Using Software Architecture for Code Testing

Henry Muccini[1], Antonia Bertolino[2] and Paola Inverardi[1]

[1] Dipartimento di Informatica, Universitá dell'Aquila, Via Vetoio 1, 67100 L'Aquila,

{muccini,inverard}@di.univaq.it

[2] Istituto di Scienza e Tecnologie dell'Informazione "A. Faedo" (ISTI-CNR), Area della Ricerca CNR di

Pisa, 56100 Pisa, Italy, antonia.bertolino@isti.cnr.it

Abstract

Our research deals with the use of Software Architecture (SA) as a reference model for testing the conformance of an implemented system with respect to its architectural specification. We exploit the specification of SA dynamics to identify useful schemes of interactions between system components and to select test classes corresponding to relevant architectural behaviors. The SA dynamics is modeled by Labeled Transition Systems (LTSs). The approach consists of deriving suitable LTS abstractions called ALTSs. ALTSs offer specific views of SA dynamics by concentrating on relevant features and abstracting away from uninteresting ones. Intuitively, deriving an adequate set of test classes entails deriving a set of paths that appropriately cover the ALTS. Next, a relation between these abstract SA tests and more concrete, executable tests needs to be established, so that the architectural tests derived can be refined into code-level tests. In the paper, we use the TRMCS case study to illustrate our hands-on experience. We discuss the insights gained and highlight some issues, problems, and solutions of general interest in architecture-based testing.

Keywords

D.2 Software Engineering/D.2.11 Software Architectures

D.2 Software Engineering/D.2.5 Testing and Debugging

D.2 Software Engineering/D.2.5 Testing and Debugging/D.2.5.k Testing strategies

D.2 Software Engineering/D.2.5 Testing and Debugging/D.2.5.s Tracing

I. Introduction

In recent years, the study of software architecture (SA) has emerged as an autonomous discipline requiring its own concepts, formalisms, methods, and tools [30], [3], [24]. SA represents a very promising approach since it handles the design and analysis of complex distributed systems and tackles the problem of scaling up in software engineering. Through suitable abstractions, it provides the means to make large applications manageable. SAs support the formal modeling of a system, allowing for both a topological (static) description and a behavioral (dynamic) one. SA models the system in terms of components and connectors, where components represent abstract computational subsystems, and connectors formalize the interactions among components. Both industry and academia have actively used SA to improve the dependability of complex systems [3], [24], and SA-based processes have been proposed to model and analyze real systems [3], [30].

A crucial part of the development process is *testing*: software testing consists of the dynamic verification of a program's behavior, performed by observing the execution of the program on a selected set of test cases [6]. In particular, specification-based testing checks that the Implementation Under Test (IUT) fulfills the specifications, used to "capture" all and only the important properties against which the IUT has to be tested.

The importance of the use of formal methods in software specification and design does not need to be stressed. Several authors have highlighted the advantages of formal methods in testing as well, and several techniques have been proposed to select tests from semi-

formal [46], algebraic [5] and model-based [19] specifications.

Recently, various approaches have been proposed on testing driven by the architectural specification [28]. Among these, we have been addressing the problem of identifying suitable functional test classes for the testing in-the-large of complex real-world systems [8], [9]. This paper builds on our previous work [8], [9], presenting the fully developed approach in a coherent, comprehensive way.

Our goal is to provide a test manager with a systematic method to extract suitable test classes for the higher levels of testing and to refine them into concrete tests at the code level. Our approach is based on the specification of SA dynamics, which is used to identify useful schemes of interactions between system components, and to select test classes corresponding to relevant architectural behaviors. Our SA dynamics is modeled by Labeled Transition Systems (LTSs). The approach consists of deriving suitable LTS abstractions, called Abstract LTSs or ALTSs. ALTSs offer specific views of the SA dynamics by concentrating on relevant features and abstracting away from uninteresting ones. Intuitively, deriving an adequate set of test classes entails deriving a set of paths that appropriately cover the ALTS. Then the architectural tests must be refined into code-level tests in order to be executed. To this end, we follow here a stepwise, manual methodology, to deal with the lack of a formal relation between the SA description and the code.

Note that our approach differs from, for example, applying a coverage criterion directly over the complete LTS. Indeed, the ALTS is a tool to understand *what* must be tested. In other words, it represents a *model* of the system, whereas a coverage criterion is just a method to efficiently produce a set of test scenarios from the specified model.

The contribution of this paper is twofold: i) it develops a methodology for the extraction

of test specifications relevant at an architectural level, and ii) it tests out the viability of the approach in a real-world context. The empirical investigation with a case study was instrumental for the latter goal.

The remainder of this paper is organized as follows: Section II presents an overview on the approach; Section III outlines the case study to which we apply the approach. Section IV describes the approach in detail, and Section V presents its application to the case study. Sections IV and V may be read concurrently: they are organized to present the theory and practice, respectively, of each step of our approach. Section VI discusses some problems and insights gained, and Section VII considers related work. Section VIII summarizes the paper and presents future work plans.

II. An Overview of Our Approach to SA Testing

Several Architectural Description Languages (ADLs) rely on Labeled Transition Systems to model the SA dynamics [31], [38], [36], [24]. In our approach, we assume the existence of an SA description, in some ADL, and that an LTS model can be derived from such a description, whose node and arc labels represent, respectively, states and transitions relevant in the context of SA dynamics. Note that the LTS model expresses aspects such as concurrency, nondeterminism, asynchrony, and so on, relative to architectural nonsequential behaviors. We recall the definition of an LTS.

Definition II.1: A Labeled Transition System (LTS) 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 (actions) 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 $\mathcal{T} = \{ \xrightarrow{l} \subseteq \mathcal{S} \times \mathcal{S} \mid l \in \mathcal{L}\}$ is the transition relation labeled with elements of $\mathcal{L}$.

In the following, we use the terms LTS "labels" and "actions" interchangeably.

A path p on an LTS, where $p = S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \xrightarrow{l_3} \ldots \xrightarrow{l_n} S_n$, is a *complete path* if S_0 is the initial state and S_n is a final state. Hereafter, for the sake of brevity, an LTS path will also be denoted by its sequence of labels (e.g., $p = \mathrm{l}_1.\mathrm{l}_2. \ldots .\mathrm{l}_n$).

Our approach to SA-based testing consists of four logical steps. As the starting point 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 (*Step 1*). Applying each *obs*-function to the LTS, an Abstract LTS (ALTS) can be generated, which is a reduced LTS showing only interesting behaviors with respect to the selected view (*Step 2*). On this graph which is much more manageable than the original LTS, the software architect chooses a set of important patterns of behaviors (paths over the ALTS) to be tested (*Step 3*). Finally, these high-level tests are passed to the software tester, who actually has to run the tests and observe whether the current implementation "conforms" to its architectural model (*Step 4*). We say that *the implementation does not conform to the specification if some interactions described at the architectural level would not be allowed in the implementation.*

Steps 1 to 3 present a rigorous method to extract architectural tests from an SA specification. Step 4 deals with the execution of these tests at the code level. As far as this paper is concerned, we could not rely on a strictly formal refinement process from SA to code. So the last step is dealt with using a less formal approach than the first three. Indeed, applying the approach to the case study is not an a posteriori validation, but is instrumental in understanding the issues that concretely arise in the execution of SA-based tests at code-level, as well as in working out a systematic procedure to address them.

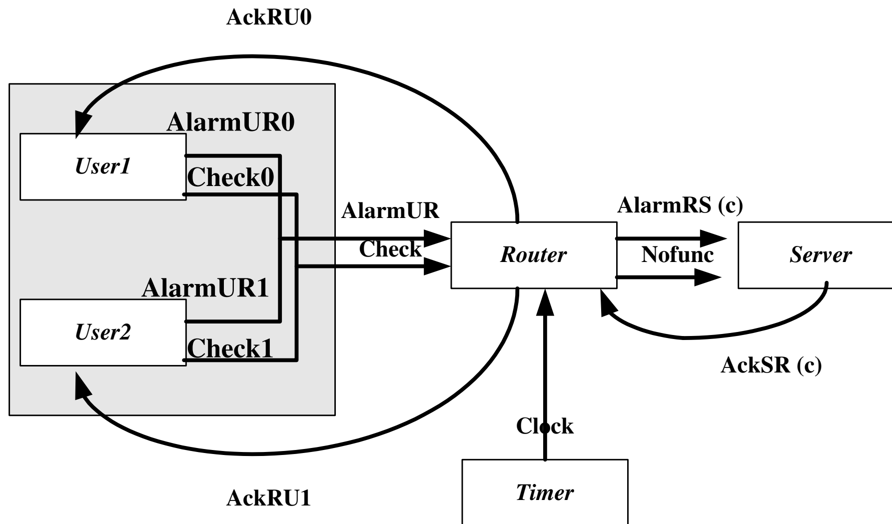

Fig. 1. TRMCS components and connectors

III. The TRMCS Case Study

The Teleservice and Remote Medical Care System (TRMCS) [32] is a system that provides monitoring and assistance to disabled or elderly people. A typical service is to send relevant information to a local phone-center so that the family, along with those concerned with medical or technical assistance, can be timely notified of critical circumstances.

From the informal requirements of TMRCS, we derive an SA model by defining the system topology (static view of components and connectors) and the system dynamics (the interactions).

The resulting TRMCS software architecture is shown in Figure 1. In it, boxes represent components (i.e., processing elements); arrows identify connectors (i.e., connecting elements, in this case channels), and arrow labels refer to the data elements exchanged through the channels. In particular, we identify four component types:

- User: sends either an "alarm" or a "check" message to the Router process. After sending an alarm, he/she waits for an acknowledgment from the Router. In our analysis, we

consider the case in which two Users can concurrently send alarms and checks.

- Router: waits for signals (check or alarm) from the User. It forwards the alarm messages to the Server and monitors the state of the User via check messages.
- Server: dispatches the help requests.
- Timer: sends a clock signal at each time unit.

