An Explorative Journey from Architectural Tests Definition downto Code Tests Execution

Antonella Bertolino
Istituto di Elaborazione dell'Informazione
C.N.R. - Area di San Cataldo
56100 Pisa, Italy
bertolino@iei.pi.cnr.it

Paola Inverardi & Henry Muccini
Dipartimento di Matematica
Universitá dell'Aquila - Via Vetoio, 1
67100 L'Aquila, Italy
{inverard, muccini}@univaq.it

Abstract

Our research deals with the use of the Software Architecture (SA) as a reference model for the conformance testing of the implemented system with respect to its architectural specification, at the integration test level. Having formerly identified an approach to derive architectural test plans, we investigate here the practical meaning of a high level test case defined in terms of architectural processes and messages, such as the ones derived by our approach. Indeed, establishing a relation between SA tests (here formulated as paths derived over Labeled Transition Systems expressing the SA dynamics) and concrete, executable tests is not obvious at all. In this paper we describe the steps to be followed to refine architectural tests into code level tests, and we do so in an empirical context by illustrating our hands-on experience in running some of the derived architectural tests on the TRMCS case study. We present interesting insights and some preliminary attempts to generalize problems and solutions.

1. Introduction

Software Architecture (SA) has emerged as a powerful means for handling the design and analysis of complex distributed systems. The peculiarity of SA abstractions is to model a system in terms of components and connectors, where components represent a suitable abstraction of computational subsystems and connectors formalize the interactions among components [22]. The ability to handle system abstractions that meaningfully represent the overall structure and behavior of a system in terms of parts and their interactions is becoming even more significant with the growing use of COTS (Components Off The Shelf), that force a black box component integration driven system design.

For a few years there has been an increasing interest in studying the use of SA abstractions for improving the system validation phase. Various approaches focused on testing driven by the architectural specification [20, 21, 7, 5]. Some of these addressed the testing of the SA specification itself, others studied the use of architectural abstractions to guide the testing of the implemented system, in particular with respect to the integration test phase. In traditional systems, integration testing was driven by the hierarchical call structure between modules. But, in modern distributed (and very often Object Oriented) systems, approaches such as top-down or bottom-up integration are no longer applicable, as no hierarchy between components can be identified. New integration test strategies must be devised.

We have been studying for some time now the use of the SA formal description as a reference model for driving the conformance testing of an implemented system with respect to its specification at a high abstraction level. Indeed, it is our conviction that SA abstractions can be useful to derive significant integration test plans at the architectural level, i.e., at the level of subsystems integration. The SA specifically focuses on subsystems interactions, that is exactly what integration tests want to observe, ideally after each subsystem has been tested (or verified) in isolation. An alternative scenario is that the subsystems are COTS, and their mutual interactions when assembled together must be tested against an overall system specification. We call this testing phase, that is the testing of subsystems integration against an SA specification, the *architectural* testing: in our view, in a modern development paradigm architectural testing lies somewhere in between the testing/verification/acquisition of system components and overall system testing (e.g., for acceptance or reliability evaluation), and replaces, or possibly complements, traditional incremental test strategies.

Our approach is based on the assumption that the SA behavior is expressed as a Labeled Transition System (LTS), which completely describes *all* and *only* the expected interactions among components, i.e., the behavior the archi-

tectural model describes must be implemented at the source code level. Although SA represents a very high level system abstraction, those interactions can very soon become complex and unmanageable. Thus it is generally impossible to straightforwardly apply coverage strategies on the architectural LTS. Moreover, as discussed in the Conclusions, the SA abstraction might not make the application of existing LTS based test techniques convenient or even possible.

In [5] we introduced an approach to derive architectural test plans based on a notion of an *observation* of the SA dynamics, aimed at focusing the tester's attention on the relevant architectural interactions, while abstracting away from uninteresting ones. We are thus able to derive from the LTS several much smaller Abstract LTSs (ALTSs), each relative to a selected test view, or criterion. The idea is that a tester can use the ALTS to easily derive test sequences useful at testing specific behaviors.

But, what is in practice the meaning of a high level test case defined in terms of architectural processes and messages, such as the ones derived by our approach? And, which problems face a software tester who is given the task to execute such architectural tests? Indeed, the distance between SA tests (in our approach formulated as paths derived over the ALTS) and tests to be executed at the code level can be very large, and establishing a relation between them is not obvious at all. In this paper we investigate the steps to be followed to refine architectural tests into concrete tests, and we do so in the empirical context of the TRMCS case study. This on one side limits the generality of the approach we have devised to map architectural test into concrete ones, on the other offers a useful, concrete ground on which we can experiment ideas, identify problems and find solutions.

The paper is organized as follows. Section 2 recalls our approach to deriving architectural tests. Section 3 introduces the Teleservice and Remote Medical Care System (TRMCS) case study, its SA and the ADL we used to derive the LTS semantics, i.e., FSP [11, 15]. Section 4 is devoted to the description of the practical methodology to map the architectural test plans onto concrete tests. We do this in the context of the case study. Section 5 draws conclusions and discusses related work.

2. Our approach to SA testing

Several Architectural Description Languages (ADL) rely on Labeled Transition Systems (LTSs) to model the set of all possible SA behaviors [4, 1, 11], i.e., the dynamic behavior. In our approach we assume the existence of a SA description in some ADL and that from such description an LTS model can be derived, whose node and arc labels represent respectively states and transitions relevant in the context of the SA dynamics. Note that the model derivation allows aspects such as concurrency, non determinism, asynchrony, etc., relative to architectural non sequential behaviors, to be expressed in the LTS model.

To introduce our approach to architectural testing [5] we recall the definition of LTS.

