Deriving Test Plans from Architectural Descriptions

A. Bertolino
Istituto di Elaborazione dell'Informazione
C.N.R. - Pisa
Via S. Maria, 46
56100 Pisa, Italy
+39 050 593478
bertolino@iei.pi.cnr.it

F. Corradini, P. Inverardi & H. Muccini
Dipartimento di Matematica
Universitá di L'Aquila
Via Vetoio, 1
67100 L'Aquila, Italy
+39 0862 433127
{flavio, inverard, muccini}@univaq.it

ABSTRACT
The paper presents an approach for deriving test plans for the conformance testing of a system implementation with respect to the formal description of its Software Architecture (SA). The SA describes a system in terms of its components and connections, therefore the derived test plans address the integration testing phase. We base our approach on a Labelled Transition System (LTS) modeling the SA dynamics, and on suitable abstractions of it, the Abstract Labelled Transition Systems (ALTSs). ALTSs offer specific views of the SA dynamics by concentrating on relevant features and abstracting away from uninteresting ones.

ALTS is a tool we provide to the software architect that lets him/her focus on relevant behavioral patterns and more easily identify those that are meaningful for validation purposes. Intuitively, deriving an adequate set of functional test classes means deriving a set of paths appropriately covering the ALTS. In the paper we describe our approach in the scope of a real-world case study and discuss in detail all the steps of our methodology, from ALTS identification to test plan generation.

Keywords
Functional Test Plans, Integration Testing, Labelled Transition Systems, Software Architectures.

1 INTRODUCTION

In recent years the focus of software engineering is continuosly moving towards systems of larger dimensions and complexity. Software production is becoming more and more involved with distributed applications running on heterogeneous networks, while emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality [21]. As a result, applications are increasingly being designed as sets of autonomous, decoupled components, promoting faster and cheaper system development based on commercial off-the-shelf integration, and facilitating architectural changes required to cope with the dynamics of the underlying environment.

The development of these systems poses new challenges and exacerbates old ones. A critical problem is understanding if system components integrate correctly. To this respect the most relevant issue concerns *dynamic integration*. Indeed, component integration can result in architectural mismatches when trying to assemble components with incompatible interaction behavior [10, 5], leading to system deadlocks, livelocks or in general failure to satisfy desired functional and non-funtional system properties.

In this context *Software Architecture* (SA) can play a significant role. SAs have in the last years been considered, both by academia and software industries, as a means to improve the dependability of large complex software products, while reducing development times and costs [20, 1]. SA represents the most promising approach to tackle the problem of scaling up in software engineering, because, through suitable abstractions, it provides the way to make large applications manageable. The originality of the SA approach is to focus on the overall organization of a large software system (the glue) using abstractions of individual components. This approach makes it possible to design and apply tractable methods for the development, analysis, validation, and maintenance of large software systems.

A crucial part of the development process is *testing*. While new models and methods have been proposed with respect to requirements analysis and design, notably the Unified Modeling Language (UML) [17], scarce attention has been devoted so far to the testing of these kinds of systems. The paradox is that these new approaches specifically address the design of large scale software systems. However, for such systems, the testing problems not only do not diminish, but are intensified. This is especially true for integration testing. In fact, due to the new paradigms centered on component-based assembly of systems, we can easily suppose a software process in which unit testing plays

ICSE 2000 Limerick Ireland

a minor role, and testers have to focus more and more on how components work when plugged together.

Lot of work has been devoted to the analysis of formal descriptions of SAs. Our concern is not in the analysis of the consistency and correctness of the SA, but rather in exploiting the information described at the SA level to drive the testing of the implementation. In other words, we assume the SA description is correct and investigate approaches to specification-based integration testing, whereby the reference model used to generate the test cases is the SA description.

In general, deriving a functional test plan means to identify those classes of behavior that are relevant for testing purposes. A functional equivalence class collects all those system executions that, although different in details, carry on the same informative contents for functional verification. The tester's expectation/hope is that any test execution among those belonging to a class would be equally likely to expose possible non conformities to the specification.

We identify interesting test classes for SA-based testing as sequences of interactions between SA components. More precisely, starting from an architectural description, we first derive a *Labelled Transition System* (LTS), that graphically describes the SA dynamics. The problem is that the LTS provides a global, monolithic description of the set of all possible behaviors of the system. It is a tremendous amount of information flattened into a graph. It is quite hard for the software architect to single out from this global model relevant observations of system behavior that would be useful during validation.

We provide the software architect with a key to decipher the LTS dynamic model: the key is to use *abstract views* of the LTS, called ALTSs, on which he/she can easily visualize relevant behavioral patterns and identify those that are more meaningful for validation purposes. Test classes in our approach correspond to ALTS paths. However, once test class selection has been made, it is necessary to return to the LTS and retrieve the information that was hidden in the abstraction step, in order to identify LTS paths that are appropriate refinements of the selected ALTS paths. This is also supported by our approach.

In the following we describe in detail the various steps of the proposed approach in the scope of a case study. In Section 2 we provide the background information: we recall the Cham formalism, that is used here for SA specification, and outline the case study used as a working example. In Section 3 we provide a general overview of the approach. In Section 4, we give examples of using the approach, and address more specific issues. In Section 5, we clarify better the relation between ALTS paths and test specifications. Finally, in the Conclusions, we summarize the paper contribution and address related work.

