11th International Conference

Software Engineering & its Applications

PARIS

Software & Systems Engineering

December, 8-9 & 10 *1998*

PROCEEDINGS - Vol. 1

Preprints

ESI European Software Institute

A Systematic Approach for Integration Testing of Complex Systems

Frédéric Mercier†,‡, Antonia Bertolino†,
Pascale Le Gall‡ and Gilles Bernot‡

† : Istituto di Elaborazione della Informazione
CNR, Via S. Maria, 46
56126 Pisa, Italy

‡ : LaMI, Universite d'Evry
Cours Monseigneur Romero
91025 Evry Cedex, France

Abstract: For modular systems with hierarchical structure, approaches like top-down, bottom-up, and mixed strategies thereof, have traditionally been used for integration testing purposes. Such approaches are evidently no longer adequate for modern complex systems, which are made of many components (each, likely, a complex system itself) variously distributed and interconnecting. New strategies for arbitrary software architecture need to be identified.

We introduce a novel approach, based on a dedicated specification formalism called *information spaces*. Such approach supports a systematic decomposition into relevant subsystems for testing, driven by system structure and information processing.
We illustrate the funding concepts of our approach, i.e. *architecture topology*, *event expressions*, *information slices* of a specification and finally *test classes*, which figure a new means to raise integration errors.

Keywords: integration testing, software architecture, test class

1 Introduction

Traditionally, we all learnt that the integration testing of large systems must proceed in incremental steps. For modular systems with hierarchical structure, top-down vs. bottom-up are two well-known alternative strategies and in practice, mixed strategies are used according to the cost of stubs and drivers (simulating resp. called and calling modules).

However, such approaches no longer fit the needs for testing and analysing modern complex systems. These ones do not follow classical hierarchical structure and are made of many components (each, likely, a complex system itself), variously distributed and interacting concurrently on network. New strategies for arbitrary software architecture need to be identified.

But before any further consideration on integration testing, it is important to state exactly what we are focusing on. Let us start from the definition of integration given in [Parr89] : "Integration is the process of assembling those modules or components, which have been developed and previously tested in isolation, into a system which will appear to be a single entity". In the case of current complex systems, this process, whose objectives still remain the same, will add three elements which have not been previously tested. These three elements are: components' connectivity, components' communication and combinations of distributed functionalities.

To fill these needs, there is a general agreement that integration testing, much more than any other test stage, must gather correlatively both structural and functional testing criteria. But the principal weakness of current approaches [Jorg95] is that,

while led by some good intuitions, they try to reuse existing techniques instead of identifying more adequate approaches and develop new techniques to support them[1].

Indeed, the most commonly used strategies for integration testing in the context of non-modular complex systems are based on data-flow or transaction-flow analysis, which is a melt of data-flow and control-flow analysis. In practice the different models used for such analysis are conceived by the integrator from various sources of information and almost always after the production of the system's components. Spanning from the detailed specification to (some part of) the code, the tester will build his/her reference object based on his/her own understanding of the system. After that, the whole set of coverage criteria [Beiz95], also used in unit testing, can be applied according to costs and objectives.

We notice at least two flaws in such approaches. First the estimation of model adequacy relies directly on the expertise of the tester and is then subject to unexpected variations. Second the way to achieve the defined coverage is let in shadow and always relies on a deep understanding of functions involved in the model. Thus is also true for structural unit testing, but we can consider that achieving a coverage criterion on a complex system architecture would be far more difficult than solving some predicates, when it is feasible.

We then present in this article an alternative, new approach technically based on a dedicated specification formalism called *information spaces*. By means of a case study (Sect. 2), we introduce (Sect. 3) a couple of funding concepts such as *system architecture* and *event expressions*. Finally (Sect. 4) we outline the notions of *information slice* of a specification and the associated *test classes*, which figure a new means to raise integration errors during systematic testing.

2 Case Study : TRMCS

To support the reader intuition while we illustrate our approach, we will use a case study titled Teleservices and Remote Medical Care System (TRMCS) [Bals98]. The system provides assistance services to users with specific needs, like disabled or elderly people. A typical assistance service is to send relevant information to a local phone-center so that family, medical and/or technical assistance can be notified of critical circumstances. A top level model of the system, which so far involves few components, is drafted in Fig. 1.