Definition 2.1 A Labeled Transition System is a quintuple $(\mathcal{S}, \mathcal{L}, S_0, \mathcal{S_F}, \mathcal{T})$, where $\mathcal{S}$ is the set of states, $\mathcal{L}$ is the set of distinguished labels denoting the LTS alphabet, $S_0 \in \mathcal{S}$ is the initial state, $\mathcal{S_F} \subseteq \mathcal{S}$ is the set of final states and $\mathcal{T} = \{\stackrel{l}{\longrightarrow} \subseteq \mathcal{S} \times \mathcal{S} \,|\, l \in \mathcal{L}\}$ is the transition relation labeled with elements of $\mathcal{L}$.

A path p on an LTS, where $p = S_0 \stackrel{l_1}{\longrightarrow} S_1 \stackrel{l_2}{\longrightarrow} S_2 \stackrel{l_3}{\longrightarrow} \ldots \stackrel{l_n}{\longrightarrow} S_n$ is a *complete path* if S_0 is the initial state and S_n is a final state. In the following, for brevity, an LTS path will also be denoted by its sequence of actions.

As the starting point for our approach we assume that the software architect, by looking at the SA dynamics from different viewpoints, defines various *obs*-functions over the SA model, each one highlighting a specific perspective of interest for a test session (step1); with some abuse of terminology, we will also refer to each of the selected viewpoints, or *obs*-function, as to a *SA testing criterion*. Applying each *obs*-function to the LTS, an Abstract LTS (ALTS) can be generated, i.e., a reduced LTS showing only interesting behaviors with respect to the selected view (step2). On this much more manageable graph than the original LTS, the software architect chooses a set of important patterns of behaviors (paths over the ALTS) to be tested (step3); in this way we select only the execution scenarios we are interested to test. These high level tests are finally passed to the software tester, who has to actually run the tests and observe if the current implementation conforms to its architectural model (step4); we will say that *the implementation does not conform to the specification if some interactions described at the architectural level are not feasible in the implementation*. In summary, our approach to SA-based testing consists of four logical steps:

1. Definition of an *obs*-function relative to a selected *test criterion*;

2. Derivation from the SA LTS of the Abstract LTS (ALTS) corresponding to the defined *obs*-function;

3. Selection of a set of test classes over the derived ALTS;

4. Execution of the selected tests on the source code.

In the following subsections we will briefly discuss the first three steps only (see [5] for a detailed discussion) while the forth step will be described in Section 4.

2.1. Step 1: Obs-function definition

In general terms, an SA testing criterion consists of selecting the relevant behavioral patterns for testing purposes, abstracting away uninteresting interactions. Referring to the LTS definition, an SA testing criterion naturally partitions the LTS *actions* [1] into two groups: relevant interactions (i.e., those we want to observe by testing) and not relevant ones (i.e., those we are not interested in at this time).

We therefore associate with an SA testing criterion an obs-function. This function maps the relevant LTS labels to a specified interpretation domain $\mathcal{D}$, while any other (not relevant) one is mapped to a distinct element τ. More precisely, we have: $obs : \mathcal{L} \longrightarrow \mathcal{D} \cup \{\tau\}$.

The intuition underlying the set $\mathcal{D}$ is that it expresses a *semantic view* of the effect of the actions on the system global state. Given an LTS, the idea is to assign a semantic meaning to the alphabet items, capturing only interesting behaviors.

2.2. Step 2: ALTS derivation

We use the obs-function to derive from the LTS a reduced automaton called ALTS (for $Abstract$ LTS). The ALTS expresses all behaviors we want to test according to the selected SA testing criterion, but hides any other not relevant behavior. The ALTS is obtained by first relabelling, according to the obs-function, each arc label in the LTS, and then by minimizing the resulting automaton with respect to a selected equivalence, preserving desired system properties (as discussed in [4]).

2.3. Step 3: Test selection on the ALTS

Deriving an adequate set of tests, according to a selected SA testing criterion, amounts to deriving a set of complete paths appropriately covering the ALTS associated with the criterion via the obs-function. That is, we match a complete ALTS path to the high level specification of a test for the SA.

Which coverage criterion should be applied to the ALTS? Depending on the size of the graph, even some demanding criterion, e.g., taking all ALTS complete paths (after fixing a maximum number of iterations for possible cycles) would *apparently* lead to a practicable number of tests. However, this is not actually the case, because, as we will see in Section 4, each ALTS path can be refined into many concrete, executable test cases: this is why we speak here in terms of test classes. Therefore, less thorough coverage criteria seems more convenient.

[1] in the following, the terms LTS "actions" and LTS "labels" will be used interchangeably

3. Modeling the TRMCS case study behavior

The Teleservice and Remote Medical Care System (TRMCS) provides monitoring and assistance to users with specific needs, like disabled or elderly people. The TRMCS has been developed at Parco Scientifico e Tecnologico d'Abruzzo, and currently a Java prototype is running and undergoes SA based integration testing.

A typical TRMCS service is to send relevant information to a local phone-center so that the family and medical or technical assistance can be timely notified of critical circumstances. We define four different processes (User, Router, Server and Timer), where:

- User: sends either an "alarm" or a "check" message to the Router process. After sending an alarm, it waits for an acknowledgement from the Router.
- Router: waits for signals (check or alarm) from User. It forwards alarm messages to the Server and checks the state of the User through the control messages.
- Server: dispatches help requests.
- Timer: sends a clock signal for each time unit.

We have already presented the TRMCS case study in [5, 2]. However, in previous papers we used a Cham SA specification, while here we refer to a new SA specification, whose dynamics is expressed using the Finite State Process (FSP) Model [11, 15]. This specification language provides a concise way of describing LTSs; each FSP expression can be mapped onto a finite LTS and vice versa. The FSP specification is based on the definition of processes, whose behavior is modeled by LTSs; each process instance implements an architectural component; several processes can be combined (with a parallel composition operator) to describe the interaction between different processes. An FSP specification is composed by a declarative section, defining variables and ranges, a section defining the process initial state and a section to describe the other reachable states (see Figure 1). Semantically, an FSP process waits for an action (e.g., for receiving messages), performs actions (e.g., for sending messages) and eventually changes its state. The LTS alphabet is composed of the exchanged messages. The LTSA tool [14] can be used to generate and graphically (or textually) depict the process behavior starting from a FSP specification. Each FSP process can be described by a LTS model that is automatically built starting from the specification. The LTSA tool can also be used to evaluate process properties (i.e., safety, deadlock, reachability) or to put in parallel different processes. The latter feature allows to obtain a global LTS model of the system.