The Java implementation of the TRMCS was developed independently, on the basis of the same informal requirements.

This case study serves three main purposes: to empirically validate the feasibility of the approach as a whole; to specifically work out the procedure relative to Step 4, as stated above; and finally, to verify to what extent the approach is able to identify conformance errors between the SA reference model and the implementation.

IV. The Approach

In this section, we describe the four steps of our approach to SA-based testing.

A. Step 1: Obs-*functions Definition*

From the SA specification of the system under analysis, we derive a Labeled Transition System that models the SA dynamics. In principle, this graph could directly be used as the reference model for deriving the test scenarios. The problem is that the LTS provides a global, monolithic description; it is a vast amount of information flattened into a graph. Extracting from this global model the observations of system behavior that are relevant for validation is a difficult task. This is a problem that always arises in formal testing: with the exception of very small routines, we need ways for exploring the LTS and deriving representative behaviors that constitute the test suite.

Our aim is to provide software architects with a key to decipher the LTS. A common practice in the analysis of complex systems is to derive from the global SA model a set of simplified models that provide different system views. Accordingly, the basic idea of our approach is to allow for the formal derivation from the LTS of reference models for testing, each representing a relevant pattern of behavior; with some abuse of terminology, we refer to each of the selected patterns of behavior as an *SA testing criterion*. Intuitively, an SA testing criterion abstracts away interactions that are uninteresting with regard to a specific test concern. Referring to the LTS definition, an SA testing criterion naturally partitions the LTS actions $\mathcal{L}$ into two groups: relevant interactions $\mathcal{R}$ (i.e., those we want to observe by testing) and nonrelevant interactions $\mathcal{NR}$ (i.e., those we are not interested in at this time), so that $\mathcal{L} = \mathcal{R} \cup \mathcal{NR}$ and $\mathcal{R} \cap \mathcal{NR} = \emptyset$.

We therefore associate with an SA testing criterion an *obs*-function that maps the relevant LTS labels to a specified interpretation domain $\mathcal{D}$, whereas any other (nonrelevant) one is mapped to a distinct element τ. The *obs*-function may also be considered as a *hiding* operator that makes a set of actions invisible to its environment and may *relabel* the others in an interpretation domain $\mathcal{D}$. The relabeling may help emphasize the semantic meaning of observable actions. More precisely:

$obs : \mathcal{L} \longrightarrow \mathcal{L}'$, so that $obs(r \in \mathcal{R}) = \mathrm{d} \in \mathcal{D}$, $obs(n \in \mathcal{NR}) = \tau$, and $\mathcal{L}' = \mathcal{D} \cup \tau$.

We can also extend the *obs*-function definition to LTS paths so that if $p = \mathrm{l}_1.\mathrm{l}_2.\ \ldots\ .\mathrm{l}_n$, $obs(p) = \mathrm{obs}(\mathrm{l}_1.\mathrm{l}_2.\ \ldots\ .\mathrm{l}_n) = obs(\mathrm{l}_1).obs(\mathrm{l}_2).\ \ldots\ .obs(\mathrm{l}_n)$.

B. Step 2: Abstract LTS Derivation

We use the *obs*-function as a means to derive a smaller automaton from the LTS which still expresses all high-level behaviors we want to test according to the selected SA testing criterion, but hides away any other irrelevant behavior. The automaton is called an *Abstract LTS* (ALTS).

The ALTS is obtained from the LTS via two transformations: i) by relabeling each transition $\mathcal{T} \in$ LTS according to the *obs*-function, and ii) by minimizing the resulting automaton with respect to a selected equivalence relation. We analyzed trace- and bisimulation-based equivalences, both familiar from the theory of concurrency [42]. If one wants to reduce as much as possible the number of τ transitions and corresponding nodes, then a *trace* equivalence can be considered. In fact, this equivalence abstracts from τ-labeled transitions and concentrates on any computational paths of transitions that are different from τ. A *bisimulation*-based equivalence is more suited when one wants to observe how the system evolves step-by-step, even along τ-moves (preserving the branching structure).

The relabeled automaton is called the ObsLTS, and the reduced one is the ALTS.

Figure 2 gives an example of the ALTS construction: the original LTS is analyzed, identifying the observation of interest (Figure 2.a); the abstraction is applied over this LTS with respect to the selected *obs*-function (Figure 2.b); the trace equivalence minimization function is applied. The resulting ALTS is shown in Figure 2.c. Figure 2.d, in contrast, presents a (bisimulation-based) branch minimization. As can be seen, the branch minimization is more informative since it gives more data on the original LTS structure.

Taking into consideration that i) the aim of ALTS is to provide a more compact and analyzable graph, and that ii) the ALTS automaton is built to highlight only interesting

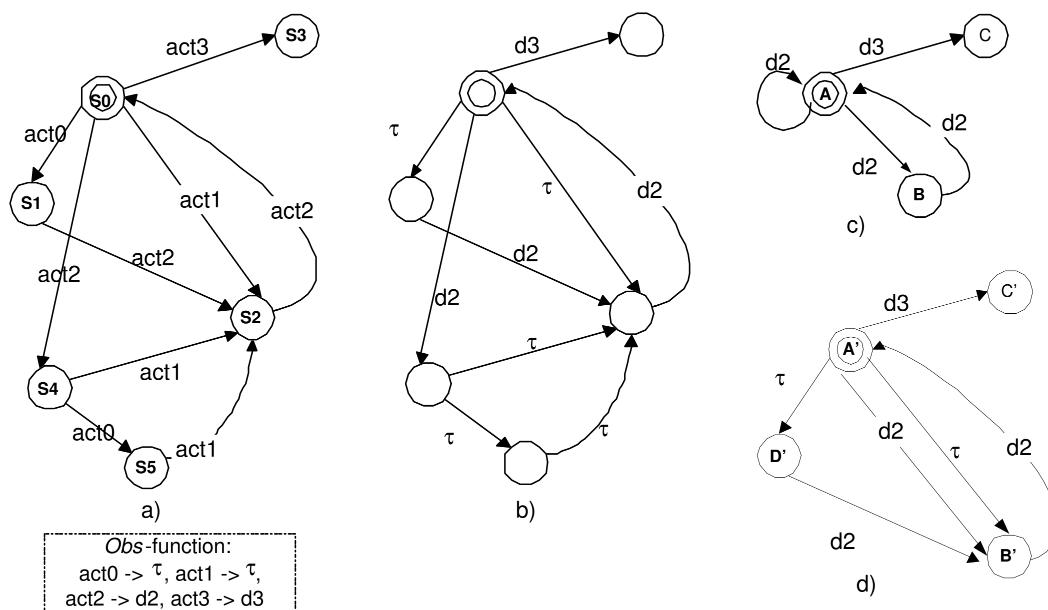

Fig. 2. a) LTS; b) ObsLTS; c) trace-based ALTS; d) bisimulation-based ALTS

behaviors, the trace equivalence is more suitable for our purposes. Moreover, full information about the LTS may be retrieved (if necessary) by using the algorithm explained in Appendix A.

The ALTS generation process[1] can be proved to be *correct* and *complete*, that is, each ALTS path comes from an LTS path (ALTS does not introduce new paths) and each LTS path can be mapped onto an ALTS path (ALTS does not loose information), as discussed in Appendix B.

C. Step 3: Test Selection

Once the ALTS associated with an SA testing criterion via an *obs*-function has been derived, the task of deriving an adequate set of tests according to the selected test criterion

[1]The ALTS derivation is completely automated: we customized the existing FC2Tools [21], taking advantage of the functions “fc2explicit” and “fc2weak.” A short description of the tools that support our approach can be found in Appendix C.

is converted into the task of deriving a set of complete paths appropriately covering the ALTS.

Given the relatively small dimensions of an ALTS, a tester might consider to apply extensive coverage criteria on it. However, ALTS paths clearly specify architectural behaviors at a higher abstraction level than the original LTS because they are based on an *obs*-function applied over the LTS. Thus, one ALTS complete path can correspond to many possible LTS paths (i.e., architectural tests). Therefore, less thorough coverage criteria seem more practical.

We apply here McCabe's [40], [54] test technique as it is a good compromise between arc and path coverage in the case of ALTS coverage; in fact, any base set of paths covers all edges and nodes in the graph (i.e., this coverage subsumes branch and statement coverage testing).

Clearly, any ALTS coverage criteria, such as McCabe's or other potential alternatives, need to be evaluated empirically, considering the cost and effectiveness of the obtained test sets (i.e., the number of test sequences and their respective effectiveness in detecting potential nonconformance failures). This evaluation, which would entail comparing various criteria applied over the same SA, is outside the scope of this paper.

When considering what to take as the specification of an "architectural test" we are left with two options. The first option, which is the one we take in this paper, is to consider an ALTS complete path as the test specification. In this case, the test is specified at a more abstract level and the tester instinctively focuses the testing on a restricted set of interactions (those in the ALTS alphabet). A second option could be to identify those LTS paths of which the selected ALTS path is an abstraction (this process is illustrated

in Appendix A). Because LTS paths are more detailed than ALTS paths, in this case the tester would have more information about how to perform the tests, but also stricter requirements; that is, the tester doesn't have as much freedom in choosing the code-level tests. In practice, it might actually be more difficult to test the conformance of source code to the test specification.

In either case (ALTS or LTS path), an architectural test is essentially a *sequence of system actions that are meaningful at the SA level*.

D. Step 4: Code-Level Testing

In this section, we discuss how a tester can use the architectural paths, as produced in Section IV-C, to test the implementation. In general, when the test cases are derived from an analysis of the specifications, two major problems remain to be solved: traceability and test execution.

- **Traceability** concerns "relating the abstract values of the specification to the concrete values of the implementation" [19]. Here we are concerned with traceability from the SA-level of abstraction. Several researchers have recognized the importance and difficulty of this problem [55], [48], but no one has yet found a general solution.

