state_machine_top (named here "top") state includes 3 states, which are children of top.Stage2. The reason why these 3 state are not held in top.Stage2 is discussed later in the thesis.
- In transitions 1.3 to 1.4 one can see how the number of inputs is counted in the join state. It eventually becomes saturated. This term is very important to the algorithm, and is discussed in chapter 7. In transition 1.7.1 one can see detailed comparison of guards and action, as well as join saturation. The flow of control continues immediately to stage3.
- The current state array inside the state_machine_top (named here as top) includes stage3, whose array includes 2 states. This is different from what was shown in stage2, where the current inner states we kept inside the states array of the state_machine_top state. The explanation will be given in the BVUML algorithm description.
- In the transition that starts with message 2.0 one can see that the saturateable count is also held for composite state (`state3`). This term is discussed in chapter 7. `state3` becomes saturated by leg 2.0.2.
- Leg 2.0.3 shows how history state works.
- State_machine_top (named here as top) is saturated in the final transition.

Some parts of the BVUML output are presented below. Note that "state_machine_top" is replaced with "top" for ease of representation in the following BVUML output:

```
   | Running Trigger=(1.0)GetGlobalUpdate Legs=[] :  // the first leg and the corresponding leg number
|        Running top with currentStates[top.Initial1] :  // the first state
|        Running top.Stage1 with currentStates[top.Stage1.Stage1Initial1] :  //get into initial state of
Stage1
|        Running top.Stage1.LocalShipUpdate (externally) :  // externally means looking at the outgoing
legs of a composite state
|               Running Trigger=GetGlobalUpdate Leg :    to top.Stage1.GlobalUpdate.  ...  success :
|                                                    Searching for completion transition in
top.Stage1.GlobalUpdate ...
|               Running Trigger=Update Leg :    to top.Fork1.  ...  failure
| Running Trigger=(1.1)Update Legs=[] :
|        Running top with currentStates[top.Stage1] :
|        Running top.Stage1 with currentStates[top.Stage1.GlobalUpdate] :  // the state from the previous
transition
|        Running top.Stage1.GlobalUpdate (externally) :
|               Running Trigger=GetLocalShipUpdate Leg :    to top.Stage1.LocalShipUpdate.  ...  failure
|               Running Trigger=Update Leg :    to top.Fork2.  ...  success :
|         // continue, because Fork2 is a transient state.  All its outgoing legs are tried.
|                Trying Leg :    to top.Stage2.ContactIntelligence ...  success.
|                Trying Leg :    to top.Stage2.GetRadetData ...  success.
|                Trying Leg :    to top.Stage2.GetRadarData ...  success.
| Leaving a composite state top.Stage1

// skipping ..
| Running Trigger=(1.3)RadarDataIn Legs=[Leg (1.4):  action=FinishedGlobalUpdateEffect,
guard=FinishedGlobalUpdateGuard] :
|        Running top with currentStates[top.Stage2.ContactIntelligence, top.Stage2.GetRadetData,
top.Stage2.CollectData] :
| // These states are held in top, not in top.State2.  This is because we came into trop.Stage2 via fork.
This is discussed
| // in depth in the design decisions section.
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.GetRadetData (externally) :
|               Running Trigger=RadetResponded Leg :    to top.Stage2.CollectRadetData.  ...  failure
|               Running Trigger=RadetTimeout Leg :  action=InformUserRadetFailed to
top.Stage2.EndRadetUpdate.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.CollectData (externally) :
|               Running Trigger=RadarDataIn Leg :    to top.Stage2.EndRadarUpdate.  ...  success :
|                                                    Searching for completion transition in
top.Stage2.EndRadarUpdate ...
|                Trying Leg :  action=FinishedRadarUpdateEffect, guard=FinishedRadarUpdateGuard to
top.Join1 ...  failure.
|                Trying Leg :  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard to top.Join2
```

```
... success.
|                        ... success - join count incremented top.Join2// a flow of control has reached top.Join2

// skipping ..
| Running Trigger=(1.7.1)FinishedRequest Legs=[Leg (1.7.2): action=SendConfirmation, guard=AnswerConfirmed,
Leg (1.7.3): action=WriteLog, guard=WriteLog, Leg (1.7.4): action=FinishedIntelligenceUpdateEffect, Leg (1.7.5):
action=StartingStage3] :
|         Running top with currentStates[top.Stage2.ContactIntelligence] :
|         Running top.Stage2.ContactIntelligence (externally) :
|                Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  success :
|                 Trying Leg :  action=SendConfirmation, guard=AnswerConfirmed to top.Stage2.WillingToAnswer
... success.
|                 Trying Leg :  guard=NoAnswer to top.Stage2.EndIntUpdate ...  failure.
|                 Trying Leg :  guard=NoLogWrite to top.Stage2.EndIntUpdate ...  failure.
|                 Trying Leg :  action=WriteLog, guard=WriteLog to top.Stage2.EndIntUpdate ...  success.
|                                                     Searching for completion transition in top.Stage2.EndIntUpdate
...
|                 Trying Leg :  action=FinishedIntelligenceUpdateEffect to top.Join2 ...  success.
|                        ...  join is saturated top.Join2
|                 Trying Leg :    to top.Stage3 entry=StartingStage3 ...  success.
| Running Trigger=(1.9)TransactionOpened Legs=[Leg (1.9.0):  action=WriteData] :
|         Running top with currentStates[top.Stage3] :
|         Running top.Stage3 with currentStates[top.Stage3.Stage3Initial1, top.Stage3.Stage3Initial2] :
| // here top.Stage3 hold the active states, unlike the fork example mentioned in the beginning (leg 1.3)
|         Running top.Stage3.OpenTransaction (externally) :
|                Running Trigger=TransactionOpened Leg :  action=WriteData to top.Stage3.CloseTransaction.  ...
success :
|                                                     Searching for completion transition in top.Stage3.CloseTransaction
...
|         Running top.Stage3.WaitForRequest (externally) :
|                Running Trigger=RequestIn Leg :  action=LogRequest, guard=FM to top.Stage3.EstablishFMConnection.
...  failure
|                Running Trigger=RequestIn Leg :  action=LogRequest, guard=Satellite to top.Stage3.EstablishSatelliteConnection.
...  failure
|                Running Trigger=Terminate Leg :    to top.Stage1.Stage1History.  ...  failure
|         Running top.Stage3 (externally) :
|                Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  failure
|                Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure

// skipping ..
| Running Trigger=(2.0)FinishedWriting Legs=[] :
|         Running top with currentStates[top.Stage3] :
|         Running top.Stage3 with currentStates[top.Stage3.CloseTransaction, top.Stage3.EstablishFMConnection]
:
|         Running top.Stage3.CloseTransaction (externally) :
|                Running Trigger=FinishedWriting Leg :    to top.Stage3.Stage3Final1.  ...  success :
|                               ...  success - saturateable count of top.Stage3 incremented
|         Running top.Stage3.EstablishFMConnection (externally) :
|                Running Trigger=Established Leg :  action=SendData to top.Stage3.CloseConnection.  ...  failure
```

```
|        Running top.Stage3 (externally) :
|               Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  failure
|               Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure

// skipping ..
| Running Trigger=(2.0.2)FinishedSending Legs=[] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.CloseConnection] :
|        Running top.Stage3.CloseConnection (externally) :
|               Running Trigger=FinishedSending Leg :    to top.Stage3.Stage3Initial2.  ...  success :
|                              ...  success - top.Stage3 is saturated.
| Running Trigger=(2.0.3)Resume Legs=[Leg (2.0.4):  action=Stage3Completed] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[] :
|        Running top.Stage3 (externally) :
|               Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  success
:
|                Trying Leg :    to top.Stage1.LocalShipUpdate ...  success.
|               Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure
| Running Trigger=(2.1)Exit Legs=[] :
|        Running top with currentStates[top.Stage1.LocalShipUpdate] :
|        Running top.Stage1.LocalShipUpdate (externally) :
|               Running Trigger=GetGlobalUpdate Leg :    to top.Stage1.GlobalUpdate.  ...  failure
|               Running Trigger=Update Leg :    to top.Fork1.  ...  failure
|        Running top.Stage1 (externally) :
|               Running Trigger=Exit Leg :    to top.Final2.  ...  success :
|                              ...  success - top is saturated.
|
| Input traversal finished.
| Run finished correctly.
```