Figure 1 shows a portion of the FSP specification for the TRMCS system. In a) we define some ranges: they

```
range N = 0..1
range K = 0..1                                                                    a)
range Sent = 0..1
--------------------------------------------------------------------------------------
USER_ALARM= (sendAlarm_To_Router -> receiveAck_From_Router -> USER_ALARM).
USER_CHECK = (sendCheck_To_Router -> USER_CHECK).
||USER = (USER_ALARM||USER_CHECK).                                                b)
--------------------------------------------------------------------------------------
ROUTER_RECEIVEALARM = (receiveAlarm_From_User -> sendAlarm_To_Server ->
        receiveAck_From_Server -> sendAck_To_User -> ROUTER_RECEIVEALARM).
ROUTER_RECEIVECHECK = (receiveCheck_From_User -> (sendInput_To_Timer ->
        ROUTER_RECEIVECHECK|pre_receiveCheck -> ROUTER_RECEIVECHECK)).
ROUTER_RECEIVETIME = (receiveTime_From_Timer ->(sendNoFunc_To_Server ->           c)
        ROUTER_RECEIVETIME|pre_receiveTime-> ROUTER_RECEIVETIME)).
||ROUTER = ([0..1]:ROUTER_RECEIVEALARM||[0..1]:ROUTER_RECEIVECHECK||
ROUTER_RECEIVETIME).
--------------------------------------------------------------------------------------
...                                                                               d)
--------------------------------------------------------------------------------------
||ALL_PROCESSES=(u[0..1]:USER||r:ROUTER||sa[0..1]:SERVER||t:TIMER)/
{
u[0].sendAlarm_To_Router/r.[0].receiveAlarm_From_User,
u[1].sendAlarm_To_Router/r.[1].receiveAlarm_From_User,                            e)
...
```

Figure 1. TRMCS FSP Specification

D = {SendAlarm1, SendAlarm2, ReceiveAck1, ReceiveAck2}

obs (u.0.sendAlarm_To_Router)	**= SendAlarm1**	**:** ***User1 issues an Alarm msg***
obs (r.0.sendAck_To_User)	**= ReceiveAck1**	**:** ***User1 receives an Ack***
obs (u.1.sendAlarm_To_Router)	**= SendAlarm2**	**:** ***User2 issues an Alarm msg***
obs (r.1.sendAck_To_User)	**= ReceiveAck2**	**:** ***User2 receives an Ack***

For any other T_i, obs (T_i) = tau

Figure 2. AlarmObs testing criterion

are used to create multiple instances of processes; in b) the User process is decomposed into two parallel processes (USER_ALARM and USER_CHECK) able to send alarms and checks; in c) the Router is decomposed into three parallel processes (to receive Alarms and Checks and to wait for the Timer); part d) describes the Server and Timer processes; part e) is used to put in parallel the various processes.

Running the LTSA tool on the TRMCS FSP specification we obtained a (minimized) LTS composed of 256 states. It specifies the behavior of the ALL_PROCESSES composite process, given by the composition of all system processes. The subset of the LTS alphabet used in Figure 1 is given by $\mathcal{L}' \subseteq \mathcal{L} = \{$sendAlarm_To_Router, receiveAck_From_Router, send_Check_To_Router, receiveAlarm_From_User, sendAlarm_To_Server, receiveAck_From_Server, sendAck_To_User, receiveCheck_From_User, sendInput_To_Timer, pre_receiveCheck, receiveTime_From_Timer, sendNoFunc_To_Server, pre_receiveTime$\}$.

Let us now apply the first three steps of our approach to the TRMCS LTS.

Applying the first step

Considering the TRMCS description in Section 3, we have defined the following testing criterion: *"all those behaviors involving the flow of an Alarm message through the system"* should be tested. This criterion is formally expressed by an *obs*-function. In this case, $\mathcal{D}$ will contain all and only the actions (i.e., elements of the LTS alphabet) that specifically involve the sending of an Alarm message by a User, or the consequent User's reception of an acknowledgement from the Router.

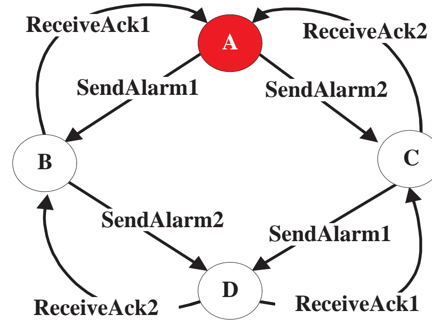

Figure 3. Alarm flow: ALTS

Figure 2 shows the obs-function defined for the given criterion and called AlarmObs: the action label "u.0.sendAlarm_To_Router" means that User1 sends an Alarm message to the Router, while "r.0.sendAck_To_User" means that Router1 sends back an Ack message to User1.

Applying the second step

Given the AlarmObs criterion (Fig. 2), and by applying reduction and minimization algorithms, we have derived the ALTS depicted in Figure 3 (the shaded circle represents the initial state, that in this example also coincides with the only final one). This ALTS represents in a concise, graphical way how the Alarm flow is handled: after an Alarm is issued (e.g., SendAlarm1), the system can nondeterministically react with one of two possible actions: elaborating this Alarm and sending back an Acknowledgement (ReceiveAck1) or receiving another Alarm message from another User (SendAlarm2).

Applying the third step

For the TRMCS case study, at the current stage we have manually extracted from each generated ALTS some interesting paths, without considering any specific criterion. However, while deriving the paths, we tried to consider different combinations of actions, to make the tests more interesting as far as architectural behaviors are concerned. These paths are discussed in the next section where we will finally see what is implied by each of the derived test classes.