2 REPRESENTING SA DYNAMICS

A key feature of SA descriptions is their ability to specify the dynamics. Finite State Machines, Petri Nets or Labelled Transition Systems (LTSs) can be used to model the set of all possible SA behaviors as a whole.

In the following subsection we briefly recall the Cham description of SA. From this description we derive an LTS which represents the (global) system behavior of a concurrent, multi-user software system.

The Cham Model

The Cham formalism was developed by Berry and Boudol in the field of theoretical computer science for the principal purpose of defining a generic computational framework [2].

Molecules $m_1, m_2, \ldots$ constitute the basic elements of a Cham, while *solutions* $S_0, S_1, \ldots$ are multisets of molecules interpreted as defining the *states* of a Cham. A Cham specification contains *transformation rules* $T_1, T_2, \ldots$ that define a *transformation relation* $S_i \longrightarrow S_j$ dictating the way solutions can evolve (i.e., states can change) in the Cham. Following the chemical metaphor, the term *reaction rule* is used interchangeably with the term *transformation rule*. In the following, with abuse of notation, we will identify with $\mathcal{R}$ both the set of rules and the set of corresponding labels.

The way Cham descriptions can model SAs has already been introduced elsewhere [12]. Here we only summarize the relevant notions. We structure Cham specifications of a system into four parts:

1. a description of the syntax by which components of the system (i.e., the molecules) can be represented;
2. a solution representing the initial state of the system;
3. a set of reaction rules describing how the components interact to achieve the dynamic behavior of the system.
4. a set of solutions representing the intended final states of the system.

The syntactic description of the components is given by a syntax by which molecules can be built. Following Perry and Wolf [16], we distinguish three classes of components: data elements, processing elements, and connecting elements. The *data elements* contain the information that is used and transformed by the *processing elements*. The *connecting elements* are the "glue"

that holds the different pieces of the architecture together. For example, the elements involved in effecting communication among components are considered connecting elements. This classification is reflected in the syntax, as appropriate.

The initial solution corresponds to the initial, static configuration of the system. We require the initial solution to contain molecules modeling the initial state of each component. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration. One can take advantage of this operational flavor to derive an LTS out of a Cham description. In this paper we will not describe how an LTS can be derived (see [9]). We only recall the LTS definition we will rely on.

Definition 2.1 A Labelled Transition System is a quintuple $(\mathcal{S}, \mathcal{L}, S_0, \mathcal{S}_\mathcal{F}, \mathcal{T})$, where $\mathcal{S}$ is the set of states, $\mathcal{L}$ is the set of labels, $S_0 \in \mathcal{S}$ is the initial state, $\mathcal{S}_\mathcal{F} \subseteq \mathcal{S}$ 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.

Each state in the LTS corresponds to a solution, therefore it is made of a set of molecules describing the states of components. Labels on LTS arcs denote the transformation rule that lets the system move from the tail node state to the head node state.

We also need the definition of a *complete path*:

Definition 2.2 Let

$$p = S_0 \xrightarrow{l_1} S_1 \xrightarrow{l_2} S_2 \xrightarrow{l_3} \ldots \xrightarrow{l_n} S_n$$

be a path in an LTS. p is complete if S_0 is the initial solution and S_n is a final one.

Although our approach builds on the Cham description of a SA it is worthwhile stressing that it is not committed to it. Our choice of the Cham formalism is dictated by our background and by its use in previous case studies. We are perfectly aware that other choices could be made and want to make clear that the use of a specific formalism is not central to our approach.

In more general terms what we have done so far can be summarized as follows: we have assumed the existence of an SA description in some Architectural Description Language (ADL) and that from such description an LTS can be derived, whose node and arc labels represent respectively states and transitions relevant in the context of the SA dynamics. We also assume that states contain information about the single state of components and that labels on arcs denote relevant system state transitions.

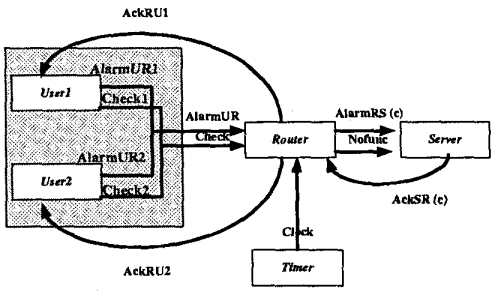

Figure 1: Processes and Channels

The TRMCS Case-Study

The Teleservice and Remote Medical Care System (TRMCS) provides monitoring and assistance to users with specific needs, like disabled or elderly people. The TRMCS is being 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.

Figure 1 shows the static TRMCS Software Architectural description, in terms of Component and Connectors. Boxes represent Components, i.e., processing elements, arrows identify Connectors, i.e., connecting elements (in this case channels) and arrows labels refer to the data elements exchanged through the channels.