If a well-formalized architecture-based development process is in place, SA specifications can be used to formally drive the generation of low-level design and code, and thus the correspondence is maintained throughout the process. For instance, some ADLs (such as C2ADL [12] and xADL [18]) provide development support for implementing software architectures in Java and C++ [13]. Explicit mapping rules drive the source-code implementation of architectural components, connectors, and messages via objects.

Considering specifically the process followed in the experience described in this paper, due to real-world constraints SA specifications and low-level design have been intermixed without any formal mapping, but using ad-hoc monitoring and code-analysis techniques. While this is certainly not the ideal process, it is a realistic and plausible approach, and we discuss some interesting insights gained. The procedure we followed consists of Step 4.1 and Step 4.2, described later in this section.

- **Test execution** entails forcing the Implementation Under Test (IUT) to execute the specific sequence of events that has been selected. This is a difficult task, which becomes harder in distributed and concurrent settings, where non-determinism is introduced. In fact, depending on the system state, a same input might excite different sequences of interactions (among several concurrent processes) and produce different results. A detailed discussion of this problem is outside the scope of this paper. In brief, one can distinguish between deterministic- and nondeterministic-testing approaches, as discussed below in Step 4.3.

In the following, we show how Step 4 is divided into three substeps, explaining how we tackled both problems.

Step 4.1: In this substep we map architectural components and actions (identified by the LTS labels) into their low-level implementation. As mentioned before, our experience was conducted in a realistic context, in which we could not make any assumptions on the existence of formal relationships between software architecture and design elements. We thus analyzed the system implementation to understand how architectural actions (e.g., high-level functionalities) have been implemented in the code by sequences of partially ordered method calls. Note that more than one implementation sequence, might correspond

to one LTS action. In such cases, to test the architectural action, all of them should be considered. Of course, the same implementation action (e.g., a method call) may occur in several implementation sequences, corresponding to different architectural actions. This is not a problem, however, because we test sequences and not single implementation actions.

Step 4.2: This second substep takes into account how an ordered sequence of actions (i.e., an architectural test) is implemented by a sequence of low-level functions. Since the TRMCS implementation is in Java, we map actions into sequences of method calls. If each action is implemented, at the low level, by a sequence of methods calls, it would be useful to understand how sequences of these actions (i.e., an architectural test) are implemented by the source code.

Two alternatives may be taken into account: i) each action is implemented by a sequence of operations, and they run sequentially; or ii) the actions can run concurrently. In the former case, a sequence of architectural actions is implemented by the sequential execution of the respective low-level sequence diagrams. In the latter case, the operations may interleave with each other. Note that in this context, "act_i before act_j" does not mean that all the operations implementing act_i must run before all the operations implementing act_j. It means that some operations that identify the action termination must be executed following a given order, whereas the others may be run in any order. In Section V, we further develop this idea in the context of the TRMCS system, and the operations identifying the termination of actions are denoted as Synchronization Methods (SMs).

Step 4.3: At this point, we run the code and evaluate the execution traces with respect to the expected ones (i.e., those identified in Step 4.2) to analyze the source code conformance with respect to the architectural behavior. To execute the desired test sequence, our

pragmatic approach is to repeat the launching of a program run under some specified input conditions several times until the sequence is observed (or a maximum number of iterations is reached).

In the literature, this is called a *nondeterministic testing* approach. In contrast, the *deterministic testing* approach (firstly proposed by Carver and Tai [16]) forces a program to execute a specified test sequence by instrumenting it with synchronization constructs that deterministically reproduce the desired sequence. Instrumentation is commonly used in software testing [28], but care must be taken to contain the consequent execution overhead and avoid the triggering of unexpected behaviors. Alternatively, a language-based dynamic approach for Java requiring no instrumentation has been recently proposed in [35]. A test driver automatically executes the calls as specified in a test sequence, controlling the synchronization of the Java threads through a clock. A related, yet different problem is execution replay for reproducing an observed trace, as required, e.g., during debugging to force a series of synchronizarion events that led to a failure. A tool for the deterministic replay of Java multithreaded applications is presented in [17].

V. Applying the Approach

We now apply the approach introduced in the previous section to the TRMCS case study presented in Section III. We specify the TRMCS components behavior by using the Finite State Process (FSP) [38] language. The behavior of each component is modeled by one or more FSP processes, and each process is described by an LTS which is automatically generated by the Labeled Transition System Analyzer (LTSA) tool [34].

By running the LTSA tool on the TRMCS FSP specification, we obtain a (minimized)

LTS composed of 256 states. The LTS labels of interest for our analysis are:

1) **u[i].sendAlarm_To_Router**: an Alarm msg is sent by $User_i$ to the Router;

2) **u[i].receiveAck_From_Router**: the Router Acknowledgment msg is received by $User_i$;

3) **r.[i].sendAlarm_To_Server**: the Router forwards the Alarm msg to the Server;

4) **sa[0].sendAck_To_Router**: the Server sends the Ack to the Router;

5) **r.sendNoFunc_To_Server**: the Router sends the NoFunction msg to the Server.

The steps presented in the following description are the same as those presented in the previous section.

A. Applying Step 1

Given the informal description of the TRMCS in Section III, due to obvious safety-critical concerns, we may want to test the flow of an Alarm message, from the moment a User sends it to the moment the User receives an acknowledgment. In the terminology used in Section IV, the software architect may decide that an important SA testing criterion is *"all those behaviors involving the flow of an Alarm message through the system."*

From this quite informal specification, a corresponding *obs*-function can be formally defined. This is called "AlarmObs" and is shown in Figure 3.a. The set of relevant actions $\mathcal{R}$ for AlarmObs contains all of the actions (i.e., elements of the LTS alphabet) that specifically involve sending an Alarm message by a User (label 1 above), or the User's subsequent reception of the Router acknowledgment (label 2). The corresponding actions must be relabeled in $\mathcal{D}$ according to their semantic interpretation. As shown in Figure 3.a, the four actions involving Alarm sending and Ack receiving are relabeled ($Alarm_i$Dispatch, Ack_iDispatch), and any other is hidden.

D = {Alarm1Dispatch, Alarm2Dispatch, Ack1Dispatch, Ack2Dispatch}
obs (**u.0.sendAlarm_To_Router**) = **Alarm1Dispatch**: *User1 issues an Alarm msg* *obs* (**u.0.receiveAck_From_Router**) = **Ack1Dispatch** : *User1 receives an Ack* *obs* (**u.1.sendAlarm_To_Router**) = **Alarm2Dispatch** : *User2 issues an Alarm msg* *obs* (**u.1.receiveAck_From_Router**) = **Ack2Dispatch** : *User2 receives an Ack*
For any other rule r, *obs* (r) = τ

a)

D = {FRa1, FRa2, TRack1, TRack2, FRno}
obs (**r.[0].sendAlarm_To_Server**) = **FRa1** : *Alarm1 from Router to Server* *obs* (**r.[1].sendAlarm_To_Server**) = **FRa2** : *Alarm2 from Router to Server* *obs* (**sa[0].sendAck_To_Router**) = **TRack1** : *Ack1 from Server to Router* *obs* (**sa[1].sendAck_To_Router**) = **TRack2** : *Ack2 from Server to Router* *obs* (**r.sendNoFunc_To_Server**) = **FRno** : *NoFunctioning From Router to Server*
For any other rule r, *obs* (r) = τ

b)

Fig. 3. a) AlarmObs *obs*-function; b) ServerRegression *obs*-function

An alternative scenario could be that the TRMCS is already functioning and one of the components is being modified. We then want to test whether the modified component still interacts with the rest of the system in conformance with the original description of the SA. In this case, the observation point of the software architect is *"all the interactions that involve this component."* If, for instance, the component being modified is specifically the Server, then the corresponding *obs*-function is the one given in Figure 3.b: FR{a1, a2, no} is used to relabel the messages the Server receives from the Router, and TR{ack1, ack2} is used to relabel the messages sent to the Router. This SA testing criterion is referred to hereafter as "ServerRegression."

B. Applying Step 2

With reference to the AlarmObs criterion, after applying reduction and minimization algorithms, we obtain the ALTS depicted in Figure 4.a (the shaded circle represents the

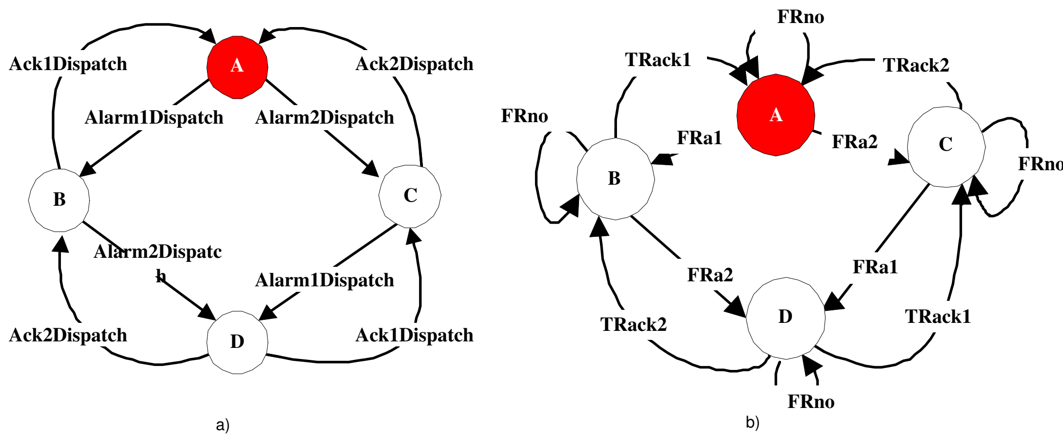

Fig. 4. a) AlarmObs ALTS; b) ServerRegression ALTS

initial state, which 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., Alarm1Dispatch), the system can nondeterministically react with one of two possible actions: i) elaborate this Alarm and send back an Acknowledgment (Ack1Dispatch), or ii) receive another Alarm message from another User (Alarm2Dispatch).