4. Code level testing

In this section we will try to understand how a tester can use the architectural paths, as produced in Section 3, to actually test whether the source code conforms to the architectural description. As said, an ALTS path can correspond to many concrete tests. Since they are based on an *obs*-

function applied over the LTS, ALTS paths clearly specify architectural behaviors at a higher level than the original LTS. Thus, a test class derived from the ALTS can also correspond to several architectural tests, or, more precisely, one ALTS complete path can correspond to many possible LTS paths. Let $p = A \xrightarrow{d_1} B \xrightarrow{d_2} C$ be a selected ALTS path, where labels $d_1, d_2 \in \mathcal{D}$, i.e., the used *obs*-function includes the transformations $obs(act_i) = d_1$ and $obs(act_j) = d_2$, with act_i and act_j belonging to the LTS alphabet $\mathcal{L}$. Then every LTS complete path of the following form: $p = S_0 \ldots \xrightarrow{act_i} \ldots \xrightarrow{act_j} \ldots S_n$ conforms to the selected ALTS path, provided that any other action implied by the dots only corresponds to τ mapped actions.

So, what do we take as an "architectural test"? A first option is to consider an ALTS complete path as our test specification. In this case, the test is specified at a more abstract level, and the tester is naturally led to focus the testing on a restricted set of interactions (those in the ALTS alphabet). A second option is to go back to the complete LTS and derive the LTS paths of which the selected ALTS path is an abstraction[2], or to select from all of them a subset of LTS paths. As LTS paths are more detailed than ALTS ones, in this case the tester has more information about how to perform the tests, but also stricter requirements, i.e., (s)he has reduced freedom to choose the code level tests: in practice, it could be more difficult to test the conformance of source code to the test specification.

In either cases (ALTS or LTS path), an architectural test is essentially a *sequence of system actions that are meaningful at the SA level*. What can be different in the two cases is (as always) the level of abstraction at which the sequence is described. This does not sensibly affect the discussion on how to derive code level tests corresponding to a specified architectural test sequence. Henceforth we will generally speak in terms of SA paths, assuming that the tester is given an SA (LTS or ALTS derived) path to be tested, and that (s)he can access the source code and relative documentation. We also assume the source code to be object-oriented since our case study has been implemented in Java.

The approach we followed to realize step 4 (source code testing) can be conceptually decomposed into four substeps:

4.1: the SA path to be tested can be represented as an ordered sequence of events, like "act1.act2.act3.act1" in Figure 4.a). Making the assumption that each event relates to a unique pair of communicating components, the sequence can be represented as a set of SA components interactions. We use UML [19] Stereotyped Sequence Diagrams [13] to represent these scenarios: each box represents an SA component, while arrows represent actions performed by the components. Figure 4.a) represents this initial step;

[2] In [6] an algorithm is presented on how to derive the LTS paths corresponding to a selected ALTS path, using some graph manipulation techniques.

4.2: for each action in the SA path, the tester identifies the code level sequence diagram(s) (specified as a UML Sequence Diagram) implementing its behavior. Each sequence diagram represents a partially ordered sequence of events and specifies how one action of the SA path (e.g., the Alarm action) is implemented in the code (e.g., how the Alarm is implemented in the source code) (Fig. 4.b). A sequence such as $\mathrm{obj}_1 \xrightarrow{m_1} \mathrm{obj}_2 \xrightarrow{m_6} \ldots \xrightarrow{m_1} \mathrm{obj}_N$ represents the (code level) sequence diagram in which obj_i represents an object, m_j represents a method and $\mathrm{obj}_1 \xrightarrow{m_1} \mathrm{obj}_2$ says that object obj_1 calls its method m_1 that passes the control to obj_2. We will use the explicit notation $\mathrm{obj}_1 \xrightarrow{obj_2.m_1} \mathrm{obj}_2$ to denote that m_1 is a method of obj_2 (called by obj_1).

4.3: given the SA path to be tested, the tester "suitably" combines the code level sequence diagrams corresponding to each label. The global sequence diagram so obtained should represent a source code scenario implementing the SA path (see Figure 4.c). We will discuss in the following what suitably means;

4.4: the tester runs the code to check whether the hypothetical sequence diagram generated in step 4.3 is correctly implemented by the system. Since the system runs several concurrent processes, in general it is very difficult to trace system behavior. In fact, multiple executions of a concurrent program with the same input may exercise different sequences of interactions and may produce different results. Our pragmatic approach here is to apply a nondeterministic testing approach for some attempts; if the expected sequence of events is not observed, then a deterministic approach [9] could be applied to force the desired behavior.

Implementing each of the above substeps in practice involves several difficult problems. Notably we have:

Problems P4.2 (relative to step 4.2):

a) it could not be so obvious to understand which classes and methods implement an architectural functionality (this also depends on the followed development paradigm);

b) more than one sequence diagram can implement the desired behavior. In some cases, it might be necessary to identify a specific sequence;

Problems P4.3 (relative to step 4.3):

a) in the SA sequence, *we only see the interactions* between specified SA components and treat these components as black box entities. But, the SA model is a high-level abstraction of the system, behind which