6.3 Third Part

In the third part the results are shown. One can see whether the Sequence diagram traversal has finished, what the current states are (this can be seen from the last step as well), and whether the run has finished correctly or not.

```
Input traversal finished.
Current States : state_machine_top
Run finished correctly.
```

Note that usually, if the run has failed and one wants to detect the problem, he or she has to look at the last transition of the second part, and at the third part. It is fairly easy to understand what went wrong. We used the tool even to find errors in our examples as they have been built. There have been much more then we originally thought. Errors have been detected in examples that are much simpler than this one. Here is an example of a failing run (of some other XMI file, with different Sequence diagram and same State diagram):

```
Running Trigger=(1.9.1)RequestIn Legs=[Leg (1.9.2):
{action=LogRequest, guard=FM}] :
        Running state_machine_top with currentStates[state_machine_top.Stage1.LocalShipUpdate] :
        Running state_machine_top.Stage1.LocalShipUpdate (externally) :
                Running  Trigger=GetGlobalUpdate Leg : {} to
                        state_machine_top.Stage1.GlobalUpdate.  ... failure
                Running  Trigger=Update Leg : {} to state_machine_top.Fork1.  ... failure
        Running state_machine_top.Stage1 (externally) :
                Running  Trigger=Exit Leg : {} to state_machine_top.Final2.  ... failure
Run Failed - Sequence Diagram is not suitable to the State
Diagram; At : RunAttributes: [state_machine_top] No options

Not all the Sequence transition legs were processed

Input traversal aborted. Not all Final (End) states were reached.
```

One can see that the run fails. There is no suitable trigger in the State diagram because `RequestIn` does not appear as a trigger of any State diagram legs. The Sequence diagram transition holds `RequestIn` as the trigger, whereas the State diagram legs emanating from the current state (state_machine_top.Stage1.LocalShipUpdate) are `GetGlobalUpdate`, `Update`, and `Exit`, and not `RequestIn`.

7 Design decisions

In this chapter we discuss some of the decisions we made while implementing BVUML.

7.1 Saturateable states

The original idea was to write the search algorithm so that it returns a "final state reached" value upon arrival to the final state. Remember that an enclosing state of a final state is a persistent (composite) one. Thus the execution of the Sequence diagram transition should come to an end, and the algorithm is terminated. This means that the algorithm can return a per-state result, after checking whether all the transition legs were traversed. The enclosing state might then examine the search algorithm activation results, and update its internal attributes accordingly. Thus the enclosing state can know which of its parts has reached the final state. This solution was not chosen, however, as we came up with another idea for the implementation – the *saturateable state*.

How does one know that a join state has received all the threads of control from all its inputs? A similar problem exists with final states inside a concurrent state. One has to know that all the final states have been reached. A saturateable state holds a counter of the control threads that enter it. This count is updated upon each state entry. In this way a join knows how many transitions have already entered it. A final state also notifies its parent when a transition enters it.

7.2 Forks implementation

Any set of transitions from a fork must enter the inner states of a concurrent composite state, and must leave the composite state via a join. The question is who should hold the *current active states* configuration, which becomes active after a fork state is left?

It is very inconvenient to hold the configuration inside the fork state, because a fork state is a transient and non-composite state. Note that a run of the fork state means a run till the next persistent state, and not a run till a join state, like one could expect from a composite state. Moreover, it is not a natural approach to call the CompositeState::step() function of the algorithm (cf. Chapter 4) in the fork state object, because this should really be called only in a persistent state, by the definition of the CompositeState::step() function.

Holding the *current active states* configuration in the concurrent state, which is the immediate parent of the state that are reached by the first legs from a fork state (like the Concurrent Composite State in Figure 6), is not such a good idea as it may seem. This concurrent state has to know whether its inner states became active via a non-fork transition to itself, or via a direct transition to them from forks. This is important to support exceptions. Notice also the case in which after a transition from a fork, all final states in the parent state are reached. This is different from a similar case in which the inner states were activated directly via the parent. In the former case, exiting must be performed using joins, whereas in the latter case, exiting the parent state is a common exiting from a concurrent composite state.

The transition from a fork is toward the inner states, not toward their immediate parent. It is not simple for the concurrent state to know whether its inner states became active via transition from the fork. From the point of view of software development, a state should not know the transition that led to it in the current step.