With reference to the ServerRegression criterion, the ALTS in Figure 4.b is obtained. It shows the interactions involving the Server component: it can receive Alarm1 ("FRa1") or Alarm2 ("FRa2") from the Router, and afterwards it can nondeterministically receive other Alarms or send back Acknowledgments ("TRack$_i$"). At any moment, it can receive the NoFunc message from the Router ("FRno").

C. Applying Step 3

With reference to the ALTS derived for the AlarmObs criterion (Figure 4.a), a list of test paths derived according to McCabe's coverage criterion [54] follows with the corresponding Test Sequences (TSs):

Path1_a: A B A, TS1_a:Alarm1Dispatch.Ack1Dispatch;

Path2_a: A B D B A, TS2_a:Alarm1Dispatch.Alarm2Dispatch.Ack2Dispatch.Ack1Dispatch;

Path3_a: A B A C A, TS3_a:Alarm1Dispatch.Ack1Dispatch.Alarm2Dispatch.Ack2Dispatch;

Path4_a: A B D C A, TS4_a:Alarm1Dispatch.Alarm2Dispatch.Ack1Dispatch.Ack2Dispatch;

Path5_a: A B D C D C A, TS5_a:Alarm1Dispatch.Alarm2Dispatch.Ack1Dispatch.Alarm1Dispatch.

Ack1Dispatch.Ack2Dispatch.

Let us consider, for example, Paths 2_a, 3_a, and 4_a. These three paths are aimed at validating that no Alarm messages in a series of two is lost, irrespectively of the order in which they are processed. Moreover, for conformance testing purposes, we want to check whether the system implementation enables all the sequences of actions envisioned at the architectural level.

With reference to the ServerRegression observation and the associated ALTS (Figure 4.b), McCabe's coverage criterion yields the following set of test paths with the corresponding TS lists:

Path1_b: A A, TS1_b: FRno;

Path2_b: A B A, TS2_b: FRa1.TRack1;

Path3_b: A B B A, TS3_b: FRa1.FRno.TRack1;

Path4_b: A B D B A, TS4_b: FRa1.FRa2.TRack2.TRack1;

Path5_b: A B A C A, TS5_b: FRa1.TRack1.FRa2.TRack2;

Path6_b: A B D C A, TS6_b: FRa1.FRa2.TRack1.TRack2;

Path7_b: A B D D B A, TS7_b:FRa1.FRa2.FRno.TRack2.TRack1;

Path8_b: A B D C C A, TS8_b: FRa1.FRa2.TRack1.FRno.TRack2;

Path9_b: A B D C D B A, TS9_b: FRa1.FRa2.TRack1.FRa1.TRack2.TRack1.

Paths 1_b, 3_b, 7_b and 8_b, for example, are meant to verify that the NoFunc message (FRno in Figure 4.b) can be received by the Server at any moment.

D. Applying Step4

We now concentrate on the test sequences generated from the AlarmObs ALTS in Section V-C. These tests involve three different components (User1, User2, and Router) and two architectural events (AlarmDispatch and AckDispatch). By following the three steps defined in Section IV-D, we have:

Applying Step 4.1:

AlarmDispatch and AckDispatch are the events of interest when considering the selected architectural tests. We need to understand how the source code implements these architectural actions.

The AlarmDispatch operation represents a system Input, and what we expect is that the alarm process is started when the User pushes the Alarm button of the hardware device. Following this reasoning, we start by analyzing the actions associated with pushing the Alarm button, and we find that the Alarm functionality is implemented by two different Java processes: the Alarm message is written 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 on a defined socket port.

The objects and the methods that implement the Alarm are shown in Figure 5. Let us very briefly comment on it: from the User side (Fig. 5.al), the xUser object creates the User object, if the if_1 condition is verified. After some time (in which internal operations can be performed), if Users press the Alarm button (if_2), they thereby create a ClientConnection and the SendNMessages (SNM) objects. The User calls the send() method of the SNM object, which tries “n” times to create the socket. If the socket is created (if_3), the SNM

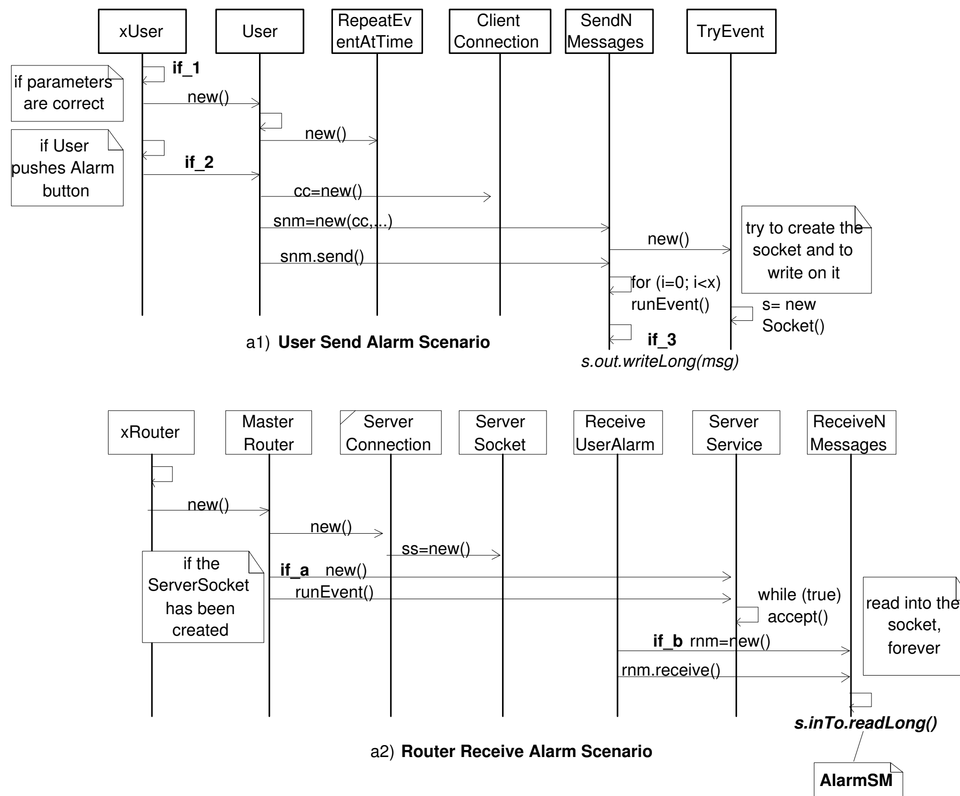

Fig. 5. The implementation of the AlarmDispatch

finally writes the Alarm message into the socket. From the Router side (Fig. 5.a2), the Router graphical interface object (xRouter) creates a new instance of the MasterRouter object, which in turn creates the ServerConnection object. It tries to create a ServerSocket "ss", and then a process is activated that continuously checks the ServerSocket port. In this way, when a new message is written in the socket (if_b) the socket is read and the Alarm is received.

A similar analysis has been conducted for the Ack function.

Applying Step 4.2:

Now we will analyze how an architectural test (i.e., a sequence of actions) is implemented by the code. Let us consider as an example $TS3_a$, Alarm1Dispatch. Ack1Dispatch.

Alarm2Dispatch. Ack2Dispatch. The actions of interest in this test sequence are Alarm_iDispatch and Ack_iDispatch (for i=1,2), and their order is such that Alarm1Dispatch happens before Ack1Dispatch, which happens before Alarm2Dispatch, which happens before Ack2Dispatch.

At the architectural level, Alarm1Dispatch happens before Ack1Dispatch if the second action is run after the first one has terminated. At the code level, the ordering relation "happens before" can be interpreted in two ways:

i) If the code-level sequences implementing the Alarm_iDispatch (Figure 5) and the Ack_iDispatch actions are run *sequentially*, Alarm1Dispatch happens before Ack1Dispatch if the Alarm scenarios in Figure 5 are run to completion before the Ack scenarios start.

ii) If the low-level scenarios can be concurrently run, their method calls may interleave. In particular, if the implementations for Alarm_i and Ack_i run concurrently, the methods in their scenarios could be called in a different order. In this context, then, what does "Alarm_i happens before Ack_i" mean? To answer this question, we identify those methods (Synchronization Methods, or SMs) that actually perform an architectural action. Therefore, Alarm1 before Ack1 means that the Alarm1 SM happens before the Ack1 SM, whereas other methods may be called in any order.

We are aware that identifying the SMs is a nontrivial task in a development process not based on strictly formalized refinement steps: in our experimentation with the TRMCS case study, this was not so difficult due to a straightforward correspondence between LTS labels and (reading and writing) operations performed over Java sockets. In Figure 5 the "s.inTo.readLong()" method represents the SM for Alarm.

Applying Step 4.3:

At this point we need to execute the test sequences and verify whether the system im-

plementation behaves, at run-time, as described by our SA test. As stated at the end of Section IV, to perform this analysis we apply a nondeterministic testing approach, instrumenting the TRMCS program P to capture run-time information. Note that the instrumentation does not introduce new threads of execution, thus reducing instrumentation overheads.

More precisely, given the TRMCS implementation P we build an instrumented version P' able to store information about the execution of the SMs. To implement this new feature, we introduce the "SynchronizationMethod" file; when an SM is called by the system, a print operation is performed in this file which captures the following information: <SM ID, User ID, SM instance, time>. For example, <AlarmSM, User 1, Alarm 3, 974910169250> records that the synchronization method AlarmSM (in Figure 5) has been called (at the system time 974910169250) in response to the third instance of an Alarm message issued by User1.

By running P' several times with the User1 and User2 Alarms as inputs in different orders, we obtain a SynchronizationMethod file report. The information collected by the report is a sequence of pairs of lines such as the following couple:

..., <AlarmSM, User i, Alarm j, ti>, <AckSM, User i, Alarm j, ti+ϵ>, ...

In other words, each Alarm operation is always immediately followed by the relative Acknowledgment, as described by behavior $TS3_a$ in Section V.C. This implies that we never observe behaviors $TS2_a$ and $TS4_a$. Triggered by this observation, code analysis pictures a conformance fault with respect to the architectural expected behaviors.