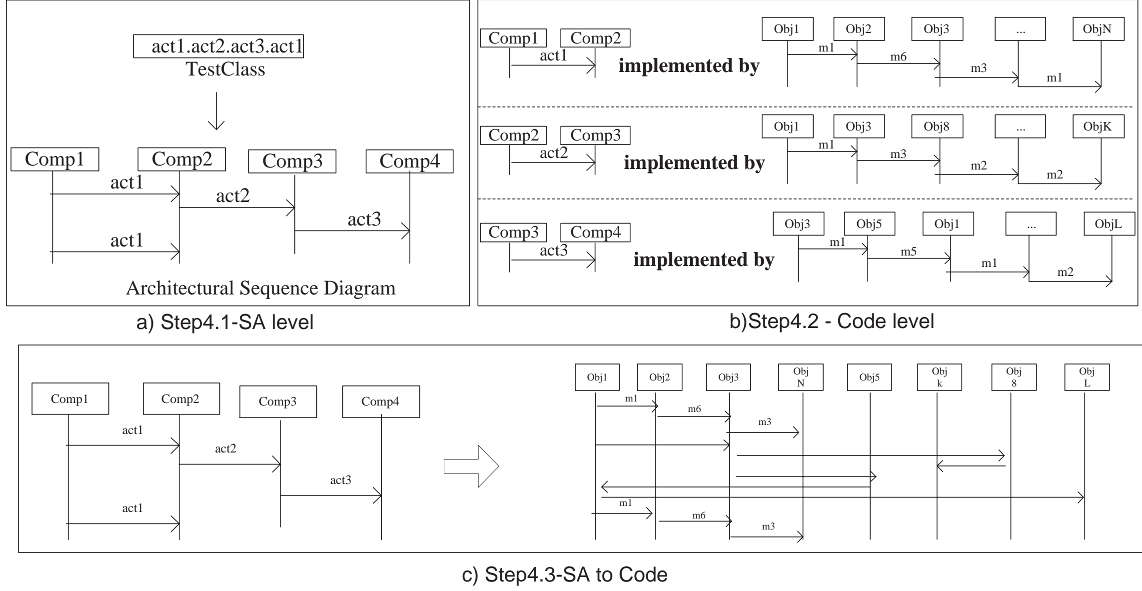

Figure 4. Substeps for step 4 - source code test

several subcomponents (and their connectors) can remain hidden. When deriving the code level sequence diagrams, we can end up with having to consider a lot of functionalities that are not described in the architectural sequence diagram: for instance, in the SA description we do not consider computations, but only interactions;

b) the system can be *concurrent*, and the well-known problem of interleaving between code level sequences arises. If the SA path is given by $L_1.L_2.L_3...$, where L_1 is implemented by $obj_1 \xrightarrow{m_1} obj_2 \xrightarrow{m_3} ... \xrightarrow{m_i} obj_k$, L_2 by $obj_a \xrightarrow{m_b} obj_d \xrightarrow{m_d} ... \xrightarrow{m_x} obj_h$ and L_3 by $obj_x \xrightarrow{m_y} ... \xrightarrow{m_i} obj_z$, how should the different calls be synchronized to obtain a feasible test?

c) the architectural model usually describes *only the expected behaviors*, while the implementation has to handle also the exceptional behaviors.

To help the tester managing the first class of problems (P4.2.a), we create a correspondence between the Inputs (and Outputs) we can observe in the source code and the LTS actions. Therefore, we classify the LTS alphabet in Input, Output and Internal Labels, with the following semantics: if the component C1 receives a message labeled by L $\in$ Input, (it means that) C1 receives an input from the system, computes the value and issues a message to some other components; when L is an Output label, the component produces an Output; if the action is labeled as an Internal one, the system performs internal operations (e.g., forwarding messages between components). In this way, if L has been classified as an Input label, then we expect it to be mapped to one sequence in which the system receives an input.

Problem P4.2.b can only be handled by requiring the tester's intervention, who manually evaluates every point of decision and performs a reasoned choice of which selection is more relevant for testing purposes. What is important here is to save memory of the choices done and of the related motivations (test pattern), for later analysis.

When deriving the combined sequence diagram (suitably combined), we have to potentially consider any computation the code could execute (P4.3.a). P4.3.b can be handled by applying this consideration: the actions L_i we use in our SA description only state that some messages are exchanged between components, while each code level sequence diagram contains all the classes and operations needed to actually implement the message exchange. In this context it is possible to focus on the code level operation(s) (m_{synch}) directly implementing the sending or the receipt of a message. If the sequence $L_1.L_2$ must be covered and $L_1 = c_1 \xrightarrow{m_1} c_2 \xrightarrow{m_3} ... \xrightarrow{m_i} c_k \xrightarrow{m_{synchL1}}$, $L_2 = c_a \xrightarrow{m_b} c_d \xrightarrow{m_d} ... \xrightarrow{m_i} c_h \xrightarrow{m_{synchL2}}$, then the only real constraint imposed by the SA path above is that $c_k \xrightarrow{m_{synchL1}}$ must occur before $c_h \xrightarrow{m_{synchL2}}$. All the other calls can be interleaved in any way. Problem P4.3.c can be handled selecting the code level scenario that implements the expected behavior.

Applying the fourth step to the TRMCS

The SA test we considered was: *does the system correctly handle the sending of an Alarm message by two different users?* We hence tried to test the system with respect to the SA test sequence

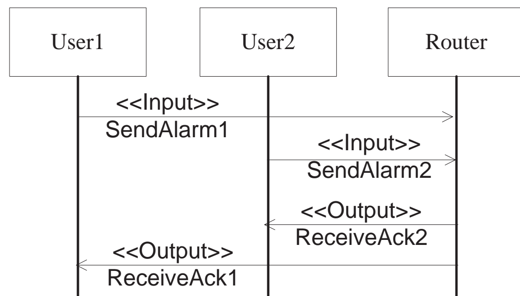

Figure 5. Step4.1: Architectural Sequence Diagram

SendAlarm1.SendAlarm2.ReceiveAck2.ReceiveAck1 (ALTS path A.B.D.B.A in Fig 3).

Accordingly to the approach previously described, we performed the following substeps:

4.1) the test sequence *SendAlarm1.SendAlarm2.Receive Ack.ReceiveAck1* can be represented by the architectural sequence diagram in Figure 5; both User1 and User2 send an alarm message to the Router and User2 is assumed to receive the ack before User1. We used the <<Input>> stereotype to identify the stimulus to which the system components react, while the <<Output>> stereotype identifies the results produced by a component and transmitted to other system components.