In the case of a transition from a fork to inner states, our solution is to hold the active inner states in the parent of the fork state. In this way every *current states configuration* in any composite state can potentially hold several descendant states. Even a simple (non-concurrent) composite state may have an inner concurrent state, and a fork/join pair that is associated with it.

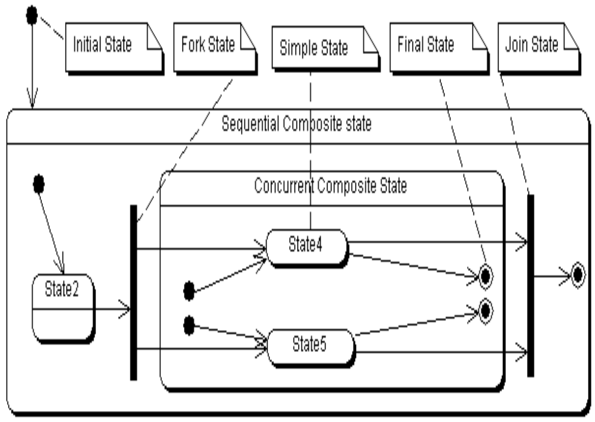

Figure 6: Target Update State Diagram

7.3 Forks and joins as states

An interesting question was how to implement forks and joins in BVUML? In some UML standard drafts, they are defined as transitions to a concurrent state, that is, as transitions and not as states.

We treat each arrow in the State diagram as a leg. Had we allowed "compound legs", i.e. legs like fork and join, then each leg would have returned a set of next states on each leg traversal in a run, instead of a single next state, which is easier to handle. Moreover, in order to know when to exit a join, we must count the number of inputs through which flows of control have already reached a join leg, i.e. keeping state information in a transition. It is much more intuitive to keep such a count for a state, rather than for a leg. The latter would require doing special leg operations in the algorithm, which is inconvenient. The XMI file that is extracted from ArgoUML [RVR$^+$] also treats forks and joins as states.

7.4 Non-determinism and per-step concurrency

What happens if, at some point during a transition traversal in the State diagram, two or more legs have the same name, and that leg name is the next one in the Sequence diagram? Should all legs be traversed, or should an error be raised?
When two candidates exist for a transition from the same persistent state, or when the candidates are from two different concurrent states, the algorithm issues an error. The second decision can be easily changed by rewriting CompositeState:step() to return success in a case of multiple options.

7.5 Single threaded solution

At the beginning, when considering how to tackle a run till a completion of a concurrent state, we thought that a parallel algorithm would be a suitable solution. But this is not as simple as it may seem. Besides the usual run conditions, there is the case of an exception. How will the thread exiting the concurrent state notify other threads? We found out that in our case a non-concurrent algorithm is more beneficial and much more simple compared to a concurrent one.

7.6 Fine granularity step

We considered using a per-leg step, instead of a per-transition step. Though the former allows more control, this control is not needed here, and it makes the program harder to manage, write, and understand.

7.7 Composite states

A sequential (a simple) composite state can be considered a concurrent composite state with only one orthogonal region. This is also the way we treated it in BVUML.

7.8 Some implementation notes

- A step result of an atomic state may return several states. This happens if there are transient forks in the outgoing transition of the atomic state.
- BVUML does not support purely syntactical states, such as a stub state.
- A concurrent state is a persistent state, thus only legs including a trigger can emanate from it (apart from a leg that starts a completion transition). This is exactly the

opposite of a join state, which is a transient one.

In case of a concurrent state, the search for a suitable State transition ends when the concurrent state is saturated. With a join, the transition continues in the same step.

- Each composite state in BVUML has a reference to a history state object. This reference might be null. When the algorithm detects that a concurrent state exists, it updates the relevant history state by the last configuration of its persistent states. This update dynamically changes the outgoing legs of the history state. A history state is a transient one, implemented as a state with outgoing legs targeted toward the last remembered configuration. In this way, when the history state is entered, the flow of control is directed to the last remembered inner states.

- BVUML keeps all the guards as strings. We use string comparison (after removal of space) to find out whether a transition is enabled. This is sufficient for our needs.

- UML specifies a format in which all the data in the diagrams can be held in one file. This format is called XML Metamodel Interchange (XMI).

  UML XMI file format specifies entirely different data structures for the Sequence diagram and for the State diagram. Also the collaboration diagram data structures are entirely different from the ones used by the Sequence diagram. This is contrary to the fact that Sequence and collaboration diagrams show the same data, but with different emphasis. Some tools, like Rational Rose, even convert from a Sequence diagram to collaboration diagram, and vice versa. This thesis also introduces a strict definition of Legs, Transitions, and Transient/Persistent states. In our opinion, UML draft is very obscure about these things. What is especially bothering is *transition* usage. Most of the times *transition* in the UML standard draft is identical to *leg* in this thesis. Another issue is that there is at most one guard per message in the Sequence diagram. The same message is associated with the outgoing link of one object in the Sequence diagram, and with the incoming link of another object. (Those links are shown as one arrow between these 2 objects.) Now, UML specifies dispatch action and a trigger, but has no guard that is assigned to the incoming object. There is only a dispatch guard.

  The pseudo state definition is copied from UML standard. We rarely use it, because the transient and persistent state definitions are better, to my mind. A *state* usually means persistent state in UML standard. The standard treats each pseudo state as a *state vertex*, meaning it is a transient state. However, this rule is not enforced throughout the standard. A fork *state*, is not called a fork *state vertex*, for example.

We think that the Sequence, Collaboration and State diagrams have a lot in common, much more than it is defined in XMI. Maybe XMI should be built in a bit different way, according to the definitions given above.

8 Related Work

In this chapter we survey other tools and theoretical work, that deals with the behavioral verification of UML diagrams.

8.1 HUGO project

A prototype tool, that is designed to automatically verify whether the interactions expressed by a collaboration can indeed be realized by a set of state machines. It compiles state machines into a PROMELA model and collaborations into sets of B"uchi automata ("never claims"). The SPIN model checker is called upon to verify the model against the automata. Hugo is always executed with the following arguments: The name of the XMI-file that contains the model, the name of the file that the Promela code should be generated into, the name of the package the classes are in and the names of these classes. A collaboration is provided by the run time option -c followed by the name of a text file.

Hugo was tested with XMI files of UML models that were made with the free CASE-tool ArgoUML v0.8.1a (see [RVR+]).

A composite state that contains the text ' (concurrent) ' in its name is considered to be a concurrent composite state.