VI. Some Considerations

Considering the four steps of our approach, it is possible to make an empirical evaluation of the *difficulty of applying it* and *its generality* which can then lead to further lines of research.

A. Difficulty of Application

Step 1. Obs-functions definition: The definition of an *obs*-function of interest is an empirical task, based on a software architect's experience. This task could be made easier through a classification of observations and by assigning semantics to the messages exchanged in the SA model. Methods similar to SAAM [33] or SCENT [49], in which architectural information is empirically captured, could help.

Step 2. Deriving ALTS: As shown, the ALTS generation can be automated. In this work, we adapt the FC2Tools to reach our goal, but other tools (such as Caesar/Aldebaran [14]) can be used as well.

Step 3. Test Classes Selection: To cover the ALTS entails applying a coverage criterion over this graph, extracting only paths that expose different behaviors. Coverage criteria, such as McCabe's [40] test technique, which is used here, could be easily automated.

Step 4. Code-level Testing: This is the most difficult step that the tester needs to apply to generate test cases. To trace high-level information into code-level tests entails relating the abstract values of the specification to the concrete values of the implementation. Four general problems appear: i) Understanding which classes and methods implement an architectural functionality is usually not obvious. Solving this problem may be facilitated by using a more informative notation for LTSs, in which Inputs and Outputs are identified

at the specification level (IO Automata [37]) and can then be compared to code-level Inputs and Outputs. ii) More than one sequence can implement the desired behavior. This problem can be handled only by requiring the intervention of the tester, who manually evaluates each point of decision and performs a reasoned choice regarding which selection is most relevant for testing purposes. iii) When deriving the code-level sequences, we can end up having to consider a lot of functionalities that are not described in the architectural sequences: notably, in the SA description we consider only interactions, but not computations. iv) The architectural model usually describes only the expected behaviors, but the implementation has to handle the exceptional behaviors as well.

Currently, our approach is not entirely formal and a discontinuity between formality (Steps 1 to 3) and informality (Step 4) can be noticed. One way to cope with this problem, is to make stronger assumptions on the process, explicitly referring to a formal mapping between architectural elements and implementation objects. This solution has been investigated in [44], as outlined in Section VIII.

B. Approach Generality

To put our approach into practice, we selected an ADL to describe the SA dynamics through an LTS, we selected a coverage criteria to extract a limited number of ALTS paths, and we used an implementation following the object-oriented paradigm.

However, we kept our approach as independent as possible from these choices: the only constraint we required is that the SA dynamics be described via an LTS conforming to the definition given in Section II. Different ADLs can be used to describe the SA. Different ALTS coverage criteria can be applied and different implementation languages

and programming paradigms can be used. Finally, several 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 can be directly mapped on the implementation [41].

VII. Related Work

The approach we have proposed here makes use of several concepts: testing concurrent and real-time systems, specification-based testing, conformance testing, tracing information, and architectural testing. This section is a brief, but for reasons of space incomplete, overview of these research areas.

Much work has been devoted to **testing concurrent and real-time systems**, both specification driven [16], [39], [15] and implementation based [29], [4], [17], [35]. These references address different aspects, from modeling time to internal nondeterminism, but all focus on unit testing. Our aim is different: we want to derive test plans for integration testing. Thus, although the technical tools some of these approaches use are the same as ours (e.g., LTS, abstractions, event sequences), we use them in a different way. This difference in goals emerges from the very beginning of our approach: we work on an architectural description that drives our selection of the abstraction and of the paths.

Our approach to defining ALTS paths in order to specify high-level test classes has much in common with Carver and Tai's use of the synchronization sequence (SYN-sequence) constraint, for specification-based testing of concurrent programs [16]. Indeed, sequencing constraints specify restrictions to apply on the possible event sequences of a concurrent

program when selecting tests, similar to what ALTS paths do for a SA. Again, the main difference is in the distance of the abstraction level of the reference model from the implementation. This prevents the straightforward application of their results for SA testing. For example, the technique defined in [16] was implemented in [4] for Java programs, and the SYN-sequences are runtime program executions; in our approach, the SYN-sequences are the test classes, and thus they are more abstract.

A number of methods regard the **selection of a test suite based on the specification of the implementation under test**, which is assumed to be given in the form of a Finite State Machine (FSM). The Transition tour, W-method, DS-method, UIO method and many variants of them (such as the Wp-method) have been proposed. A review of these approaches is given in [10]. Bochmann and colleagues present a test generation tool (TAG) based on FSM and generated by a specification [51]. The drawbacks of this tool are due to the use of FSM with a limited number of states in the implementation and the impossibility of ensuring a complete coverage with nondeterministic specifications.

Other approaches have been proposed to **test software systems using statecharts**: In [49], natural language scenarios are formalized through statecharts, annotated with helpful information for test case generation. Test cases are then generated through path traversal. Bogdanov [11] presents a method to extract test cases from statecharts in order to prove that the implementation is behaviorally equivalent to the design. The Das-Boot tool presented in [53] generates a test suite for object-oriented systems specified through statecharts.

Several authors [25], [52], [22] have recently dealt with the problem of automatically generating test suites to **test the conformance** of an implemented system to its spec-

ification. Bochmann and colleagues [25], [10] use FSMs to drive a conformance testing approach: the system specification (S) and its implementation (I) are formalized using FSMs; "conformance relations" (equivalence, quasi-equivalence, reduction) are defined and the Wp-method is used to generate test sequences.

Tretmans [52] presents some approaches for LTS-based conformance testing. The specification (S) is formalized using LTSs, the implementation (I) is expressed using LTSs or Input/Output Transition Systems (IOTSs), and tests (T) are formalized using LTSs. Given S and one of its possible implementations I, Tretmans defines some "implementation relations" (imp) to correlate S with I: I conforms to S iff I is "imp" with respect to S.

Fernandez et al. [22], [23] use Input/Output LTS (IOLTS) to formalize the specification, the implementation, and the test purposes. They propose an approach for an automatic, on-the-fly generation of test cases, embodied in the Test Generation and Verification (TGV) environment [23].

We also use the SA-derived LTS as a reference model to derive test cases but in a different manner. The behavioral model of the implementation under test could be modeled through an LTS (or an IOLTS). This model is required in order to properly define what conformance means in a formal manner. Based on these models, they define some implementation relations (conf, ioconf, ioco, etc.) and they test whether the implementation is correct with respect to the specification. In our approach, we do not assume that we are able to produce an LTS over the implementation. In fact, we compare architecture level sequence of events with lower level execution paths.

The problem of **tracing information** is not new, and some relevant papers are listed

in the references of the following sources. Egyed [20] shows a way to detect traceability between software systems and their models, and gives a list of interesting references on traceability techniques. Some work has been done in bridging the gap between requirements and software architectures (e.g., [50], [45]), and much other work addresses the traceability of requirements (e.g., [26]). The problem of mapping abstract tests into the System Under Test is currently being studied in the AGEDIS project [1].

The topic of **architectural testing** has recently raised some interest [47], [7], [27], [28], [48]. In [47], the authors define six architectural-based testing criteria, adapting specification-based approaches; in [7], the authors analyze the advantages in using SA-level testing for reuse of tests and to test extra-functional properties. In [27] Harrold presents approaches for using software architecture for effective regression testing, and in [28], she also discusses the use of software architecture for testing. In [48], the authors present an architecture-based integration testing approach that takes into consideration architecture testability, simulation, and slicing.

All of the above works give interesting insights. However, to the best of our knowledge, this paper represents the first attempt to tackle the whole cycle of SA-based testing with a comprehensive and concrete approach. It spans the spectrum from test derivation based on architecture dynamics down to test execution over system implementation, and relies on empirical hands-on experience derived from a real-world case study.

VIII. Conclusions and Future Work

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 improve the conformance testing of a large distributed system. As such, this paper relies on two prolific branches of current software engineering literature: software architecture and specification-based testing.

In terms of software architecture, although the literature is rich in models and tools for design and analysis based on SA [24], very little has been proposed for SA-based testing. Despite the claim that SA must play a role throughout the software life cycle, there are still many problems due to the practical implications of using SA models and tools. In particular, while it may be relatively easy to conceive of a method for deriving suites of architectural test cases according to some notion of coverage, subsequently establishing a relationship between these high-level tests and the implementation under test may be quite difficult, and as yet no complete solution exists.

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 our approach and existing approaches:

i) The SA description tries to capture SA-relevant behaviors alone, while abstracting away other system functions. Thus, our tests specifically belong to the integration testing stages and certainly do not aim to test the system as completely as possible, as in traditional specification-based test approaches.

ii) The abstraction level of the reference model and its relative "distance" from the implementation under test may vary much in the two contexts. In SA-based approaches, this distance is *purposely* very high, whereas in existing approaches to specification-based testing, this is often assumed to be low. In other words, a high abstraction level is a basic assumption of our approach, whereas traditional approaches require a close distance

between the reference model and the implementation, and typically both are represented by automata [52].

We have provided here an excerpt from the empirical insights gained while performing SA-based testing on the TRMCS case study. Although the experience reported may appear to be specific to the case study, we believe that in most cases the problems encountered as well as the solutions found can easily be generalized to any SA-centered context.

Our future work will go in several directions. We consider the approach presented here as a foundational contribution that might not be directly applied in practice. For example, we might not always have a global architectural model at our disposal. This can happen for several reasons: i) architectural components may be described through complex models, in terms of states and transitions and putting these models together may give rise to a state explosion problem. ii) The architectural models may be incomplete, which means that some component behaviors are unknown or components are not completely specified. These are very common situations in industrial contexts. We are currently investigating the possibility of generating abstract observations and test cases directly from partial architectural models.

Another area under investigation is how to further formalize our approach, with stricter assumptions on the SA-based development process. In [44] we show how the testing process proposed in this paper can be made completely systematic, by specializing and refining our general approach to a more specific context. By adopting a C2 style architecture [12] and the related C2 framework [13], we were able to handle the traceability/mapping among SA and code problem (by adapting the category partition method [46]) and the execution over the identified test cases problems (through the Argus-I tool [2]). The approach has

been applied to a case study, gaining interesting results.