4.2) SendAlarm1 and SendAlarm2 (as well as ReceiveAck1 and ReceiveAck2) are implemented by the same classes and the same methods. In particular, the Alarm function is implemented by two different Java processes: the User writes the Alarm message into a socket and the Router periodically reads the socket looking for new messages. The Ack function is implemented following a similar scheme: the Router writes into a socket and the User waits, but this message is written on a different socket port. The Alarm function is clearly an Input function (e.g., the User pushes an Alarm button), while the Ack is an Output. The objects and the methods implementing the Alarm function are shown in Figure 6. Let us very briefly comment it: from the User side (Fig. 6.a1), the xUser object first checks if every required parameter has been introduced (if_1). If this is the case, the xUser performs some internal operations and creates the User object. After some time (in which internal operations can be performed), if the User pushes the Alarm button (if_2) then the User creates a ClientConnection and the SendNMessage (SNM) objects. The User calls the send() method of the SNM object that tries "n" times to create the socket. If the socket is created (if_3) the SNM finally writes the Alarm message into the socket. From the Router side (Fig. 6.a2), the Router graphical interface object (xRouter) creates a new instance of the MasterRouter object, that in turn creates the ServerConnection object. It tries to create a ServerSocket: if the ServerSocket ss is created (if_a) then a process is activated that checks the ServerSocket port forever. In this way, when a new message is written in the socket (if_b) the socket is read and the Alarm is received.

4.3) in this step, the architectural sequence diagram depicted in Figure 5 must be implemented in terms of code level sequence diagrams. As explained in step 4.3, the tester must combine the various sequence diagrams corresponding to each SA label into one global diagram. In particular, the TRMCS implementation concurrently manages several processes, so there does not exist a unique flow of communication. For each User sending an alarm, an instance of the Alarm flow (as shown in Figure 6.a1 or as summarized in Fig. 7.a1) concurrently runs in the system; many instances of the Router ReceiveAlarm process are running, one for each User (Fig. 7.a2). When the Router has received the Alarm, it generates a new process (Fig. 7.b1) trying to send the ack msg to the User while the User is waiting (Fig. 7.b2). We handled the combination of the sequence diagrams by considering the synchronization points (m_{synch}) as described in P.4.3.b. Looking at Figure 7, the *write()* and *read()* operations called by SendNMessages and ReceiveNMessages objects represent the synchronization points; "Process a" happens before "Process b" if the read() operation in a2 happens before the read() operation in b2.

4.4) Is the SA path *SendAlarm1.SendAlarm2.Receive Ack2. ReceiveAck1* correctly implemented in the TRMCS? At the code level sequence diagrams, it is translated into: can we execute the sequence (Process a)(x=1).(Process a)(x=2).(Process b)(x=2).(Process b)(x=1)? We repeatedly run the system with Alarm1 and Alarm2 as Inputs, and observed the following unexpected behaviour: User2 never received the Acknowledgment. Instead, User1 (the first User sending an Alarm) received both the ack to its own Alarm message and the ack issued for User2 Alarm. Analyzing the source code we could understood the problem origin, i.e., the Router always wrote the ack message to the same port. Note that this malfunctioning clearly affected the SA behavior and could not have been observed by testing the Router in isolation.

Some Considerations:

P4.2.b: From the SA description, we get the basic requirement that if User1 sends an Alarm msg, (s)he must receive an Ack. In the system implementation there are several conditions and constrains to be verified to check whether this behavior is enabled: the User must exist (if_1 condition in Figure 6), the User must push the Alarm button (if_2 condition), the socket is successfully created (if_3 clause), the ServerSocket is successfully created (if_a) and the message is received (if_b); only the Sequence Diagram(s) validating these constrains are those that correctly

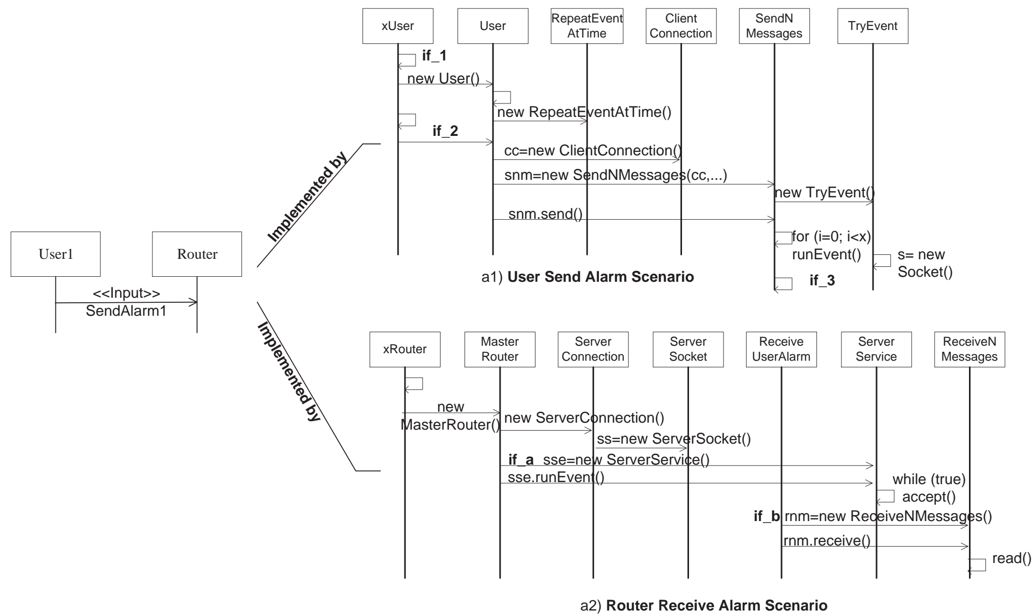

Figure 6. Step4.2: Send and Receive Alarm Sequence Diagram

map the architectural path Alarm1.Ack1.