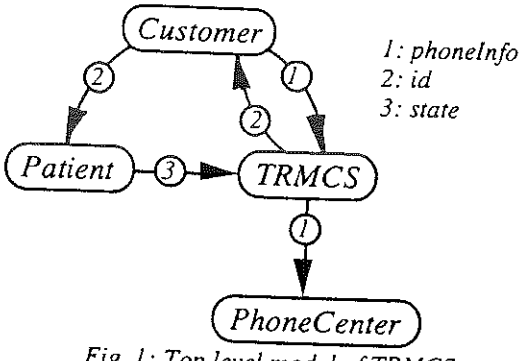


Fig. 1: Top level model of TRMCS

In this model the component named *Customer* holds for a human customer that enrols in the system. The component named *Patient* holds for a possible technical device

[1]Probably because in the same way than the "Bourgeois Gentilhomme" of Moliere was speaking in prose without knowing it, we were integrating without any necessity to fix our intuition in a theoretical corpus.

(e.g. a PC) that interfaces a human patient with the rest of the system. Typically, such a device regularly sends information about the state of the monitored patient.

Such a model does not justify so far the label of a complex system. However, a description of the same system at an intermediate level of abstraction, such as in Fig. 2, already reveals many interactions hidden at the top level.

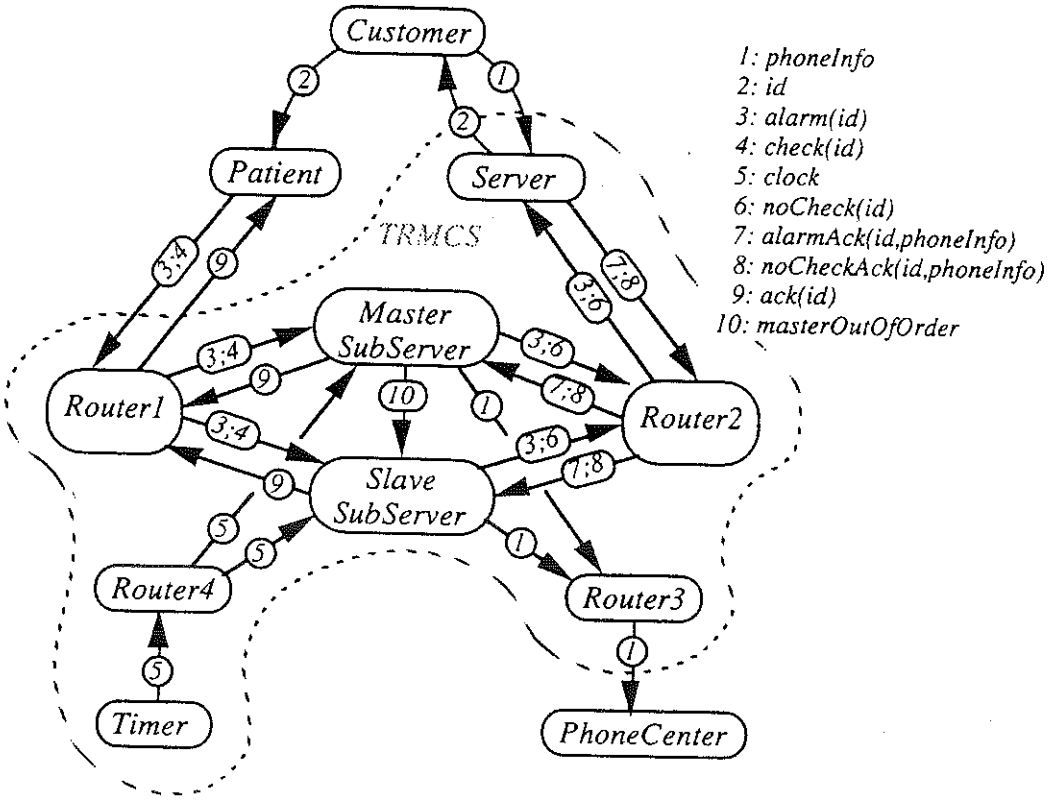


Fig. 2: A more detailed model of TRMCS

The *TRMCS* component (dashed area) has taken shape in this model. For geographical needs it has been configured as a main server and a sub-server. To ensure availability, the latter has been doubled in a master / slave architecture. Finally an external timer has been added. One can also notice that some information from and to previous *TRMCS* component has a more precise description. In particular, the previous *state* information has been refined in *check* and *alarm*, where *check* informs regularly the system of the steady state of the real patient (and in the same time of the working order of the *Patient* device), and *alarm* warns of a critical state of the real patient.