Figure 2 shows the reaction rules of the TRMCS Cham Specification. Rules from T_0 to T_4 represent system startup, rules T_5 and T_6 enable the two users to send an alarm message, rules from T_7 to T_{10} handle the check messages, from T_{11} to T_{18} handle the alarm messages, from T_{19} to T_{21} finally manage the Timer component.

A portion of the LTS of the TRMCS SA is given in Fig. 3. The whole LTS is around 500 states. Note that arc labels 0, 1, ..., 21 correspond, respectively, to T_0, T_1, ..., T_{21} (the labels of the TRMCS reaction rules),"0"

Reaction Rules

T0: User = User1, User2
T1: User1= User1.o(check1), User1.o(alarmUR1).i(ackRU1)
T2: User2= User2.o(check2), User2.o(alarmUR2).i(ackRU2)
T3: User1.o(check1)= o(check1).User1
T4: User2.o(check2)= o(check2).User2
T5: User1.o(alarmUR1).i(ackRU1) = o(alarmUR1).i(ackRU1).User1
T6: User2.o(alarmUR2).i(ackRU2) = o(alarmUR2).i(ackRU2).User2
T7: o(check1).User1, i(check).Router, NoSent = User1.o(check1) , i(check).Router, Sent
T8: o(check1).User1, i(check).Router, Sent = User1.o(check1) , i(check).Router, Sent
T9: o(check2).User2, i(check).Router, NoSent = User2.o(check2) , i(check).Router, Sent
T10: o(check2).User2, i(check).Router, Sent = User2.o(check2) , i(check).Router, Sent
T11: o(alarmUR1).i(ackRU1).User1, i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router
= i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router,o(alarmRS1).i(ackSR1).o(ackRU1).Router, i(ackRU1).User1.o(alarmUR1)
T12: o(alarmUR2).i(ackRU2).User2, i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router
= i(alarmUR).o(alarmRS).i(ackSR).o(ackRU).Router,o(alarmRS2).i(ackSR2).o(ackRU2).Router, i(ackRU2).User2.o(alarmUR2)
T13: o(alarmRS1).i(ackSR1).o(ackRU1).Router , i(alarmRS).o(ackSR).Server
= i(ackSR1).o(ackRU1).Router , i(alarmRS).o(ackSR).Server, o(ackSR1).Server
T14: o(alarmRS2).i(ackSR2).o(ackRU2).Router , i(alarmRS).o(ackSR).Server
= i(ackSR2).o(ackRU2).Router , i(alarmRS).o(ackSR).Server, o(ackSR2).Server
T15: o(ackSR1).Server, i(ackSR1).o(ackRU1).Router = o(ackRU1).Router
T16: o(ackSR2).Server, i(ackSR2).o(ackRU2).Router = o(ackRU2).Router
T17: o(ackRU1).Router , i(ackRU1).User1.o(alarmUR1) = User1.o(alarmUR1).i(ackRU1)
T18: o(ackRU2).Router , i(ackRU2).User2.o(alarmUR2) = User2.o(alarmUR2).i(ackRU2)
T19: m1.Router, Timer, Sent = o(nofunc).Router, m1.Router, NoSent
T20: m1.Router, Timer, Sent = m1.Router, Timer, NoSent
T21: o(nofunc).Router, i(nofunc).Server = i(nofunc).Server, Timer

Figure 2: TRMCS Cham Reaction Rules

denotes the initial state and box states denote pointers to states elsewhere shown in the picture (to make the graph more readable). Double arrows denote the points in which the figure cuts LTS paths.

3 AN APPROACH TO SA-BASED TESTING

In this section we introduce our approach to SA-based testing. Our goal is to use the SA specification as a reference model to test the implemented system. Needless to say, there exists no such thing as an *ideal* test plan to accomplish this goal. It is clear, on the contrary, that from the high-level, architectural description of a system, several different SA-based test plans could be derived, each one addressing the validation of a specific functional aspect of the system, and different interaction schemes between components.

Therefore, what we assume as the starting point for our approach is that the software architect, by looking at the SA from different viewpoints, chooses a set of important patterns of behavior to be submitted to testing. This choice will be obviously driven by several factors, including specificity of the application field, criticality, schedule constraints and cost, and is likely the most crucial step to a good test plan (we give examples of some possible choices in Section 4).

With some abuse of terminology, we will refer to each of the selected patterns of behavior as to an *SA testing criterion*. With this term we want to stress that our approach will then derive a different, specific set of tests so as to fulfill the functional requirements that each "criterion" (choice) implies.

Two remarks are worth noting here. One is that as the derived tests are specifically aimed at validating the high-level interactions between SA components, the test plans we develop apply to the integration test stage. The second remark is that since we are concerned with testing (i.e., with verifying the software in execution), we will greatly base our approach on the SA dynamics. In particular, starting from a selected SA testing criterion, we will primarily work on the SA LTS and on other graphs derived from the latter by means of abstraction (as described in the following).

Introducing *obs*-functions over SA dynamics

An SA testing criterion is initially derived by the software architect in informal terms. We want to translate it in a form that is interpretable within the context of the SA specification, in order to allow for automatic processing.