P4.3.a: The architectural description is an abstraction of the real implementation; as a consequence, the Alarm, Check, Ack and Clock messages we described are only a simplified view of the system implementation and the User, Router, Server and Timer components could be decomposed in several other components. With the code level sequence diagrams we have also to consider internal system operations that are not described in the SA sequence diagram. Considering the Alarm sequence described in Figure 6, for instance, the system checks if the User exists, creates a Socket and writes into the Socket, creates a ServerSocket and reads from the Socket. Many more functionalities are implemented in the source code: there exist local and global databases; each interaction is stored in a log file; there are actually many Routers, each one serving some Users, etc. In our experiment we considered only the functionalities needed to implement the SA actions, not including the others. Another approach could be to refine our SA description to include all the implemented functions.

5. Conclusions

Our research investigates how the SA notions, methods and tools can be usefully exploited within a software development process. In particular, this paper focuses on using SA descriptions to guide the conformance testing of a large distributed system. As such, this paper stands on two prolific branches of the current Software Engineering literature: SA and specification-based testing.

For the former, the literature is rich of models and tools for the design and analysis based on the SA (a little less for SA-based testing, so far). Despite of such a big emphasis, we believe that the practical implications of using these models and tools, and in particular their relationship with source code level descriptions, are not well understood yet. In our view, the only means to investigate such tough problems is through empirical, hands-on, investigation, and this is what we have reported here.

As far as specification-based testing is concerned, although the methods and tools used basically remain the same, there are at least two main differences between what we try to do, and the existing approaches:

i) the SA description only tries to capture SA relevant behaviors, while abstracting away other important system functions, that are not related to component interactions. For this reason our tests specifically belong to the integration testing stages, and certainly do not strive for testing the system as completely as possible, like in traditional specification-based test approaches;

ii) the abstraction level of the reference model, and its relative ”distance” to the implementation under test, varies much in the two contexts: in SA-based approaches this distance is *purposely* very high, while in existing approaches this has hopefully to be low. In other words, an high abstraction level is instrumental to our approach, while in traditional approaches a close distance between the reference model and the implementation is desirable.

We have provided here an excerpt from the empirical insights gained while performing SA-based testing on the

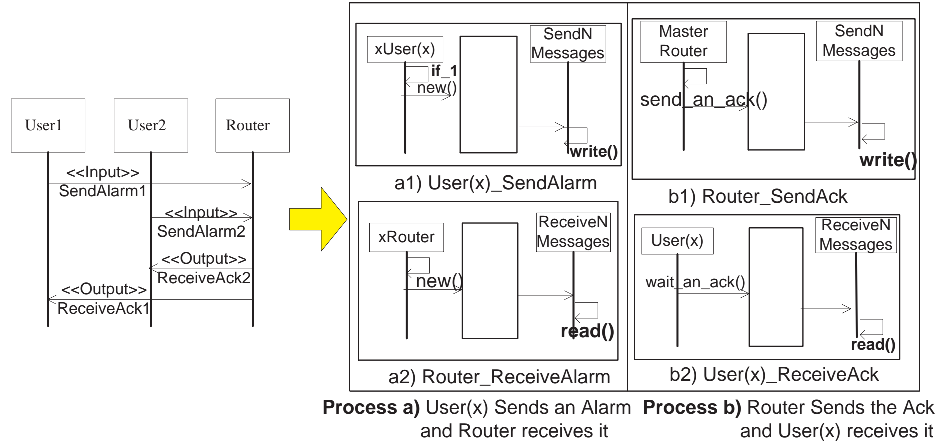

Figure 7. Step4.3: implementing the SA level Alarm Sequence Diagram

TRMCS case study. Although the experience report appears specific to the analyzed case study, we believe that in most cases the problems encountered as well as the solutions taken are easily generalizable to any SA centered context. In the following subsection we make a preliminary attempt to generalize the approach.

What we hypothesize in our method, in summary, is a stage in which an SA description, and in particular the SA dynamics, is analysed to derive architecturally relevant test sequences (this stage was studied in [5]), and then a subsequent stage in which those architectural tests are executed. In this paper we have discussed a practical approach for the second stage, by empirically taking an exploratory journey from SA tests downto code level tests: along this journey, we could show how the SA test execution implies several intriguing problems. Most notably, we could see how the tester's expertise is required in several decision points.

Indeed, we do not trust a complete automated architectural test setting: our emphasis is rather in making this necessary and expensive activity more formal and systematic than it is now, using automated support for LTS analysis, for ALTS derivation and ALTS path generation, while still relying on the invaluable judgement of the tester for critical decisions, like ALTS paths selection. Our future work consists in generalizing the lessons learnt into a methodical approach, while continuing empirical investigation.

5.1. Approach generalization and evaluation

How general is our approach?

i) different ADLs can be used to describe the SA; our only constraint is that the SA dynamics must be described via an LTS conforming to the definition given in Section 2;

ii) different ALTS coverage criteria can be applied: for example in [5] we used McCabe's [17] path coverage criteria to reduce the number of selected ALTS path.

iii) different implementation languages and programming paradigms can be used. In our approach we tested an object oriented implementation of the system, but we think that standard imperative programs can be tested in the same way.

iv) different software development processes can be used; the SA description can be obtained a-posteriori in a Reverse Engineering step; or, the SA description can be created *before* the system design thus driving the definition of the components and of their interactions; or, the SA description is directly mapped on the implementation [18]. This case study followed a Reverse Engineering process, but in future investigations we will analyze the differences with respect to different development processes.

How easy is it to use the approach? Based on our experience so far, we can say that the difficulty of our approach increases with: i) the difficulty to assign a semantic to the LTS labels (e.g., to create the observation): this is related to the knowledge on the system under analysis; ii) the difficulty to map SA items to the source code level; iii) the degree of concurrency and nondeterminism of the system.

How many test cases can we generate by this approach? The number depends on: i) the complexity of our observation (*obs*-function): the more labels are mapped in $\mathcal{D}$, the more complex is the obtained ALTS; ii) the ALTS coverage criteria; iii) the potential LTS paths derived backwards from the ALTS path; iv) how many sequence diagrams can implement the same architectural sequence of events.