Hugo can only check state machines that are syntactically correct, according to the OMG Unified Modelling Language Specification, Version 1.3. Hugo may abort upon a syntactical incorrectness, yet it does not recognize all syntax errors. Currently unsupported UML constructs: internal transitions, event parameters, signal hierarchies, deferred events, synch states, choice states, change events, time events.

The project involves translation to PROMELA and may require additional knowledge from UML designers. Using collaboration information in the form of rather simplified text file could be substituted by direct retrieval from the XMI file containing the model.

For full documentation of this project see [hug] and [TS01].

8.2 IIOSS Model Debugging Facility (MDF)

IIOSS is a software development environment that offers a tool whose multiple facets include a UML editor (MEF), a model simulator, an interface builder, a file converter and an object-oriented database. MDF enables one to simulate a UML model she has created using MEF. This simulation can help to verify whether her model exhibits required behaviors as she wanted to design with MEF.

This project is quite recent, open source and under development. IIOSS documentation is for the time being available only in Japanese, but from the source code it is clear that, though the simulations are done in separate for each kind of diagram, the tool just follows the UML logic and does not generate any intermediate language code.

For full documentation of this project see [iioa] and [iiob].

8.3 Prosasim UML simulator

A simulator helps in detecting design errors in very early phase, which minimizes the costs of corrective actions. Prosasim presents the real behavior of the UML model graphically. Prosasim is an integrated component of Prosa UML modeler. When user is using Prosasim, he runs the application and in real-time follows the execution in the UML model context. In Prosasim simulator the user selects interactively from UML model those diagrams, which he wants to simulate. The execution of those parts of the UML model, which have already been simulated and tested, proceed with full speed. The progress of the execution is animated with a simulator token. The designer may set breakpoints to critical parts of the model and in breakpoints he has an option to examine and set attribute values. Prosasim simulator supports simulation of C++, C and Java applications. Because the simulation is based on native source code, visual model and the developed application will behave exactly in the same way.

This is one of many component simulation tools based on generated code.

For full documentation of this project see [Ins].

8.4 PecSee project

The main aim of the PecSee project, led by Jon Whittle and Johann Schumann, is to develop techniques and tools for maintaining consistency and resolving ambiguities between UML models. An algorithm has been developed which supports the design process by generating statechart designs automatically from scenarios (expressed as Sequence diagrams), taking into account information present in the class diagrams. The algorithm first takes a collection of Sequence diagrams and generates automatically a state machine corresponding to the behavior in the Sequence diagrams.

The project introduces ideas in favor of using state and Sequence diagrams relationship.

For full documentation of this project see [SW01]

8.5 LSC: Live Sequence Charts

This is an early theoretical work with emphasis on one of the central problems in the behavioral specification of systems, and the problem in object oriented specification: Relating interobject specification to intraobject specification (grand dichotomy of reactive behavior) and an introduction of Sequence diagrams interspersed with hierarchical LSC charts.

The BVUML run diagram includes names of states in the *current states* set as labels after each link of a Sequence diagram. This can be seen as an expansion of the *state-marks in Sequence diagram approach*, which was described in the LSC article.

For full documentation of this project see [DH01]

9 Conclusions and Future Work

9.1 Partial legs ordering in a Sequence diagram

Consider running a Sequence transition. We might not care about the order of some of the outgoing legs. We thought of some expansion of UML where one can actually visually set **partial ordering** of the Sequence diagram legs. Legs among which the order is not important can emanate from the same spot in the Sequence diagram. (Note that this should suffice for most practical uses). This complicates greatly the running algorithm, but note that it still remains BFS. This is not implemented in the project.

9.2 Compound initial transition

The UML standard is somewhat vague about the transition concept, as we mentioned earlier. We assumed that there is a single empty transition from an initial state to a persistent state. However, UML also uses the term compound transition. We might have a transition emanating from the initial state, which includes several choice states for example (which will make it several transitions, not the standard). Moreover, it might even end outside the composite state. Note that in this case it's much harder to locate the flow of control, because it's cumbersome to make hierarchical running algorithm like this. One might consider global current states instead of current states per every composite state, in this case. This makes exception control harder and time-costly, though.

One will have to detect exception, and then undo the steps of the states that were in the same father of the exception receiving step. Note also that the exception can go up several levels in the hierarchy. Then the steps taken in these levels will have to be undone. This makes exception much harder to implement.

9.3 Several actions for a leg

In the beginning I allowed several actions per leg. That made the code a bit harder to understand, because we were dealing with arrays now. We've done this because we did not know the standard well enough that time, and because we wanted to allow ordered/non-ordered inputs in the Sequence diagram. We recommend encapsulating these arrays in objects, and then to treat each entry in the array as a leg, is you are going to implement something similar.

9.4 Summary

BVUML provides a simple solution for the UML dynamic diagrams consistency problem. It is more than a boolean result consistency checker; it is a tool that enables fast error detection and recognition. It supports standard UML, and was checked with the ArgoUML CASE tool.

Its main advantage is that it is straightforward to use and the output is easy to understand, unlike some other similar tools, and the input is given in a straightforward form of a Sequence diagram. No first or second order logic knowledge is needed. Such a knowledge is not always present at the client side (analysts and designers).

Its main disadvantage is that it gives one information about *specific* runs that are designed in a Sequence diagram, and not any possible runs. The solution of this would probably require solid computer science logic background at the client side.

Adding a collaboration diagram to the BVUML should be very simple indeed. It was not added because it has no added value to the research.