For space limitation we will not exhibit the full specification of this example but we will focus on subsets to illustrate our formalism and testing framework.

3 Specification Formalism

This work lays within the frame of the emerging field of Software Architecture [ROSA98], although we prefer to talk more generally of System Architecture, with a special focus on *integration testing* concerns. As most current studies in this field essentially focus on the specification and verification of models, our approach stands out by considering the use of architecture models as reference object for the testing of implemented systems.

Consequently, our approach is complementary to current trend and opens the question of testability of system architecture.

In the following we define the funding notions of *architecture topology* and *event expressions* before introducing the formalism of *information space* on which our approach relies.

3.1 Architecture Topology

The most intuitive representation of a complex system made of interacting components remains that of an oriented graph.

Definition. *(Architecture) An architecture is an oriented graph* $G = \langle N, A \rangle$ *so that :*
N *is a set of nodes corresponding to the components of the described architecture;*
$A \in N \times N$ *is a set of arcs corresponding to the communication channels between components.*

What we describe using such a graph is a constrained space of communication, driven by our intuition of the expected role and behaviour of each component, as well as the intuition of the objective and the meaning of each connection. Nevertheless, we know from experience that several architectural solutions can fill the same requirements. Consequently, one can consider architecture topology like a flexible support to what we are really interested in: the information processing, i.e., the global behaviour of the system.

3.2 Event Expressions

The information associated to any system is basically of two kinds: primary information, like that derived from physical measurements, or more generally any piece of information indivisible at the level of abstraction we are. The second kind corresponds to the information that is the result of a computation and so carries out the system's functionality. Using the notions of *information signature* and *information terms*, we define these two kinds respectively as basic and functional information terms.

Definition. *(Information Signature) An information signature is a couple* $\Sigma = \langle I, F \rangle$ *so that:*
I *is a set of basic information names;*
F *is a set of function names with strictly positive arity;*
The infinite set T_Σ *of information terms on* Σ *contains* I *and all terms recursively derivable by functions in* F.

For example, *id* is a basic information associated with the *Customer/Patient* concept, while *alarm(id)* extends the meaning of *id* building a new information.

To express the computation occurring between a basic primary information and a synthesised one, we then define a language of event expressions :

_ holds for the empty event.

$i[t]$ (resp. $o[t]$) where $t \in T_\Sigma$, holds for the input (resp. output) event of the information t.

$e_1.e_2$ where e_1 and e_2 are event expressions, holds for a sequence of event expressions. For instance, $i[id].o[alarm(id)]$ specifies that after the input of id, the output of $alarm(id)$ occurs.

$e_1 \& e_2$ holds for a conjunction of event expressions. Intuitively, this operator expresses that either the sequence $e_1.e_2$ or the sequence $e_2.e_1$ are expected to occur.

$e_1 | e_2$ holds for an exclusive disjunction of event expressions. This operator expresses that either e_1 or e_2 are expected to occur.

$e!$ holds for a "one time" expectation of e. This specific statement introduces a modal dimension in the possible execution paths associated to event expressions. For example, the event expression $i[id]!.(o[alarm(id)] \| o[check(id)])$ describes the sequences $i[id].o[alarm(id)]$ or $i[id].o[check(id)]$ for the first possible execution paths and $o[alarm(id)]$ or $o[check(id)]$ for the following ones.

$e_1 ; e_2$ holds for a set of independent event expressions. Each of e_1 and e_2 can occur independently of the other. Intuitively this corresponds to the allocation of threads to e_1 and e_2.

Finally, given a signature Σ, we define the set $sen(\Sigma)$ of all possible event expressions (or *sen*tences) on Σ (i.e. event expressions with information terms belonging to T_Σ).