Acknowledgments

The authors would like to acknowledge the Italian MURST/MIUR national projects SALADIN and SAHARA, which partially supported this work as well as Thierry Jeron and the anonymous reviewers for their constructive comments and suggestions.

References

[1] AGEDIS Project. Automated Generation and Execution of Test Suites for Distributed Component-based Software. On-line at: <http://www.agedis.de/index.shtml>.

[2] The Argus-I project. University of California, Irvine. Information on-line at <http://www.ics.uci.edu/∼mdias/research/ArgusI>.

[3] L. Bass, P. Clements, and R. Kazman. Software Architecture in Practice. *SEI Series in Software Engineering*, Addison-Wesley, 1998.

[4] A. Bechini and K.-C. Tai. Design of a Toolset for Dynamic Analysis of Concurrent Java Programs. In *Proc. IEEE 6th Int. Workshop on Program Comprehension*, pp. 190-197, June 1998.

[5] G. Bernot, M. C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, Vol. 6, N. 6, pp. 387-405, 1991.

[6] A. Bertolino. Knowledge Area Description of Software Testing. In *SWEBOK*, Joint IEEE-ACM Software Engineering Coordinating Committee. On-line at: <http://www.swebok.org>.

[7] A. Bertolino and P. Inverardi. Architecture-based software testing. In *Proc. ISAW96*, October 1996.

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

[9] A. Bertolino, P. Inverardi, and H. Muccini. An Explorative Journey from Architectural Tests Definition downto Code Tets Execution. In *IEEE Proc. Int. Conf. on Software Engineering* (ICSE2001), pp. 211-220, May 2001.

[10] G. v. Bochmann and A. Petrenko. Protocol Testing: Review of Methods and Relevance for Software Testing. In *ACM Proc. Int. Symposium on Software Testing and Analysis*, ISSTA '94, pp. 109-124, 1994.

[11] K. Bogdanov. Automated Testing of Harel's Statecharts. Ph.D. thesis, The University of Sheffield, 2000.

[12] The C2 style and ADL. Project web page on-line at: <http://www.isr.uci.edu/architecture/c2.html>.

[13] The C2 Framework. On-line at: <http://www.isr.uci.edu/architecture/software.html>.

[14] Caesar/Aldebaran Tool (CADP). On-line at: <http://inrialpes.fr/vasy/cadp>.

[15] R. Cardell-Oliver and T. Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. In Anders P. Ravn and Hans Rischel (Eds.), *Proc. 5th Int. School and Symposium on Formal Techniques in Real Time and Fault Tolerant Systems* (FTRTFT'98), pp. 251-261. *LNCS 1486*, September 1998.

[16] 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.

[17] J. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *ACM Proc. of the Sigmetrics Symposium on Parallel and Distributed Tools*, pp. 48-59, 1998.

[18] E. M. Dashofy, A. van der Hoek, and R.N. Taylor. An Infrastructure for the Rapid Development of XML-Based Architecture Description Languages. In Proc. of the 24th Int. Conf. on Software Engineering, 2002.

[19] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J.C.P. Woodcock and P.G. Larsen (Eds.), *FME'93: Industrial-Strenght Formal Methods*, pp. 268-284. *LNCS 670*, Springer Verlag, 1993.

[20] A. Egyed. A Scenario-Driven Approach to Traceability. In *IEEE Proc. Int. Conf. on Software Engineering* (ICSE2001), pp. 123-132, May 2001.

[21] FC2Tools. On-line at: <http://www-sop.inria.fr/meije/verification/quick-guide.html>.

[22] 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. of the Eighth Int. Conf. on Computer Aided Verification* (CAV'96), USA, pp. 348-359, 1996. *LNCS 1102*, Springer, 1996.

[23] J.-C. Fernandez, C. Jard, T. Jeron, L. Nedelka, and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. Special Issue of *Science of Computer Programming*, Vol. 29, pp. 123-146, 1997.

[24] Formal Methods for Software Architectures. Tutorial book on Software Architectures and formal methods. In SFM-03:SA Lectures, Eds. M. Bernardo and P. Inverardi, LNCS 2804, 2003.

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

[26] O. C. Z. Gotel and A. C. W. Finkelstein. An Analysis of the Requirements Traceability Problem. In *IEEE Proc. of the First Int. Conf. on Requirements Engineering* (ICRE '94), Colorado, USA., pp. 94-102, April 1994.

[27] M. J. Harrold. Architecture-Based Regression Testing of Evolving Systems. In *Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis* (ROSATEA), CNR-NSF, pp. 73-77, July 1998.

[28] M. J. Harrold. Testing: A Roadmap. In A. Finkelstein (Ed.), *ACM ICSE 2000, The Future of Software Engineering*, pp. 61-72, 2000.

[29] D. M. Hoffman and P. A. Stropper. Techniques and tools for Java API testing. In *IEEE Proc. of the 2000 Australian Software Engineering Conference*, pp. 235-245, 2000.

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

[31] P. Inverardi and A. L. Wolf. Formal Specifications and Analysis of Software Architectures Using the Chemical Abstract Machine Model. *IEEE Trans. on Software Engineering*, Vol. 21, N. 4, pp. 100-114, April 1995.

[32] P. Inverardi and H. Muccini. The Teleservices and Remote Medical Care System (TRMCS). In *IEEE Proc. IWSSD-10*, San Diego, California, November 2000.

[33] R. Kazman, L. Bass, G. Abowd, and M. Web. SAAM: A Method for Analyzing the Properties of Software Architectures. In *Proc. Int. Conf. on Software Engineering* (ICSE16), Sorrento, Italy, pp. 81-90, May 1994.

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

[35] B. Long, D. Hoffman, and P. Strooper. Tool Support for Testing Concurrent Java Components. *IEEE Trans. on Soft. Engineering*, Vol. 29, N. 6, pp. 555-566, June 2003.

[36] D. C. Luckham, J. J. Kenney, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Trans. on Software Engineering*, Special Issue on Software Architecture, Vol. 21, N. 4, pp. 336-355, April 1995.

[37] N. A. Lynch and M. R. Tuttle. An Introduction to Input/Output Automata. In *CWI Quarterly*, Vol. 2, N. 3, pp. 219-246, September 1989.

[38] J. Magee and J. Kramer. *Concurrency: State Models & Java Programs*. Wiley, April 1999.

[39] 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, November 1995.

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

[41] N. Medvidovic, D. S. Rosenblum and R. N. Taylor. A Language and Environment for Architecture-based Software Development and Evaluation. In *IEEE Proc. Int. Conf. on Software Engineering* (ICSE'99), pp. 44-53, 1999.

[42] R. Milner. *Communication on Concurrences*. International Series on Computer Science. Prentice Hall, 1989.

[43] H. Muccini, A. Bertolino, and P. Inverardi. Using Software Architecture for Code Testing. Long version of the paper submitted for publication to IEEE TSE. On-line at <http://www.HenryMuccini.com/Research/TSE01.htm>.

[44] H. Muccini and M.Dias. Systematic Testing of Software Architectures in the C2 style. Submitted for publication. On-line at: <http://www.HenryMuccini.com/Research/ETAPS04_Submitted.htm>.

[45] B. Nuseibeh. Weaving Together Requirements and Architectures. In *IEEE Computer*, Vol. 34, No. 3, pp. 115-117, March 2001.

[46] T. J. Ostrand and M. J. Balcer. The Category-Partition Method for Specifying and Generating Functional Tests. *Communications of the ACM*, Vol. 31, N. 6, pp. 676-686, June 1988.

[47] 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.

[48] D. J. Richardson, J. Stafford, and A. L. Wolf. A Formal Approach to Architecture-based Software Testing. Technical Report, University of California, Irvine, 1998.

[49] J. Ryser and M. Glinz. A Practical Approach to Validating and Testing Software Systems Using Scenarios. In *Proc. QWE'99: Third International Software Quality Week Europe*, Brussels, Nov. 1999.

[50] Straw '01. First Int. Workshop "From Software Requirements to Architectures" (STRAW'01), May 14, 2001, Toronto, Canada.

[51] Q. M. Tan, A. Petrenko, and G. v. Bochmann. A Test Generation Tool for Specifications in the Form of State Machines. In *IEEE Proc. Int. Communications Conference* (ICC) 96, pp. 225-229, June 23-27, 1996.

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

[53] M. Vieira, M. Dias, and D. J. Richardson. Object-Oriented Specification-Based Testing Using UML Statechart Diagrams. In *Proc. of the ICSE 2000 Workshop on Automated Program Analysis, Testing and Verification*, pp. 758-761, June 2000.

[54] A. H. Watson and T. J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric.* NIST Special Publication 500-235, August 1996.

[55] M. Young. Testing Complex Architectural Conformance Relations. In *Proc. Int. Workshop on the ROle of Software Architecture in TEsting and Analysis* (ROSATEA), CNR-NSF, pp. 42-45, July 1998.

Authors Biography

Henry Muccini is an assistant professor at the Computer Science department, University of L'Aquila (Italy) since 2002 and he has been visiting professor at Information & Computer Science, University of California, Irvine. Henry's research interests are in software architecture-based analysis techniques; specifically, in testing of software subsystems integration against a software architecture specification, in product line architecture

specification and testing, in integrating coordination policies inside the architectural description and validating software architectures using semi-formal specifications. Recently, he is also investigating how Web applications may be modeled using UML formalisms and through product line analysis.