Intuitively, an SA testing criterion abstracts away uninteresting interactions. Referring to the Cham formalism, an SA testing criterion naturally partitions the Cham reaction rules 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). This suggests to define an interpretation domain $\mathcal{D}$, to which the relevant transformation rules (i.e., the arc labels of the LTS) are mapped, and a distinct element τ,

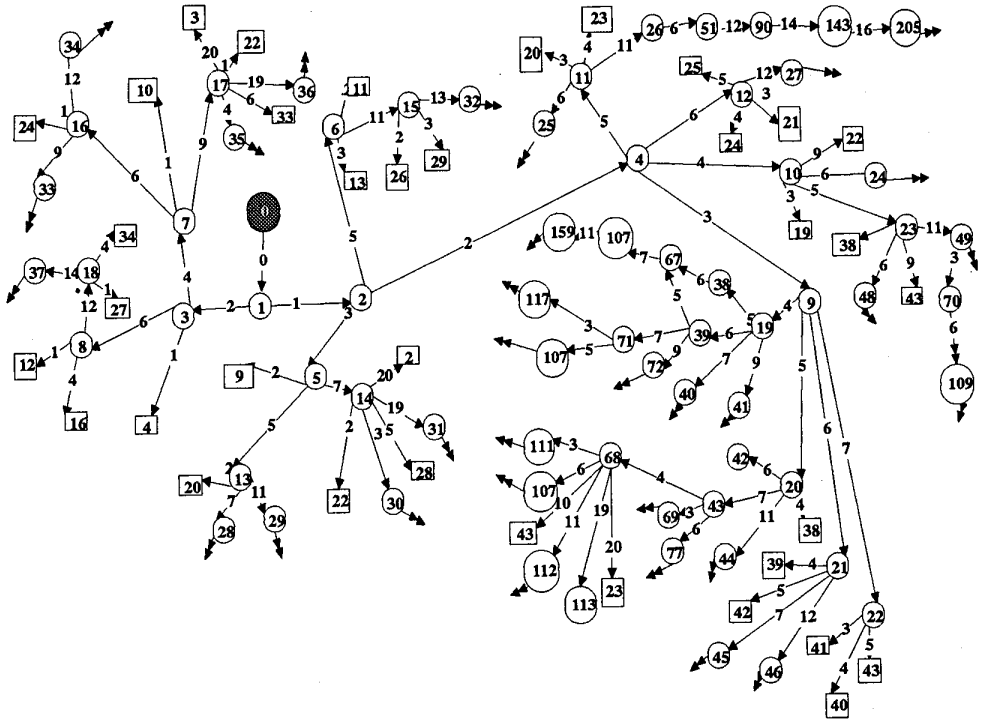

Figure 3: A portion of the TRMCS LTS

to which any other (not relevant) rule is mapped.

We therefore associate with an SA testing criterion an *obs*-function. This is a function that maps the relevant reaction rules of the Cham SA description to a particular domain of interest $\mathcal{D}$. More precisely, we have:

$obs : \mathcal{R} \longrightarrow \mathcal{D} \cup \{\tau\}$.

The idea underlying the set $\mathcal{D}$ is that it expresses a semantic view of the effect of the transition rules on the system global state.

From LTS to ALTSs

We use the *obs*-function just defined as a means to derive from the LTS an automaton still expressing all high level behaviors we want to test according to the selected SA testing criterion, but hiding any other unrelevant behaviors. The automaton is called an ALTS (for *Abstract* LTS).

This is the LTS that is obtained by relabelling, according to the function *obs*, each transition in $\mathcal{R}(S_0)$, and by minimizing the resulting automaton with respect to a selected equivalence (*trace*- or *bisimulation*-based equivalence), preserving desired system properties (as discussed in [3]).

If we derive a complete path over an ALTS (see Definition 2.2), this quite naturally corresponds to the high level specification of a test class of the SA (we give examples in the next section). Therefore, the task of deriving an adequate set of tests according to a selected SA testing criterion is converted to the task of deriving a set of complete paths appropriately covering the ALTS associated with the criterion via an *obs*-function.

In an attempt of depicting a general overview of the approach, we have so far deliberately left unresolved some concrete issues. Most importantly, what does it mean to look at the SA from a selected observation point, i.e., which are meaningful *obs*-functions? And, also, once an ALTS has been derived, how are paths on it selected? Which coverage criterion could be applied? We will devote the next section to answer these questions, with the help of some examples regarding the TRMCS case study.

4 APPLYING SA-BASED TESTING TO THE TRMCS CASE-STUDY

Considering the informal description of the TRMCS in Section 2, because of obvious safety-critical concerns, we may want to test the way an Alarm message flows in the system, from the moment a User sends it to the moment the User receives an acknowledgement. Casting this in the terms used in the previous section, the software architect may decide that an important SA testing criterion is "all those behaviors involving the flow of an

D = {SendAlarm1, SendAlarm2, ReceiveAck1, ReceiveAck2}
obs (T_{11}) = SendAlarm1 : *User1 issues an Alarm msg* obs (T_{17}) = ReceiveAck1 : *User1 receives an Ack* obs (T_{12}) = SendAlarm2 : *User2 issues an Alarm msg* obs (T_{18}) = ReceiveAck2 : *User2 receives an Ack*
For any other T_i, obs (T_i) = tau

Figure 4: Alarm flow: Obs-function

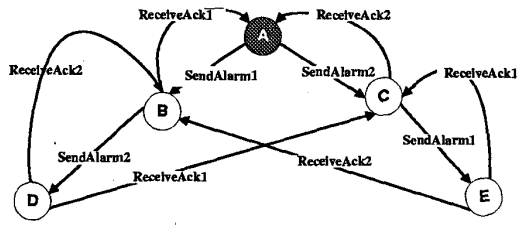

Figure 5: Alarm flow: ALTS

Alarm message through the system".