10 Appendix A: Full output example

```
| BVUML - Behavioral Verification of UML diagrams.
|
| ======== Input data =========
| Sequence Transitions:
| Trigger=(1.0)GetGlobalUpdate Legs=[]
| Trigger=(1.1)Update Legs=[]
| Trigger=(1.2)RadarResponded Legs=[]
| Trigger=(1.3)RadarDataIn Legs=[Leg (1.4):  action=FinishedGlobalUpdateEffect,
guard=FinishedGlobalUpdateGuard]
| Trigger=(1.5)RadetResponded Legs=[]
| Trigger=(1.6)RadetDataIn Legs=[Leg (1.7.0):  action=FinishedGlobalUpdateEffect,
guard=FinishedGlobalUpdateGuard]
| Trigger=(1.7.1)FinishedRequest Legs=[Leg (1.7.2):  action=SendConfirmation, guard=AnswerConfirmed, Leg
(1.7.3):  action=WriteLog, guard=WriteLog, Leg (1.7.4):  action=FinishedIntelligenceUpdateEffect, Leg
(1.7.5):  action=StartingStage3]
| Trigger=(1.9)TransactionOpened Legs=[Leg (1.9.0):  action=WriteData]
| Trigger=(1.9.1)RequestIn Legs=[Leg (1.9.2):  action=LogRequest, guard=FM]
| Trigger=(2.0)FinishedWriting Legs=[]
| Trigger=(2.0.1)Established Legs=[Leg (2.0.1.0):  action=SendData]
| Trigger=(2.0.2)FinishedSending Legs=[]
| Trigger=(2.0.3)Resume Legs=[Leg (2.0.4):  action=Stage3Completed]
| Trigger=(2.1)Exit Legs=[]
|
| State Transitions:
| top[], parent:  null;
| top.Initial1[Leg :    to top.Stage1], parent:  top
| top.Stage1[ Trigger=Exit Leg :    to top.Final2], parent:  top
| top.Stage1.Stage1Initial1[Leg :    to top.Stage1.LocalShipUpdate], parent:  top.Stage1
| top.Stage1.LocalShipUpdate[ Trigger=GetGlobalUpdate Leg :    to top.Stage1.GlobalUpdate, Trigger=Update
Leg :    to top.Fork1], parent:  top.Stage1
| top.Stage1.Stage1History[], parent:  top.Stage1
| top.Stage1.GlobalUpdate[ Trigger=GetLocalShipUpdate Leg :    to top.Stage1.LocalShipUpdate,
Trigger=Update Leg :    to top.Fork2], parent:  top.Stage1
| top.Final2[], parent:  top
| top.Stage2[], parent:  top
| top.Stage2.GetRadarData[ Trigger=RadarResponded Leg :    to top.Stage2.CollectData, Trigger=RadarTimeout
Leg :  action=InformUserRadarFailed to top.Stage2.EndRadarUpdate], parent:  top.Stage2
| top.Stage2.CollectData[ Trigger=RadarDataIn Leg :    to top.Stage2.EndRadarUpdate], parent:  top.Stage2
| top.Stage2.GetRadetData[ Trigger=RadetResponded Leg :    to top.Stage2.CollectRadetData,
Trigger=RadetTimeout Leg :  action=InformUserRadetFailed to top.Stage2.EndRadetUpdate], parent:  top.Stage2
| top.Stage2.CollectRadetData[ Trigger=RadetDataIn Leg :    to top.Stage2.EndRadetUpdate], parent:
top.Stage2
| top.Stage2.ContactIntelligence[ Trigger=FinishedRequest Leg :    to top.Stage2.Answered], parent:
top.Stage2
| top.Stage2.Answered[Leg :  action=SendConfirmation, guard=AnswerConfirmed to top.Stage2.WillingToAnswer, Leg
:  guard=NoAnswer to top.Stage2.EndIntUpdate], parent:  top.Stage2
| top.Stage2.Stage2Initial1[Leg :    to top.Stage2.GetRadarData], parent:  top.Stage2
```

```
| top.Stage2.Stage2Initial2[Leg :    to top.Stage2.GetRadetData], parent:  top.Stage2
| top.Stage2.Stage2Initial3[Leg :    to top.Stage2.ContactIntelligence], parent:  top.Stage2
| top.Stage2.WillingToAnswer[Leg :  guard=NoLogWrite to top.Stage2.EndIntUpdate, Leg :  action=WriteLog, guard=WriteLog
to top.Stage2.EndIntUpdate], parent:  top.Stage2
| top.Stage2.EndRadarUpdate[Leg :  action=FinishedRadarUpdateEffect, guard=FinishedRadarUpdateGuard to top.Join1,
Leg :  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard to top.Join2], parent:  top.Stage2
| top.Stage2.EndRadetUpdate[Leg :  action=FinishedRadetUpdateEffect, guard=FinishedRadetUpdateGuard to top.Join1,
Leg :  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard to top.Join2], parent:  top.Stage2
| top.Stage2.EndIntUpdate[Leg :  action=FinishedIntelligenceUpdateEffect to top.Join2], parent:  top.Stage2
| top.Fork1[Leg :    to top.Stage2.GetRadarData, Leg :    to top.Stage2.GetRadetData], parent:  top
| top.Fork2[Leg :    to top.Stage2.ContactIntelligence, Leg :    to top.Stage2.GetRadetData, Leg :    to top.Stage2.GetRadarData
parent:  top
| top.Join1[Leg :    to top.Stage3 entry=StartingStage3], parent:  top saturateable count = 2
| top.Stage3[exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History, exit=Stage3Completed Trigger=Exit
Leg :    to top.Final1], parent:  top Entry:  StartingStage3 Exit:  Stage3Completed
| top.Stage3.OpenTransaction[ Trigger=TransactionOpened Leg :  action=WriteData to top.Stage3.CloseTransaction],
parent:  top.Stage3
| top.Stage3.CloseTransaction[ Trigger=FinishedWriting Leg :    to top.Stage3.Stage3Final1], parent:  top.Stage3
| top.Stage3.EstablishSatelliteConnection[ Trigger=Established Leg :  action=SendData to top.Stage3.CloseConnection],
parent:  top.Stage3
| top.Stage3.CloseConnection[ Trigger=FinishedSending Leg :    to top.Stage3.Stage3Initial2], parent:  top.Stage3
| top.Stage3.Stage3Initial1[Leg :    to top.Stage3.OpenTransaction], parent:  top.Stage3
| top.Stage3.Stage3Initial2[Leg :    to top.Stage3.WaitForRequest], parent:  top.Stage3
| top.Stage3.Stage3Final1[], parent:  top.Stage3
| top.Stage3.Stage3Initial2[], parent:  top.Stage3
| top.Stage3.WaitForRequest[ Trigger=RequestIn Leg :  action=LogRequest, guard=FM to top.Stage3.EstablishFMConnection,
Trigger=RequestIn Leg :  action=LogRequest, guard=Satellite to top.Stage3.EstablishSatelliteConnection, Trigger=Terminate
Leg :    to top.Stage1.Stage1History], parent:  top.Stage3
| top.Stage3.EstablishFMConnection[ Trigger=Established Leg :  action=SendData to top.Stage3.CloseConnection],
parent:  top.Stage3
| top.Final1[], parent:  top
| top.Join2[Leg :    to top.Stage3 entry=StartingStage3], parent:  top saturateable count = 3
| ======== End of input data =========
|
| Running Trigger=(1.0)GetGlobalUpdate Legs=[] :
|        Running top with currentStates[top.Initial1] :
|        Running top.Stage1 with currentStates[top.Stage1.Stage1Initial1] :
|        Running top.Stage1.LocalShipUpdate (externally) :
|               Running Trigger=GetGlobalUpdate Leg :    to top.Stage1.GlobalUpdate.  ...  success :
|                                                     Searching for completion transition in top.Stage1.GlobalUpdate
...
|               Running Trigger=Update Leg :    to top.Fork1.  ...  failure
| Running Trigger=(1.1)Update Legs=[] :
|        Running top with currentStates[top.Stage1] :
|        Running top.Stage1 with currentStates[top.Stage1.GlobalUpdate] :
|        Running top.Stage1.GlobalUpdate (externally) :
|               Running Trigger=GetLocalShipUpdate Leg :    to top.Stage1.LocalShipUpdate.  ...  failure
|               Running Trigger=Update Leg :    to top.Fork2.  ...  success :
|                Trying Leg :    to top.Stage2.ContactIntelligence ...  success.
```

```
|                 Trying Leg :    to top.Stage2.GetRadetData ...  success.
|                 Trying Leg :    to top.Stage2.GetRadarData ...  success.
| Leaving a composite state top.Stage1
| Running Trigger=(1.2)RadarResponded Legs=[] :
|        Running top with currentStates[top.Stage2.ContactIntelligence, top.Stage2.GetRadetData, top.Stage2.GetRadarData]
:
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.GetRadetData (externally) :
|               Running Trigger=RadetResponded Leg :    to top.Stage2.CollectRadetData.  ...  failure
|               Running Trigger=RadetTimeout Leg :  action=InformUserRadetFailed to top.Stage2.EndRadetUpdate.
...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.GetRadarData (externally) :
|               Running Trigger=RadarResponded Leg :    to top.Stage2.CollectData.  ...  success :
|                                                   Searching for completion transition in top.Stage2.CollectData
...
|               Running Trigger=RadarTimeout Leg :  action=InformUserRadarFailed to top.Stage2.EndRadarUpdate.
...  failure
| Running Trigger=(1.3)RadarDataIn Legs=[Leg (1.4):  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard]
:
|        Running top with currentStates[top.Stage2.ContactIntelligence, top.Stage2.GetRadetData, top.Stage2.CollectData]
:
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.GetRadetData (externally) :
|               Running Trigger=RadetResponded Leg :    to top.Stage2.CollectRadetData.  ...  failure
|               Running Trigger=RadetTimeout Leg :  action=InformUserRadetFailed to top.Stage2.EndRadetUpdate.
...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.CollectData (externally) :