3.3 Information Space

We can finally define *information spaces* that gather the concepts of architecture topology, information terms and event expressions. Precisely, in an *information space,* the behaviour of each component of the architecture is described by an event expression and each communication channel is labelled with carried information terms.

Definition. *(**Information Space**) An information space is a tuple* $IS = \langle \Sigma, G, \bar{\mu} \rangle$, *such that :*

Σ *is an information signature;*

G *is an architecture topology;*

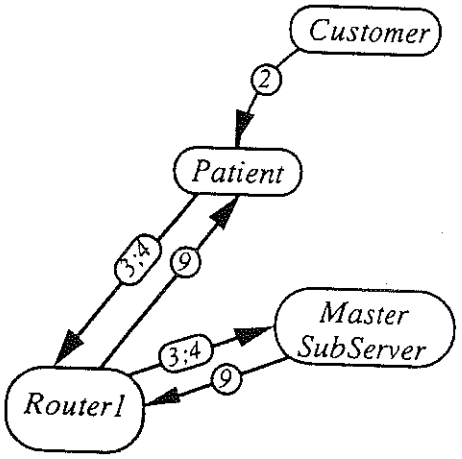
$\bar{\mu} = (\mu_A, \mu_N)$ *is a vector of mapping functions :* $\begin{bmatrix} \mu_A : A \to T_\Sigma \\ \mu_N : N \to sen(\Sigma) \end{bmatrix}$,

and such that for each event expression associated with a node, input (resp. output) information label the incoming (resp. outgoing) arcs[2].

Since our approach is based on the former definitions, we now present part of the TRMCS specification based on the above formalism of information spaces.

Let us first put the layout of a specification. An information space specification has a title and is built of three parts. The first part corresponds to the list of all information terms involved in the information space. The second part is the list of the architecture components with their associated event expressions. Finally, the third part is the list of all communication channels (i.e. couple of components) with their associated set of carried information. For example, if we focus on the part of the architecture containing the *Customer*, the *Patient*, the *Router1* and the *MasterSubServer*, we have the following piece of the TRMCS specification (not relevant event statements are written in light):

[2] If necessary, to avoid some ambiguity we express which arc is concerned with a given input (resp. output) event.




```
info_spec: TRMCS
view:   global
info_description:
  id:
    "user identifier."
  alarm(id):
    "alarm signal."
  check(id):
    "check signal."
  ack(id):
    "acknowledgement to the user."
  ...
components:
  Customer:
    (o[phoneInfo] . i[id] . o[id])!
  Patient:
    i[id]! . (o[alarm(id)] . i[ack(id)] ! o[check(id)])
  Router1:
    (i[check(id)] . o[check(id)]
      ! i[alarm(id)] . o[alarm(id)] . i[ack(id)] . o[ack(id)])
  MasterSubServer:
    (i[alarm(id)] . o[alarm(id)] . i[alarmAck(id,phoneInfo)] .
    o[ack(id)] . o[phoneInfo]
      ! (i[check(id)] . i[clock])! . (i[check(id)] . i[clock]
        ! i[clock] . o[nofunc(id)] . i[noCheckAck(id,phoneInfo)]
          . o[phoneInfo]);
    o[masterOO]
  ...
connections:
  Customer -> Patient          : id
  Patient -> Router1           : alarm(id); check(id)
  Router1 -> Patient           : ack(id)
  Router1 -> MasterSubServer   : alarm(id); check(id)
  MasterSubServer -> Router1   : ack(id)
  ...
```

Fig.3: Part of TRMCS information space specification.

We are now ready to address integration testing concerns within the framework of information spaces.

4 Test Classes

Building up a specification using the information space formalism gives us a strongly defined reference object from which it is possible to conceive tests. In particular verifying the adequacy of the implemented system to its information space obviously fits the goals of integration testing, since both architectural and information processing aspects of system's components are expressed within the specification.