From this quite informal specification, a corresponding *obs*-function could be formally defined as in Figure 4. As shown, we have included in the interpretation domain $\mathcal{D}$ all and only the Cham transition rules that specifically involve the sending of an Alarm message by a User, or the User's reception of an acknowledgement of the Alarm message from the Router. Note that this information is encoded, at the LTS level, in the arcs labels.

With reference to this *obs*-function, and applying reduction and minimization algorithms (in this case we have minimized with respect to trace equivalence), we have derived the ALTS depicted in Figure 5 (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)).

Note the rather intuitive appeal of such a small graph with regard to the (much more complex) complete LTS (for the TRMCS it is one hundred times bigger). One could be tempted to consider some rather thorough coverage criterion of the ALTS, such as taking all complete paths derivable by fixing a maximum number of cycles iterations. However, as we will see better in the next section, each ALTS path actually will correspond to many concrete test cases. Therefore, less thorough coverage

D = {SendCheck1_1, SendCheck1_2, SendCheck2_1, SendCheck2_2, CheckOk, CheckERR}
obs (T_7) = SendCheck1_1 : *User1 sends the first Check msg* obs (T_8) = SendCheck1_2 : *User1 sends a further Check msg* obs (T_9) = SendCheck2_1 : *User2 sends the first Check msg* obs (T_{10}) = SendCheck2_2 : *User2 sends a further Check msg* obs (T_{20}) = CheckOk : *every User has sent a Check msg* obs (T_{21}) = CheckERR : *some User has not sent a Check msg*
For any other T_i, obs (T_i) = tau

Figure 6: Check flow: Obs-function

criteria seem more practical. In particular, we found that McCabe's technique of selecting all basic paths [14] offers here a good compromise between arc and path coverage. A list of ALTS test paths derived according to McCabe's technique is the following:

Path1: A B A
Path2: A B D B A
Path3: A B A C A
Path4: A B D C A
Path5: A B D C E C A
Path6: A B D C E B A

Let us consider, for example, Paths No. 2, 3 and 4. These three paths are all devoted to verify that the system correctly handles the consecutive reception of two Alarm messages issued by two distinct Users. By putting these three ALTS paths in the list, we explicitly want to distinguish in the test plan the cases that: i) each Ack message is sent rightly after the reception of the respective Alarm message (Path3); or, the acknowledgements are sent after both Alarms are received and ii) in the same order of Alarm receptions (Path4); or finally iii) in the opposite order (Path2). So the three test classes are aimed at validating that no Alarm message in a series of two is lost, whichever is the order they are processed in.

Still considering the TRMCS, the software architect could decide that also the Check flow is worth testing. Thus, analogously to what we have done for the Alarm flow, the Check flow *obs*-function is derived in Figure 6 and the corresponding ALTS is depicted in Figure 7. It represents a different "observation" of the TRMCS behavior.

We have reasoned so far in the hyphotetical scenario of the TRMCS system being developed and of a software architect that is deriving interesting architectural behaviors to be tested. An alternative scenario could be that the TRMCS is already functioning, and that one of the components is being modified.

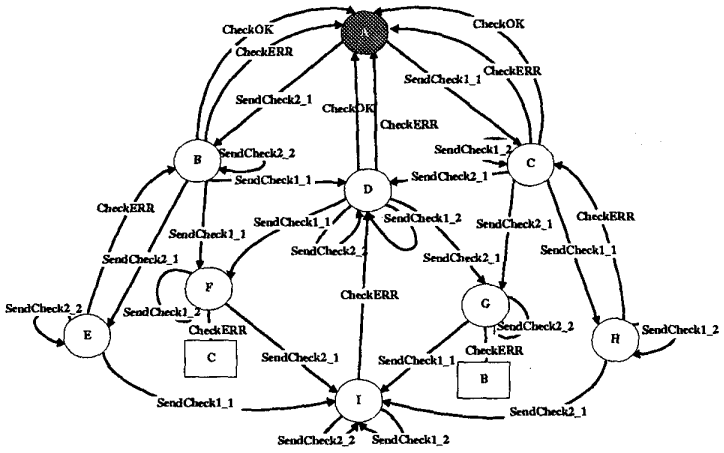

Figure 7: Check flow: ALTS

We want then to test whether the modified component still interacts with the rest of the system in conformance to the SA original description. In this case, the observation point of the software architect will be "all the interactions that involve this component". If, specifically, the component being modified is the Server, then the corresponding *obs*-function is given in Figure 8 and the resulting ALTS in Figure 9. McCabe's coverage criterion yields the following set of test classes:

Path1: A A
Path2: A B A
Path3: A B B A
Path4: A B D B A
Path5: A B A C A
Path6: A B D C A
Path7: A B D D B A
Path8: A B D C C A
Path9: A B D C D B A

This example evidences that even in deriving the basic ALTS paths we do not blindly apply a coverage criterion, but somehow exploit the semantics behind the elements in $\mathcal{D}$. For instance, consider Path5 above. If we interpret it in light of McCabe's coverage criterion, it is aimed at covering transition **FRa2** from State A to State C. The shorter path A C A would be equally good for this purpose. But for functional testing this shorter path is useless, because it would be perfectly equivalent to the already taken Path2 (A B A): both paths in fact test the forwarding of one Alarm message to the Server. Therefore, to cover the transition from A to C we have instead selected the longer path A B A C A that serves the purpose to test the consecutive forwarding of two Alarms.

D = {FRa1, FRa2, TRack1, TRack2, FRno}
obs (T_{13}) = FRa1 : *Alarm1 msg From Router* obs (T_{14}) = FRa2 : *Alarm2 msg From Router* obs (T_{15}) = TRack1 : *Ack1 msg To Router* obs (T_{16}) = TRack2 : *Ack2 msg To Router* obs (T_{21}) = FRno : *NoFunction msg From Router*
For any other T_i, obs (T_i) = tau

Figure 8: Component Based: Obs-function

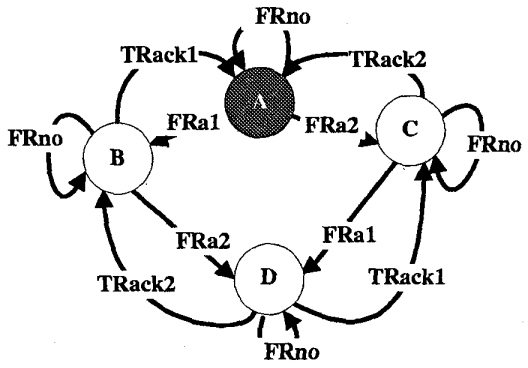

Figure 9: Component Based: ALTS

5 FROM ALTS PATHS TO TEST SPECIFICATION

ALTS paths specify functional test classes at a high abstraction level. One ALTS path will generally correspond to many concrete test cases (i.e., test executions at the level of the implemented system).

It is well-known that several problems make the testing of concurrent systems much more difficult and expensive than that of sequential systems (for reasons of space we do not discuss these problems in depth here; see, e.g., [7]). Said simply, a trade-off can be imagined in general between how tightly is the test specification of an event sequence given, and how much effort will be needed by the tester to force the execution of that sequence. The point is that the tester, on receiving the high level test specifications corresponding to ALTS paths, could choose among many concrete test executions that conform to them. For example, considering Path2: A B D B A on Fig. 5, a possible test execution can include the sending of an Alarm message from User1 immediately followed by the sending of an Alarm message from User2; another test execution could as well include, between the two Alarm messages, the Router reception of other messages, e.g., a Check or a Clock, and still conform to the given high level test specification.

This flexibility in refining test specifications descends

from the fact that to derive the ALTS from the complete LTS we have deliberately abstracted away transitions not involving the Alarm flow from and to a User. However after an ALTS-based list of paths has been chosen, we can go back to the complete LTS and observe what the selected abstraction is hiding, i.e., we can precisely see on the SA LTS which are the *equivalence assumptions*[1] behind ALTS paths (test classes) selection.

This is a quite attractive feature of our approach for SA-based test class selection. When functional test classes are derived ad-hoc (manually), as is often the case for the high level test stages, equivalence assumptions those test classes rely upon remain implicit, and are hardly recoverable from the system specification. In our approach, first an explicit abstraction step is required (ALTS derivation). Second, going back from the ALTS to the complete LTS, we can identify which and how many LTS paths fulfill a given ALTS path.

We can better explain this by means of an example. Considering the ALTS for the Alarm flow (Fig. 5), State B is equivalent (under the test assumptions made) roughly to forty states in the complete LTS (of course, we can automatically identify all of them). Not only, but there are more valid LTS subpaths that we could traverse to reach each of these forty states. The valid subpaths for this example are all those going from the initial state S_0 of the LTS to any of the forty states equivalent to state B of the ALTS, without including any of the transformation rules in the domain $\mathcal{D}$ defined for the Alarm flow *obs*-function, except for the last arc that must correspond to the transformation rule T_{11}. All of these (many) subpaths would constitute a valid refinement of the abstract SendAlarm1 transition in Path2.

As such a refinement should be applied to each state and each arc of the ALTS paths, it is evident then how the number of potential LTS paths for one ALTS path soon becomes huge. We cannot realistically plan test cases for all of them; so the pragmatic question is: how do we select meaningful LTS paths (among all those many refining a same ALTS path)? We don't believe that a completely automatic tool (i.e., a smart graph processing algorithm) could make a good choice. What we prospect, rather, is that the software architect, with the indispensable support of appropriate graphical tools processing aids, can exploit his/her semantic knowledge of the SA dynamics to discern between LTS paths that are equivalent with respect to an ALTS abstraction. In

[1]The term "equivalence" here refers to its usual meaning in the testing literature, i.e., it denotes test executions that are interchangeable with respect to a given functional or structural test criterion, and not to the more specific trace/bisimulation equivalence used so far for graph minimization.