Antonia Bertolino is a researcher with the Institute of Information Science and Technologies (ISTI) of the Italian National Research Council (CNR), in Pisa, Italy, where she leads the Software Engineering group and the Pisatel Laboratory. Her research interests are in software engineering, especially software testing and dependability. Currently she investigates approaches for systematic integration test strategies, for architecture and UML-based test approaches, and for component-based software analysis and testing. She is an Associate Editor of the *Journal of Systems and Software*, of *Empirical Software Engineering Journal*, and of the *IEEE Transactions on Software Engineering*. She was the General Chair of the *ACM Symposium on Software Testing and Analysis ISSTA 2002* and of the *2nd Int'l Conf. on Achieving Quality in Software AquIS '93*. She has served in the Program Committees of several conferences and symposia, including *ISSTA*, *Joint ESEC-FSE*, *ICSE*, *SEKE*, *Safecomp*, and *Quality Week*. She has (co)authored over 60 papers in international journals and conferences.

Paola Inverardi is full professor at University of L'Aquila. Previously she has worked at IEI-CNR in Pisa and at Olivetti Research Lab. in Pisa. She is head of the Department of Computer Science at University of L'Aquila, where she leads the Software Engineering and Architecture Research Group. Her main research area is in the application of formal methods to software development. Her research interests primarily concentrates in the field of software architectures. She has actively worked on the verification and analysis of

software architecture properties, both behavioral and quantitative for component-based, distributed and mobile systems. She has served as general chair, program chair and program committee member for many international conferences.

Appendix A

This appendix presents a procedure to derive one or more LTS paths corresponding to a selected ALTS path. The following discussion uses the terminology of Definition II.1: $\mathcal{T}$ identifies the LTS transition rules and $\mathcal{S}$ the LTS states. $\mathcal{D}$ here identifies the ALTS transition rules.

Definition VIII.1: (t_m-derivative of S_i) Let $t_m \in \mathcal{T}$ and S_i, $S_j \in \mathcal{S}$. A t_m-derivative of S_i is a set of states S_j such that S_j is reachable from S_i, applying rules in $\mathcal{T}$, and $S_j \xrightarrow{t_m} S_k$, for any $S_k \in \mathcal{S}$.

In short, t_m-d $(S_i) = \{S_j\text{: } S_i \xrightarrow{t \in \mathcal{T}} S_j \xrightarrow{t_m} S_k \}$.

When we refine ALTS paths, it is important to be able to distinguish those transition rules in the LTS that correspond to arcs in the ALTS. We introduce a notion of validity with respect to a selected ALTS path for an identified LTS sequence of rules.

Definition VIII.2: (Valid t_m-derivative) Let $d_i \in \mathcal{D}$, an ALTS transition rule, and $t_i \in \mathcal{T}$, the corresponding LTS label, that is, $t_i = obs^{-1}(d_i)$. S_j is said to be a valid derivative of S_i (with respect to $\mathcal{D}$) if S_j is a t_m-derivative of S_i, with $S_i \xrightarrow{t \in \mathcal{G}} S_j \xrightarrow{t_m} S_k$, and $\mathcal{G} \subseteq \mathcal{T}$ does not include any rule mapped in $\mathcal{D}$.

In short, Vt_m-d$(S_i) = \{S_j\text{: } S_i \xrightarrow{t \in \mathcal{G}} S_j \xrightarrow{t_m} S_k \}$ and $\mathcal{G} = \{\mathcal{T}\}$ - $\{t_x\text{: } obs(t_x) \in \mathcal{D}\}$.

To check whether this notion of validity holds, simple graph manipulation techniques can be applied, based on the concept of the Reachability Matrix (well known algorithms may be found in [?]).

Definition VIII.3: (Valid path) A valid path between S_i and $S_j \in Vt_m\text{-}d(S_i)$ is every path connecting S_i and S_j.

We can now present the procedure to derive LTS paths conforming to a selected ALTS path.

Procedure RefinePath:

1. Select an ALTS path to be tested: it can be identified by a sequence of ALTS labels, such as, $d_1.d_2. \ldots .d_k$.

2. Applying $obs^{-1}(d_i)$, $\forall$ i $\in$ {1, 2, ..., k}, derive an ordered sequence of transition rules that identifies a partial test sequence over the LTS. It can be written as $t_1.t_2. \ldots .t_p$. The latter means that, starting from S_0, the initial state of the LTS, rule t_1 must be applied first, followed in order by rules $t_2. \ldots .t_k$, until eventually a final state S_n is reached.

3. Set the source state $S_{Source} = S_0$.

4. Set p = S_0 {p will yield an LTS path that refines the ALTS path}.

For i =1, ... ,k Do

5. Set the current rule CR = t_i.

6. Calculate $Vt_i\text{-}d(S_{Source})$, that is, the set of target solutions S_{Target} so that $S_{Source} \xrightarrow{t \in \mathcal{G}} S_{Target} \xrightarrow{CR} S_k$.

7. Derive a valid LTS path q from S_{Source} to "some" of the S_{Target} states, that is, a LTS

path q labeled only with actions in $\mathcal{G}$.

8. Set S_{Source}=$S_{Target'}$, with $S_{Target} \xrightarrow{CR} S_{Target'}$.

9. Put $p = p.q.S_{Source}$, where "." denotes the concatenation operation between paths.

End_For

10. Derive a valid LTS path r from S_{Source} to S_n, where S_n denotes a final state of the LTS that is reachable by applying rule t_k.

11. Set $p = p.r$.

end procedure.

Given one ALTS path, this procedure can be repeated more times, each time deriving one different LTS test sequence. In fact, this procedure includes two points of selection: several valid LTS states can be selected at step 6 and different strategies can be applied at step 7 (and finally at step 10) to derive an LTS path connecting the selected LTS states. These are the decision points that call for the test manager judgment. To keep the number of tests limited, we may chose to apply at step 7 a transition rules coverage criterion. In other words, for each ALTS path, we want derive enough LTS paths to cover as many transitions rules as possible, in a sense trying to consider all possible system behaviors in correspondence with an abstract test sequence.

Appendix B

In Sections IV-A and IV-B, we described how, from an LTS, an Abstract LTS (ALTS) can be extracted. We reached this goal by defining an *obs*-function (based on hiding and relabeling) and applying it to the LTS, thus producing an observed LTS (ObsLTS) from which by minimization the ALTS is generated, as depicted in Figure **??**.

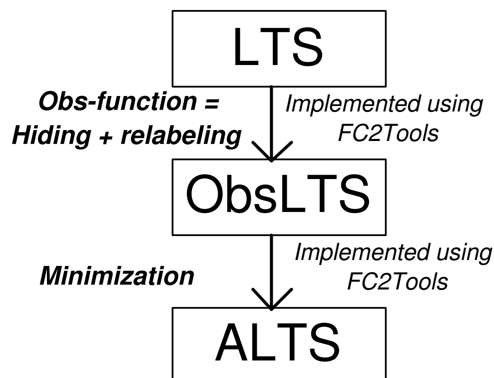

Fig. 6. The abstraction process

What we prove in this appendix is that ALTS paths are suitable abstractions of LTS paths (correctness) and that an ALTS path corresponds to each LTS path (completeness). These properties imply that ALTS paths do not introduce new information with respect to the original one and that all the original information is suitably preserved.

In the following discussion, $obs : \mathcal{L} \longrightarrow \mathcal{L}'$ so that $\mathcal{L} = \mathcal{R} \cup \mathcal{NR}$, $obs(\mathcal{R}) = \mathcal{D}$, and $obs(\mathcal{NR}) = \tau$. We assume that $\mathcal{D} = \mathcal{R}$ to simplify definitions and proofs. Notice, though, that the following proofs are still valid when $\mathcal{D} \neq \mathcal{R}$. With p we identify an LTS path, with p' an ALTS (or ObsLTS) one. With αp ($\alpha p'$), we denote the p (p') alphabet. With $\mathcal{O}$, we denote an *obs*-function.

Correctness and completeness properties may be now formalized as follows:

- *Correctness:*

For each p' there exists at least one p such that $\alpha p \supseteq \alpha p'$, and if $p' = \mathrm{l}_1.\ \mathrm{l}_2.\ \ldots\ .\mathrm{l}_k$ then the label ordering in p induced by p' is preserved.

- *Completeness:*

For each p, there exists a p' so that $\alpha p' \subseteq \alpha p$, and if $p = \mathrm{l}_1.\ \mathrm{l}_2.\ \ldots\ .\mathrm{l}_n$, then the label ordering in p' is consistent with the label ordering in p.

To prove that our abstraction process is correct and complete, we proceed in two steps. Referring to Figure 6, we first prove that the mapping LTS-ObsLTS (step 1) is correct and complete, and then that the ObsLTS-ALTS mapping (step 2) is correct and complete.

Step 1 proofs: $\mathcal{O}$ applies hiding and relabeling. The *hiding* transitional semantics may be formally defined as follows:

$$1)\frac{P \xrightarrow{l} P'}{P{\upharpoonright}\mathcal{R} \xrightarrow{l} P'{\upharpoonright}\mathcal{R}}\ \mathrm{l} \in \mathcal{R},\ 2)\frac{P \xrightarrow{l} P'}{P{\upharpoonright}\mathcal{R} \xrightarrow{\tau} P'{\upharpoonright}\mathcal{R}}\ \mathrm{l} \in \mathcal{NR}$$

in which $\upharpoonright$ represents the hiding operator, $\mathcal{R}$ the set of actions we want to observe, and P is the LTS we are analyzing.

The *relabeling* transitional semantics follows:

$$3)\frac{P \xrightarrow{l} P'}{P/g \xrightarrow{g(l)} P'/g},\ 4)\frac{P \xrightarrow{\tau} P'}{P/g \xrightarrow{\tau} P'/g}$$

in which g : $\mathcal{R} \longrightarrow \mathcal{D}$. These formalizations come from the FSP operators [**?**]. Assuming that $\mathcal{D} = \mathcal{R}$, $\mathcal{O}$ behaves as the hiding operator.

Correctness proof: Let P' be an obsLTS, obtained from P by applying $\mathcal{O}$. Let p' be a generic P' path. p' may be composed only of elements in $\mathcal{R}$ (case1) or it may also contain τ actions (case 2).