We now introduce a concept of *coverage over information spaces*, standing on the intuition that testing the processing of an information at a particular point of the system corresponds to testing a related sub-part of the system.
The functional meaning associated to this intuition is clear: this information has a definition domain and is possibly the result of several computations through the system; so the underlying goal is to verify the adequate composition of the computations leading to this information.
The structural counterpart is also obvious: as the building process of an information is variously distributed over the components of the architecture, testing an information also verifies the connectivity and the interfaces of the components involved in its

building process. Clearly to achieve a complete verification of the interaction between system's components, it is not only necessary to test all information handled by the system, but also to do so at any sensitive point of the architecture.

Deriving from an information space, which implicitly defines the global event schedule over the system, all and only the part of the specification relevant to a specific information corresponds to extracting a *slice* of the information space. Given an input event *i[t]* and its location, an *information slice* is an information space segment whose behaviour description is identical to the global information space with respect to *i[t]*.

For example in TRMCS, the slice associated to the input of the information *alarm(id)* from the *Server* component is the following :

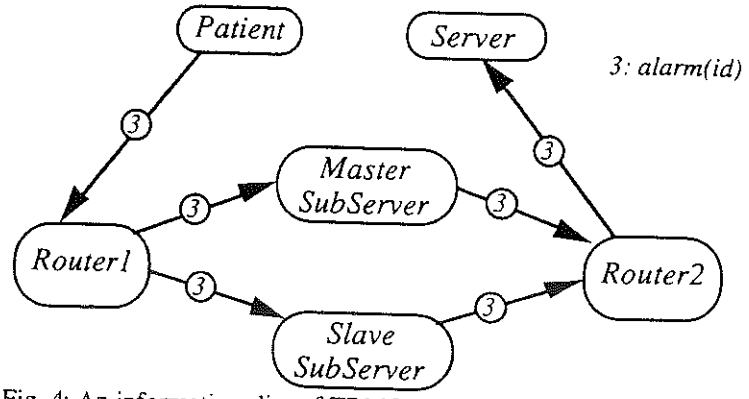


Fig. 4: An information slice of TRMCS w.r.t. the information alarm(id)

If we now consider the set of slices related to all information handled by the system, we can observe that some slices are included in some other. Indeed, this simply reflects the intuition that the building process of an information defines a hierarchy of information dependencies. More interesting is the range of integration testing strategies that can stand on this observation: from the more internal slices to the more external ones, or the opposite or something else... the field is still open.

We are now in a position to introduce *test classes* that constitute a first step towards how to test information slices. Given an information slice, defining a test class (which has to be filled with test cases associated to a more concrete level) corresponds to assigning specific test roles to some of the slice components. Basically, we distinguish between those components from which test input data are launched (the *launchers*) and those other ones on which the test oracle could be located. What we do is a partition of the set of components related to the slice. Oracles and associated launchers are the borders of the slice, since the launchers produce basic information necessary to build the information tested on the oracle. The other components in the slice form the part of the system that is actually tested to verify their interactions.

Intuitively, oracles are those components that receive the selected information as input and keep it (i.e., do not re transmit it).
Considering again the slice associated with the information *alarm(id)* given as input to the *server* component (Fig. 4), we select the *Server* as the sensitive oracle, while the two routers and the two sub-server have been rejected since they only act as transmitters of the information *alarm(id)*. Then we back track on the slice from the *Server* till the *Patient* component. This one produces the information *alarm(id)* and is then selected as a launcher.

But such a test class is not yet precise enough. In particular we do not state if the transmission process we test should be handled by *MasterSubServer* or by *SlaveSubServer*.

We note that the execution of a class related to a specific information may depend on other information terms, which condition the achievement of desired schedules. We call these particular information the *triggers* of a given class and add it to the test class definition.

Finally, a test class has the following profile :

$$(info, oracle, launchers, triggers)$$
$$T_\Sigma \times N \times \wp(N) \times \wp(T_\Sigma)$$

For the information *alarm(id)* we obtain the two following test classes: *(alarm(id), server, {Patient}, Ø)* and *(alarm(id), server, {Patient}, {masterOutOfOrder})*. In fact, when we back track from *Router2*, we get two different contexts. In particular, the part of the slice associated with the *SlaveSubServer* is conditioned by the receipt of the *masterOutOfOrder* information. In such cases, we further, subdivide the test class by giving the variation in terms of triggers.