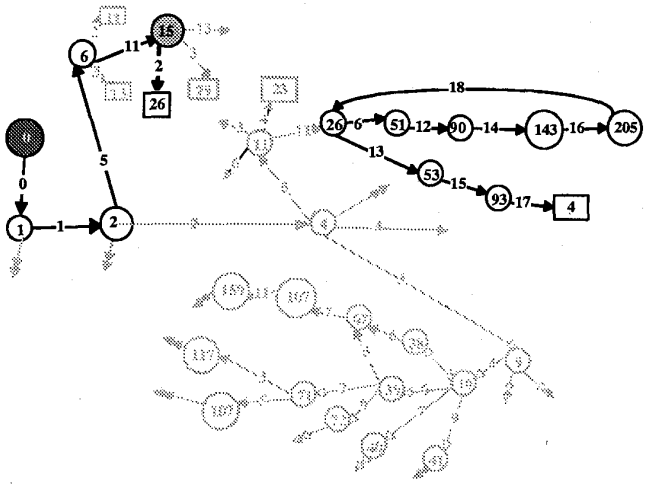

Figure 10: An LTS test path

this perspective, we are currently investigating different criteria. One straightforward approach is structural criteria, variously based on path length and on limiting the number of loop interactions within paths. Another interesting possibility could be to also consider prioritizing some nodes, in the construction of paths, because of their operational profile, or because of their relevance in an ALTS other than the current one. On the other hand, the software architect will also take into account other important factors, not captured in the SA description, such as safety-critical requirements, or time and cost constraints. Thus we finally expect that the software architect produces from the list of ALTS paths a refined list of LTS paths.

In Fig. 10, for instance, we show an LTS path that is a valid refinement of Path2 for the Alarm flow ALTS (uninteresting LTS arcs and nodes are marked in light grey). This example is the shortest path we could take to instantiate the ALTS path, in that it only includes indispensable TRMCS transformation rules to fulfill the path. We precise that S_{15} in particular is the state equivalent to State B of the Alarm flow ALTS (note in fact that the entering arc is labelled 11). Another of the forty LTS states equivalent to B is State S_{159} (see Fig. 3). There is a semantic difference between S_{15} and S_{159} that could be relevant for integration testing purposes. Before reaching S_{159} (i.e., before User1 sends an Alarm), User2 can send a Check message, while this is never possible for any of the LTS subpaths reaching S_{15}. We could see this from analyzing the state information that is associated with LTS nodes. In the refinement of Path2, the software architect could then decide to pick one LTS path that includes S_{159} in order to test that a Check from another user does not interfere with an Alarm from a certain user.

6 CONCLUSIONS

The contribution of this paper consists of an approach to the use of the architectural description of a system to define test plans for the integration testing phase of the system implementation. The approach starts from a correct architectural description and relies on a labelled transition system representation of the architecture dynamics.

In summary, the proposed approach consists of the following steps:

1. the software architect selects some interesting SA testing criteria;
2. each SA testing criterion is translated into an *obs*-function; in some case, a criterion could also identify several related *obs*-functions;
3. for each *obs*-function, an ALTS is (automatically) derived from the global LTS corresponding to the SA specification;
4. on each derived ALTS, a set of coverage paths is generated according to a selected coverage criterion. Each path over the ALTS corresponds to the high-level specification of a test class;
5. for each ALTS path, the software architect, by tool supported inspection of the LTS, derives one or more appropriate LTS paths, that specify more refined transition sequences at the architectural level.

Our approach allows the software architect to move across abstractions in order to get confidence in his/her choices and to better select more and more refined test plans. It is worth noticing that in our approach a test plan is a path, that is not only a sequence of events (the labels on a path), but also a set of states, which describes the state of the system in terms of the single state components. This is a much more informative test plan with respect to the one that could be derived from e.g., the requirements specifications. In fact, using the SA LTS, we also provide the tester with information about state components that can be used to constrain the system to exercise that given path.

Related Work

In conventional software systems, integration testing is generally approached by trying to systematically exercise the system subunits in an incremental fashion, following a bottom-up or top-down order, or a variously mixed strategy. Such approaches are clearly no longer adequate for modern systems, where a structural hierarchy cannot be identified, and the pervasiveness of characteristics such as concurrency and distribution make the integration testing task more difficult.

Lot of work has been devoted to testing concurrent and real-time systems, both specification driven and implementation based [7, 13, 6]. We do not have room here to carry out a comprehensive survey; we will just outline some main differences with our approach. These works addressed different aspects, from modelling time to internal nondeterminism, but all focus on unit testing, that is they either view the concurrent system as a whole or specifically look at the problem of testing a single component when inserted in a given environment. Our aim is different, we want to derive test plans for integration testing. Thus although the technical tools some of these approaches use are obviously the same of ours (e.g., LTS, abstractions, event sequences), their use in our context is different. This goal difference emerges from the very beginning of our approach: we work on an architectural description that drives our selection of the abstraction (i.e., the testing criterion) and of the paths (i.e., the actual test classes).