|               Running Trigger=RadarDataIn Leg :    to top.Stage2.EndRadarUpdate.  ...  success :
|                                                   Searching for completion transition in top.Stage2.EndRadarUpdate
...
|                 Trying Leg :  action=FinishedRadarUpdateEffect, guard=FinishedRadarUpdateGuard to top.Join1
...  failure.
|                 Trying Leg :  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard to top.Join2
...  success.
|                       ...  success - join count incremented top.Join2
| Running Trigger=(1.5)RadetResponded Legs=[] :
|        Running top with currentStates[top.Stage2.ContactIntelligence, top.Stage2.GetRadetData] :
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.GetRadetData (externally) :
|               Running Trigger=RadetResponded Leg :    to top.Stage2.CollectRadetData.  ...  success :
|                                                   Searching for completion transition in top.Stage2.CollectRadetData
```

```
...
|               Running Trigger=RadetTimeout Leg :  action=InformUserRadetFailed to top.Stage2.EndRadetUpdate.
...  failure
| Running Trigger=(1.6)RadetDataIn Legs=[Leg (1.7.0):  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard]
:
|        Running top with currentStates[top.Stage2.ContactIntelligence, top.Stage2.CollectRadetData] :
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  failure
|        Running top.Stage2 (externally) :
|        Running top.Stage2.CollectRadetData (externally) :
|               Running Trigger=RadetDataIn Leg :    to top.Stage2.EndRadetUpdate.  ...  success :
|                                                    Searching for completion transition in top.Stage2.EndRadetUpdate
...
|                Trying Leg :  action=FinishedRadetUpdateEffect, guard=FinishedRadetUpdateGuard to top.Join1
...  failure.
|                Trying Leg :  action=FinishedGlobalUpdateEffect, guard=FinishedGlobalUpdateGuard to top.Join2
...  success.
|                       ...  success - join count incremented top.Join2
| Running Trigger=(1.7.1)FinishedRequest Legs=[Leg (1.7.2):  action=SendConfirmation, guard=AnswerConfirmed,
Leg (1.7.3):  action=WriteLog, guard=WriteLog, Leg (1.7.4):  action=FinishedIntelligenceUpdateEffect, Leg (1.7.5):
action=StartingStage3] :
|        Running top with currentStates[top.Stage2.ContactIntelligence] :
|        Running top.Stage2.ContactIntelligence (externally) :
|               Running Trigger=FinishedRequest Leg :    to top.Stage2.Answered.  ...  success :
|                Trying Leg :  action=SendConfirmation, guard=AnswerConfirmed to top.Stage2.WillingToAnswer
...  success.
|                Trying Leg :  guard=NoAnswer to top.Stage2.EndIntUpdate ...  failure.
|                Trying Leg :  guard=NoLogWrite to top.Stage2.EndIntUpdate ...  failure.
|                Trying Leg :  action=WriteLog, guard=WriteLog to top.Stage2.EndIntUpdate ...  success.
|                                                    Searching for completion transition in top.Stage2.EndIntUpdate
...
|                Trying Leg :  action=FinishedIntelligenceUpdateEffect to top.Join2 ...  success.
|                        ...  join is saturated top.Join2
|                Trying Leg :    to top.Stage3 entry=StartingStage3 ...  success.
| Running Trigger=(1.9)TransactionOpened Legs=[Leg (1.9.0):  action=WriteData] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.Stage3Initial1, top.Stage3.Stage3Initial2] :
|        Running top.Stage3.OpenTransaction (externally) :
|               Running Trigger=TransactionOpened Leg :  action=WriteData to top.Stage3.CloseTransaction.  ...
success :
|                                                    Searching for completion transition in top.Stage3.CloseTransaction
...
|        Running top.Stage3.WaitForRequest (externally) :
|               Running Trigger=RequestIn Leg :  action=LogRequest, guard=FM to top.Stage3.EstablishFMConnection.
...  failure
|               Running Trigger=RequestIn Leg :  action=LogRequest, guard=Satellite to top.Stage3.EstablishSatelliteConnection.
...  failure
|               Running Trigger=Terminate Leg :    to top.Stage1.Stage1History.  ...  failure
|        Running top.Stage3 (externally) :
```

```
|               Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  failure
|               Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure
| Running Trigger=(1.9.1)RequestIn Legs=[Leg (1.9.2):  action=LogRequest, guard=FM] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.CloseTransaction, top.Stage3.WaitForRequest] :
|        Running top.Stage3.CloseTransaction (externally) :
|               Running Trigger=FinishedWriting Leg :    to top.Stage3.Stage3Final1.  ...  failure
|        Running top.Stage3 (externally) :
|               Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  failure
|               Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure
|        Running top.Stage3.WaitForRequest (externally) :
|               Running Trigger=RequestIn Leg :  action=LogRequest, guard=FM to top.Stage3.EstablishFMConnection.
...  success :
|                                                    Searching for completion transition in top.Stage3.EstablishFMConnection
...
|               Running Trigger=RequestIn Leg :  action=LogRequest, guard=Satellite to top.Stage3.EstablishSatelliteConnection.
Sequence diagram transition is shorter that required by the State diagram.
| ...  failure
|               Running Trigger=Terminate Leg :    to top.Stage1.Stage1History.  ...  failure
| Running Trigger=(2.0)FinishedWriting Legs=[] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.CloseTransaction, top.Stage3.EstablishFMConnection]
:

|        Running top.Stage3.CloseTransaction (externally) :
|               Running Trigger=FinishedWriting Leg :    to top.Stage3.Stage3Final1.  ...  success :
|                              ...  success - saturateable count of top.Stage3 incremented
|        Running top.Stage3.EstablishFMConnection (externally) :
|               Running Trigger=Established Leg :  action=SendData to top.Stage3.CloseConnection.  ...  failure
|        Running top.Stage3 (externally) :
|               Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  failure
|               Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure
| Running Trigger=(2.0.1)Established Legs=[Leg (2.0.1.0):  action=SendData] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.EstablishFMConnection] :
|        Running top.Stage3.EstablishFMConnection (externally) :
|               Running Trigger=Established Leg :  action=SendData to top.Stage3.CloseConnection.  ...  success
:
|                                                    Searching for completion transition in top.Stage3.CloseConnection
...
| Running Trigger=(2.0.2)FinishedSending Legs=[] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[top.Stage3.CloseConnection] :
|        Running top.Stage3.CloseConnection (externally) :
|               Running Trigger=FinishedSending Leg :    to top.Stage3.Stage3Initial2.  ...  success :
|                              ...  success - top.Stage3 is saturated.
| Running Trigger=(2.0.3)Resume Legs=[Leg (2.0.4):  action=Stage3Completed] :
|        Running top with currentStates[top.Stage3] :
|        Running top.Stage3 with currentStates[] :
|        Running top.Stage3 (externally) :
```

```
|                Running exit=Stage3Completed Trigger=Resume Leg :    to top.Stage1.Stage1History.  ...  success
:
|                 Trying Leg :    to top.Stage1.LocalShipUpdate ...  success.
|                Running exit=Stage3Completed Trigger=Exit Leg :    to top.Final1.  ...  failure
| Running Trigger=(2.1)Exit Legs=[] :
|        Running top with currentStates[top.Stage1.LocalShipUpdate] :
|        Running top.Stage1.LocalShipUpdate (externally) :
|                Running Trigger=GetGlobalUpdate Leg :    to top.Stage1.GlobalUpdate.  ...  failure
|                Running Trigger=Update Leg :    to top.Fork1.  ...  failure
|        Running top.Stage1 (externally) :
|                Running Trigger=Exit Leg :    to top.Final2.  ...  success :
|                               ...  success - top is saturated.
|
| Input traversal finished.
| Run finished correctly.
|
```