In case 1, p' does not contain τ actions. It means that its corresponding p does not contain $\mathcal{NR}$ elements ($\alpha p \subseteq \mathcal{R}$). This implies that $\alpha P = \alpha P'$ and $p = p'$.

Case 2 may present two different scenarios: i) all the τ actions in p' also exist in p. In this case, $p = p'$. Otherwise, ii) some τ actions in p' have been generated by the observation process. In this case, some actions in $\mathcal{L}$ have been hidden by means of $\mathcal{O}$. Thus, p' is the *obs*-image of all those LTS paths p in which the τ label in p' is substituted by a label in $\mathcal{L}$ - $\alpha p'$. For example, let $\mathcal{R}$ = {l_1, l_2,..., l_n}. Then if $p' = \mathrm{l}_1.\mathrm{l}_2.\tau.\mathrm{l}_n$, all the LTS paths p

$= \mathrm{l}_1.\mathrm{l}_2.\mathrm{l}_x.\mathrm{l}_n$ so that $\mathrm{l}_x \in \mathcal{L}$ and $\mathrm{l}_x \notin \mathcal{R}$ generate p' via obs. By construction of P', at least one of these paths must exist in the LTS. Moreover, $\alpha p \supseteq \alpha p'$ (since p includes at least all the p' labels) and the label order is preserved (by construction of P').

Completeness proof: Let p be a generic path in the LTS P. We want to prove there exists an ALTS path p' corresponding to p. Given p and $\mathcal{R}$, there are two cases: 1) $\alpha p \subseteq \mathcal{R}$, and we do not hide anything, thus $p' = p$, or 2) αp contains elements in $\mathcal{NR}$. Applying the hiding rules 1 and 2, labels in $\mathcal{R}$ are preserved in p' whereas those in $\mathcal{NR}$ become τ actions. Recursively applying rules 1 and 2 to the path labels, we obtain the ALTS path p', satisfying the completeness properties.

Step 2 proofs: Let $p \in$ ObsLTS and $p' \in$ ALTS. p and p' are, for construction, trace equivalent. This implies that $\alpha p = \alpha p'$ and that labels in p are ordered in the same way as in p'. It suffices to prove correctness and completeness.

Appendix C: Tools Support

In the presented approach, some tools are used to implement the different steps. Initially, an architectural language is used to specify our software architecture. An LTS model of the SA dynamics is then automatically generated from this specification, and abstraction and minimization are applied over the LTS to build an Abstract LTS. Finally, we instrument the source code to analyze the TRMCS behavior with respect to the architectural tests. Figure 7 summarizes the framework we use:

1. The **Finite State Process (FSP)** [?], [38] process algebra is used to specify software component behaviors.

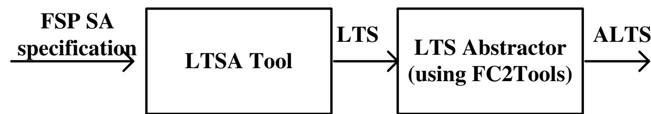

Fig. 7. The framework

2. The **LTSA tool** [34] takes an FSP SA specification and gives the corresponding LTS as a result.

3. The **LTS Abstractor** builds abstracted views of the LTS (based on the previously discussed theory). It has been implemented by using the existing FC2Tools [21].

The FSP 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 comprises a declarative section defining variables and ranges, a section defining the process initial state, and a section describing the other reachable states. Semantically, an FSP process waits for an action (e.g., for receiving messages), performs actions (e.g., for sending messages) and changes its state. The LTS alphabet is composed of the exchanged messages.

Figure 8 shows the FSP specification for the TRMCS system. Figure 8.a defines some ranges used to create multiple instances of processes and Figure 8.b specifies the behavior of the TRMCS components. The User process is separated into two parallel processes (USER_ALARM and USER_CHECK) which are able to send alarms and checks. The Router is separated into three parallel processes (to manage Alarms and Checks and to wait for the Timer). The Server is implemented by two processes, and the Timer com-

ponent sends some clock messages to the Router. For those not familiar with FSP, the USER_ALARM line in Figure 8.b says that the USER_ALARM process sends a message called “SendAlarm_To_Router”, waits for the “ReceiveAck_From_Router” message reception and goes back to the initial state (i.e., USER_ALARM). Figure 8.c is used to put the various processes in parallel, specifying how the LTSs cooperate. This specifies how the TRMCS system behaves, that is, how the User, Router, Server, and Timer processes have to be put in parallel to describe the whole system behavior.

Each FSP process can be described by an LTS model that contains all the states a process may reach and all the transitions it may perform. The LTSA tool supports the FSP language by automatically generating the LTSs of each FSP process. The tool allows graphical and textual visualization of the resulting LTSs, allows an evaluation of process properties (i.e., safety, deadlock, reachability), supports specification animation to facilitate interactive exploration of system behavior and can be used to put different processes in parallel. This last feature allows us to obtain a global LTS model of the system. By running the LTSA tool on the TRMCS FSP specification, we obtained an LTS composed of 256 states.

The observation of the LTS via an abstraction mechanism has been implemented by using the FC2Tool. In particular, we took advantage of a function called “fc2explicit” provided by the tool for comparing two “.FC2” graphs. The first graph is the one we want to abstract (the architectural LTS), and the second one (in the following, *Obs*-graph) is a graph we generate once we know which are the observable and unobservable labels. In practice, it represents the observation. The *Obs*-graph is easily built by following these rules: i) draw an initial state, ii) for each LTS action by act_i we want to observe, create

```
range N = 0..1                                                                   a)
range K = 0..1
range Sent = 0..1
---------------------------------------------------------------------------------------
/************************ User Process  ************************/              b)
USER_ALARM= (sendAlarm_To_Router -> receiveAck_From_Router -> USER_ALARM).
USER_CHECK =  (sendCheck_To_Router -> USER_SENDCHECK).
||USER = (USER_ALARM||USER_CHECK).
/************************ Router Process ************************/
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 ->
        ROUTER_RECEIVETIME|pre_receiveTime-> ROUTER_RECEIVETIME)).
||ROUTER = ([0..1]:ROUTER_RECEIVEALARM||[0..1]:ROUTER_RECEIVECHECK||
        ROUTER_RECEIVETIME).
/****************** Server Process ************************/
SERVER_RECEIVEALARM = (receiveAlarm_From_Router ->  sendAck_To_Router ->
        SERVER_RECEIVEALARM).
SERVER_RECEIVETIME = (receiveNoFunc_From_Router -> SERVER_RECEIVETIME).
/***********************Timer Process ************************/
TIMER = (receiveInput_From_Router -> sendTime_To_Router -> TIMER).
---------------------------------------------------------------------------------------
/*********************** TRMCS ***********************/                        c)

||USER_ROUTER=(u[0..1]:USER||r:ROUTER||sa[0..1]:SERVER_RECEIVEALARM||
        st:SERVER_RECEIVETIME||t:TIMER)/
{
u[0].sendAlarm_To_Router/r.[0].receiveAlarm_From_User,
u[1].sendAlarm_To_Router/r.[1].receiveAlarm_From_User,

r.[0].sendAlarm_To_Server/sa[0].receiveAlarm_From_Router,
r.[1].sendAlarm_To_Server/sa[1].receiveAlarm_From_Router,

sa[0].sendAck_To_Router/r.[0].receiveAck_From_Server,
sa[1].sendAck_To_Router/r.[1].receiveAck_From_Server,

r.[0].sendAck_To_User/u[0].receiveAck_From_Router,
r.[1].sendAck_To_User/u[1].receiveAck_From_Router,

u[0].sendCheck_To_Router/r.[0].receiveCheck_From_User,
u[1].sendCheck_To_Router/r.[1].receiveCheck_From_User,

r.[0].sendInput_To_Timer/t.receiveInput_From_Router,
r.[1].sendInput_To_Timer/t.receiveInput_From_Router,

t.sendTime_To_Router/r.receiveTime_From_Timer,

r.sendNoFunc_To_Server/st.receiveNoFunc_From_Router
}.
```

Fig. 8. TRMCS FSP Specification

a node labeled by act_i and an edge from the initial state labeled act_i; iii) for all the LTS actions $\{\text{act}_j, ..., \text{act}_k\}$ we want to hide, create an edge from the initial state to itself labeled $\{\text{act}_j + ... + \text{act}_k\}$.

By running the "fc2explicit -abstract LTS.fc2 Obsgraph.fc2 > ALTS-nm.fc2" command, we can compare the two graphs and generate a nonminimized ALTS. The "fc2explicit -<opt> ALTS-nm.fc2 > ALTS.fc2" command generates the minimized graph. Option

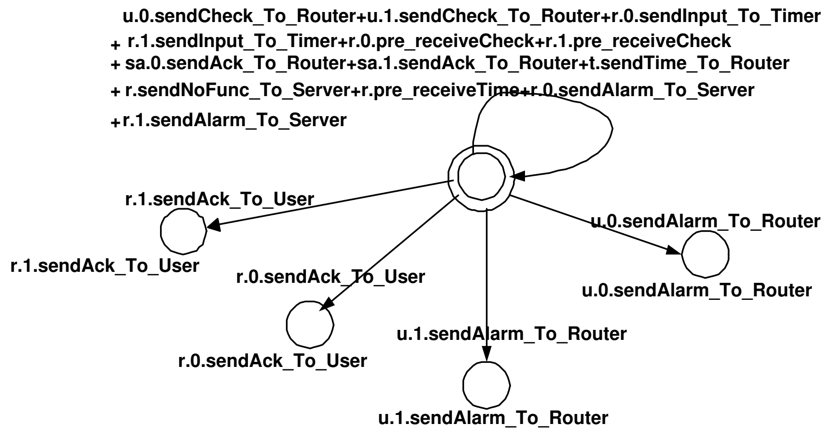

Fig. 9. The Obs-graph for the Alarm ALTS

"opt" allows for selecting different equivalences. The *Obs*-graph in Figure 9 has been used to generate the AlarmObs ALTS in Figure 3.a.