In a related research work [15], a dedicated specification formalism is being defined, called *Information Space*, in parallel with the development of a method for the automatic derivation of integration test steps, deploying a notion of slicing over the system specification. Although using different approaches and terminologies, this research and ours address the same goal, and in fact they can be considered as parties of a common project.

Our approach of defining ALTS paths for specifying high level test classes has lot in common with Carver and Tai's use of *Sequencing Constraints* for specification-based testing of concurrent programs [7]. Indeed, sequencing constraints specify restrictions to apply on the possible event sequences of a concurrent program when selecting tests, very similarly to what ALTS paths do for a SA. In fact, we are currently working towards incorporating within our framework Carver and Tai's technique of *deterministic testing* for forcing the execution of the event sequences (refined LTS paths) produced with our approach.

As far as architectural testing is concerned, the topic has raised interest and received a good deal of attention in recent years [18, 4, 19]. Our approach indeed stems from this ground.

The Future

Our aim is to achieve a usable set of tools that would provide the necessary support to our approach. So far we have experimented with our approach the described case study of which a running Java prototype exists. The way we did it was not completely automatically supported with respect to the ALTS definitions, the criterion identification and path selection. As tool support we could rely on a LTS generator starting from the Cham description, which also allows for keeping track

of the state and arc labels. Work is ongoing to generalize it to ALTS generation and to implement a graphical front-end for Cham descriptions. We definitely believe that the success of such an approach heavily depends on the availability of simple and appealing supporting tools. Our effort goes in two directions, on one side we are investing on automating our approach and we would also like to take advantage of other existing environments and possibly integrate with them, e.g. [11, 8], on the other we are involved in more experimentation. The latter is not an easy job. Experimenting our approach requires the existence of a correct architectural description and a running implementation. The case study presented here could be carried out since the project was entirely managed under our control, from the requirements specification to the coding. This is obviously not often the case. The results we got so far are quite satisfactory and there are other real world case studies we are working on at the moment. For them we have already a running implementation and we have been asked to give a model of their architectural structure. We are confident these will provide other interesting insights to validate our approach.

REFERENCES

[1] Bass, L., Clements, P., Kazman, R. *Software Architecture in Practice.* SEI Series (1998), Addison-Wesley.

[2] Berry, G., Boudol, G. The Chemical Abstract Machine. *Theoretical Computer Science* 96, (1992), 217-248.

[3] Bertolino, A., Corradini, F., Inverardi, P., Muccini, H. Architectural Abstractions to Support Analysis and Testing. *IR 50/99*, University of L'Aquila, Online at <http://univaq.it/~inverard>.

[4] Bertolino, A., Inverardi, P., Muccini, H., Rosetti, A. An Approach to Integration Testing Based on Architectural Descriptions. *IEEE Proc. ICECCS-97* (Como, 1997).

[5] Boehm, B., Abts, C. Cots integration: Plug and pray? *IEEE Computer* 32, 1 (January 1999).

[6] Cardell-Oliver, R., Glover, T. A Practical and Complete Algorithm for Testing Real-Time Systems. *Springer Verlag LNCS 1486*, FTRTFTS98 (September 98), On-line at <http://cswww.essex.ac.uk/>.

[7] Carver, R.H., Tai, K.-C. Use of Sequencing Constraints for Specification-Based Testing of Concurrent Programs. *IEEE Trans. on Software Engineering* 24, 6 (June 1998).

[8] Cleaveland, R., Parrow, J., Steffen, B. The Concurrency Workbench. *ACM Toplas* 15, 1 (1993), 36-72.

[9] Compare, D., Inverardi, P., Wolf, A.L. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming* 33, 2 (1999), 101-131.

[10] Garlan, D., Allen, R., Ockerbloom, J. Architectural mismatch: Why reuse is so hard. *IEEE Software* 12, 6 (November 1995).

[11] Giannakopoulou, D., Magee, J., Kramer, J. The TRACTA Project, On-line at <http://www-dse.doc.ic.ac.uk/projects/tracta>.

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

[13] Mandrioli, D., Morasca, S., Morzenti, A. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Trans. on Computer Systems* 13, 4 (1995).

[14] McCabe, T.J. A Complexity Measure. *IEEE Trans. on Software Engineering* 2, 4 (1976).

[15] Mercier, F., Le Gall, P., Bertolino, A. Formalizing Integration Test Strategies for Distributed Systems. *1st Int. ICSE Workshop on Testing Distributed Component-based Systems* (Los Angeles (CA), USA, 1999).

[16] Perry, D.E., Wolf, A.L. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes* 17, 4 (October 1992), 40-52.

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

[18] Richardson, D.J., Wolf, A.L. Software testing at the architectural level. Second Int. Software Architecture Workshop ISAW-2 in *Joint Proc. of the ACM SIFSOFT '96 Workshops*, (October 1996).

[19] ROSATEA group. Architecture-based Analysis and Testing, On-line at <http://www.ics.uci.edu/~rosatea>.

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

[21] Szyperski, C. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow (England, 1998).