11 Appendix B: Definitions

In this section I define the terms that are used through the entire text. Some of the definitions are identical to the standard UML definitions. I also rephrased some UML definitions for easier understanding. Some definitions are non-UML ones.
To cover most UML features, I used two UML drafts specified in [OMGb] and [OMGa].

11.1 Basic Behavior definitions

Event: Either a signal emission (in real-time environments) or a function call. Defined in UML as "The specification of noteworthy occurrence that has a location in time and space".

Action: An outgoing event, i.e. an event that is triggered by the associated transition. I use the same term both in the Sequence diagram and in the State diagram. In the standard UML Sequence diagram it is named *action* as well, whilst in the standard UML State diagram it is called *effect*.

Trigger: A trigger is an event that initialtes an associated transition. I use the same term both in the Sequence diagram and in the State diagram. In the standard UML State diagram it is named *trigger* as well, whilst in the standard UML Sequence diagram it is called *stimulus*.

Guard: A condition that must be satisfied in order to enable the associated transition to be performed.

Leg: A basic component of a Sequence diagram or a State diagram, depicted in UML as an arrow with an optional guard and an optional action. In UML, a *Leg* is called a *simple transition* in a State diagram, whereas in a Sequence diagram it is called a *message*.

Transition: Consists of a trigger and a nonnegative number of legs.

Step: An execution of transition actions, when the trigger occurs and the guards are satisfied.

Run: The Sequence of steps that are executed from the initial step to the last one.

11.2 Sequence Diagram definitions.

Object: An instance that is instantiated from a class.

Sequence (diagram) transition: A transition in a Sequence diagram that consists of an incoming leg (message) and all the outgoing legs (messages), if any, triggered by it.