5.2. Related work

In [5] we notice that although lot of work has been devoted to testing concurrent and real-time systems, both specification driven and implementation based [9, 16, 8], they all focus on unit testing while we want to derive test plans for integration testing.

Our approach of defining ALTS paths for specifying high level test classes has lot in common with Carver and Tai's use of *Synchronization Sequence (SYN-sequence) Constraint*, for specification-based testing of concurrent programs [9]. Again, the main difference is in the distance of the abstraction level of the reference model to the implementation. This prevents the straightforward application of their development. For example, the technique defined in [9] has been implemented in [3] for Java programs and the SYN-sequences are runtime program executions; in our approach the SYN-sequences are the test classes and thus they instead are more abstract.

Several authors [23, 10, 12] have recently dealt with specification-based testing from LTSs, and more specifically with the problem of automatically generating test suites for testing the conformance of an implemented system to its LTS specification. We also use the SA derived LTS as a reference model to derive test cases but in a different manner: they make the hypothesis that it is possible to have the model of the implementation behavior expressed by an LTS (or an Input/Output LTS); using some implementation relations (conf, ioconf, ioco, etc.) they test if the implementation is correct with respect to the specification. Again in our approach we do not assume to have the LTS of the implementation.

As far as architectural testing is concerned, the topic has raised interest and received a good deal of attention in recent years [21, 7]. Our approach indeed stems from this ground.

Acknowledgments

The authors would like to acknowledge the Italian M.U.R.S.T. national project SALADIN that partly supported this work and the anonymous reviewers for their constructive remarks.

References

[1] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Trans. on Software Engineering and Methodology*, Vol. 6, N. 3, pp. 213-249, July 1997.

[2] S. Balsamo, P. Inverardi, C. Mangano, and F. Russo. Performance Evaluation of a Software Architecture: A Case Study. *In IEEE Proc. IWSSD-9*, pp. 116-125, Japan, 1998.

[3] A. Bechini and K.-C. Tai. Design of a Toolset for Dynamic Analysis of Concurrent Java Programs. *In the 6th IEEE IWPC*, pp. 190-197, Ischia (Italy), 1998.

[4] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Architectural Abstractions to Support Analysis and Testing. *IR 50/99, University of L'Aquila*, On-line at: <http://www.dm.univaq.it/∼muccini/Page2.html>.

[5] A. Bertolino, F. Corradini, P. Inverardi, and H. Muccini. Deriving Test Plans from Architectural Descriptions. *In ACM Int. Conf. on Software Engineering (ICSE2000)*, pp. 220-229, Limerick (Ireland), June 2000.

[6] A. Bertolino, P. Inverardi, and H. Muccini. Specification-based Testing In-The-Large Driven by the Software Architecture. *IR 22/00, University of L'Aquila*, On-line at: <http://www.dm.univaq.it/∼muccini/Page2.html>.

[7] A. Bertolino, P. Inverardi, H. Muccini, and A. Rosetti. An Approach to Integration Testing Based on Architectural Descriptions. *In IEEE Proc. Intern. Conference on Engineering of Complex Computer Systems (ICECCS-97)*, pp. 77-84, Como (Italy), 1997.

[8] R. Cardell-Oliver and T. Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. *In Proc. FTRTFT'98*, LNCS Vol. 1486, pp. 251-261, 1998.

[9] R. H. Carver and K.-C. Tai. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Trans. on Software Engineering*, Vol. 24, N.6, pp. 471-490, June 1998.

[10] J.-C. Fernandez, C. Jard, T. Jeron, L. Nedelka, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. *In Proc. Computer Aided Verification (CAV'96)*, pp. 348-359, New Brunswick (USA), 1996.

[11] FSP. Finite State Process. On-line at: <http://www-dse.doc.ic.ac.uk/∼jnm/book/ltsa/Appendix-A.html>.

[12] S. Fujiwara and G. Bochmann. Test Selection Based on Finite State Models. *IEEE Trans. on Software Engineering*, Vol. 17, N. 6, pp. 591-603, June 1991.

[13] C. Hofmeister, R. L. Nord, and D. Soni. *Applied Software Architecture*. Addison Wesley, 1999.

[14] LTSA. Labelled Transition System Analyzer. On-line at: <http://www-dse.doc.ic.ac.uk/∼jnm/book/>.

[15] J. Magee and J. Kramer. *Concurrency: State models & java programs*. Wiley publisher, April 1999.

[16] D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Trans. on Computer Systems*, Vol. 13, N. 4, pp. 365-398, 1995.

[17] T. J. McCabe. A Complexity Measure. *IEEE Trans. on Software Engineering*, Vol. 2, N. 4, pp. 308-320, 1976.

[18] N. Medvidovic, D. S. Rosenblum, and R. N. Taylor. A language and environment for architecture-based software development and evaluation. *In Proc. IEEE Int. Conf. on Software Engineering (ICSE'99)*, pp. 44-53, Los Angeles, 1999.

[19] Rational-Corporation. Uml Resource Center. UML documentation, version 1.3. On-line at: <http://www.rational.com/uml/index.jtmpl>.

[20] D. J. Richardson, P. Inverardi, and A. Bertolino. *ACM Proc. Int. Workshop on the role of Software Architecture in testing and Analysis (ROSATEA)*, CNR-NSF, July 1998. On-line at: <http://www.ics.uci.edu/∼djr/rosatea/>.

[21] D. J. Richardson and A. L. Wolf. Software testing at the architectural level. *ISAW-2 in Joint Proc. of the ACM SIGSOFT '96 Workshops*, pp. 68-71, 1996.

[22] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[23] J. Tretmans. Conformance Testing with Labeled Transition Systems: Implementation Relations and Test Generation. *Computer Networks and ISDN Systems*, Vol. 29, pp. 49-79, 1996.