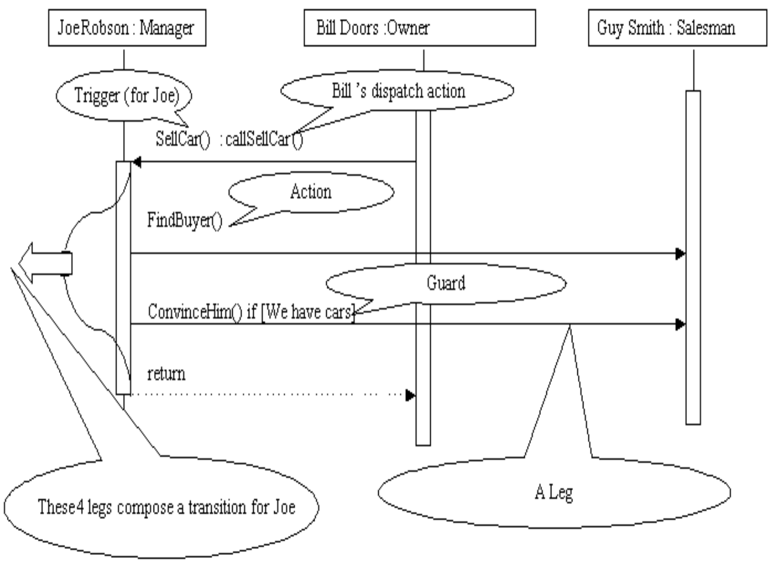

Figure 7: Sequence Diagram Definitions Example

11.3 State Diagram definitions.

State: A situation during the life of an object during which it satisfies some condition or waits for some event.

Pseudo state: An abstraction that encompasses different types of transient vertices in the same state machine graph.

Persistent state: A state where a thread of control stops till the end of one step.

Transient state: A state that is not a persistent state. This is equivalent to Pseudo

State.

Persistent-state leg: A leg that emanates from a persistent state. It always includes a trigger.

Transient-state leg: A leg that emanates from a transient state.

Substate: A state which is composed in some composite state.

Simple state: A persistent state that does not have substates.

Fork State: A transient state that serves to split the incoming leg into two or more legs terminating on orthogonal target vertices, which are substates in the same object state.

Join state: A transient state that serves to merge several legs emanating from source vertices in different orthogonal vertices.

Shallow history: A shorthand notation for a transient state, which represents the most recent active configuration of a composite state that directly contains this shallow history state.

Final State: A special kind of persistent state signifying that the enclosing composite state is completed. Defined as a transient state in UML.

Choice state: A state which, when reached, result in dynamic evaluation of guards on its outgoing legs.

Initial State: A transient state, which is a source for a single leg emanating towards the *default* substate of a composite state.

State (diagram) transition: A transition in a State diagram that includes the trigger and all the legs on the path between two persistent states, with no persistent states in between.

Composite State: A state that consists of concurrent (orthogonal) substates or of sequential (disjoint) substates.

Concurrent Composite State: A state that consists of concurrent (orthogonal) substates.

Exception: A leg from some state, which is located in a concurrent state, to some other state outside that concurrent state.

Saturateable state: A state that holds a count of the number of control flows that reached it. This count is called a *s*aturateable count. Fork, Composite and Concurrent composite states are saturateable states. The count is used later by the algorithm to know whether the execution should proceed beyond the saturateable state or not.

Current object state: The conjunction of all the active states in a State diagram, that defines the behavior of the object for any incoming event. At the beginning of the object lifetime, its *current object state* includes only all the initial State diagram states. The *current object state* is updated with each state transition. It can include multiple simple states, if the diagram contains concurrent composite or fork states.

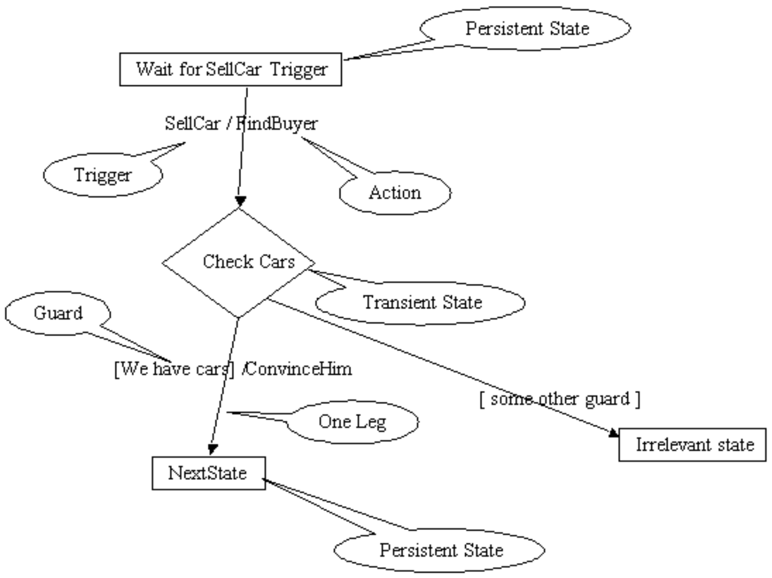

Figure 8: State Diagram Definitions Example

References

[DH01] W. Damm and D. Harel. Breathing life into message sequence charts. *Formal Methods in System Design*, 19:45–80, 2001.

[Hal93] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[hug] Hugo. http://www.pst.informatik.uni-muenchen.de/projekte/hugo/usage.html.

[iioa] Iioss. http://www.iioss.org/About/index-e.html.

[iiob] The IIOSS project. Integrated Inter-Exchangeable Object-Modeling and Simulation system. http://www.iioss.org/index-e.html.

[Ins] Using Prosa professional UML case envirinment together with Microsoft Visual Studio 6.0 and application wizard. http://www.insoft.fi/eng/pr2Proc.htm.

[Mar] R. Martin. UML tutorial. http://www.objectmentor.com/courses/resources/articles/umlsequencediagrams.pdf. Cf. Race conditions chapter.

[OMGa] OMG. Unified Modeling Language (uml) 1.3 specification. http://www.omg.org/cgi-bin/doc?formal/00-03-01.

[OMGb] OMG. Unified Modeling Language (uml) 1.4 specification. http://www.omg.org/cgi-bin/doc?formal/01-09-67.

[RVR+] A. Ramirez, P. Vanpeperstraete, A. Rueckert, K. Odutola, and J. Bennett. ArgoUML User Manual. Technical report.

[SW01] J. Schumann and J. Whittle. Automatic synthesis of UML designs from requirements in an iterative process. In *The Workshop for Precise Modeling and Deduction for Object-oriented Software Development (PMD '01)*, Siena, Italy, 2001.

[TS01] S. Merz T. Schafer, A Knapp. Model checking UML state machines and collaborations. *Proceedings Workshop software model checking*, 55(3), 2001.