In particular this notion of a context (on which the test classes extraction process is basically funded) can be extended to give further details like "on which intermediate information we focus our attention to fill a given test class". The set of these extensions based on information space specification is yet not fully developed and we currently investigate some clues more specifically related to the test plan building process.

Finally the following table gives all the results we obtained for the TRMCS case study with an enhanced version of test classes description.

Information	Oracle	Launchers	Triggers	Via
phoneInfo	*Server*	*{Customer}*		
	PhoneCenter	*{Server; Patient}*		*alarmAck(...)*
	PhoneCenter	*{Server; Patient}*	*{masterOO}*	*alarmAck(...)*
	PhoneCenter	*{Server; Patient}*		*noCheckAck(...)*
	PhoneCenter	*{Server; Patient}*	*{masterOO}*	*noCheckAck(...)*
ack(id)	*Patient*	*{Server; Patient}*		*alarmAck(...)*
	Patient	*{Server; Patient}*	*{masterOO}*	*alarmAck(...)*
alarmAck(...)	*MasterSubServer*	*{Server; Patient}*		*alarm(id)*
	SlaveSubServer	*{Server; Patient}*	*{masterOO}*	*alarm(id)*
noCheckAck(...)	*MasterSubServer*	*{Server; Patient}*		*noCheck(id)*
	SlaveSubServer	*{Server; Patient}*	*{masterOO}*	*noCheck(id)*
alarm(id)	*Server*	*{Patient}*		
	Server	*{Patient}*	*{masterOO}*	
noCheck(id)	*Server*	*{Patient}*	*{clock}*	*check(id)*
	Server	*{Patient}*	*{clock; masterOO}*	*check(id)*
check(id)	*MasterSubServer*	*{Patient}*		
	SlaveSubServer	*{Patient}*	*{masterOO}*	
clock	*MasterSubServer*	*{Timer}*		
	SlaveSubServer	*{Timer}*	*{masterOO}*	
masterOO	*SlaveSubServer*	*{MasterSubServer}*		
id	_			

With respect to system level testing, we can observe that the testing of *phoneInfo* involved transitively almost all information of the system. Moreover as the test classes of *ack(id)* are included in those of *phoneInfo* (intuitively a kind of slice dependency), it is possible (and advisable) to plan these test classes to be run together, which supports our intuition of the strict correlation between these two information. Finally we observe that *id* cannot be tested with our current definition of oracles selection. Indeed, *id* has a slightly different role than the other information and stands more as a kind of internal variable of the system. It is to say that testing more significant information like *phoneInfo* implicitly tests information like *id*.

5 Conclusions and Future Work

Supported by information spaces and test classes, we have presented a flexible integration testing approach based on the notion of information processing by components.

But this work is still ongoing and for the future we plan the following investigations:

- to enrich the language of event expressions while keeping the information slice extraction decidable.
- to address test plan extraction based on information slicing and related information test classes.
- to establish links between our abstract specification of information spaces and a more concrete one in order to generate test cases and fill information test classes.

Finally, as a practical counter part, an interactive compiler of information spaces specifications is currently under development. This is actually designed to extract information test classes as presented in this article. One can assume that the variety of case studies we will be able to handle with this tool will allow us to address new development stages of strongly associated theoretic and practical aspects of integration testing of distributed systems.

Acknowledgements

Thanks are due to the "OLOS" European HCM research network (Contract CHRX-CT94-0577), which supported Frédéric Mercier's work at IEI in Pisa under its programme of young researcher exchanges. This work was also partly supported by the ESPRIT-IV Working Group 22704 ASPIRE, the ESPRIT-IV Working Group 23531 FIREworks and the French research project "PRC-GDP ALP".

References

[Bals98] S. Balsamo, P. Inverardi, C. Mangano and F. Russo "Performance Evaluation of a Software Architecture : A Case Study", IEEE Proc. IWSSD-9, April 1998.

[Beiz95]. Boris Beizer "Black Box Testing", Wiley, 1995.

[Jorg95] Paul C. Jorgensen "Software Testing, A Craftsman's Approach", CRC Press, 1995.

[Parr89] Norman Parrington and Marc Roper "Understanding Software Testing", Ellis Horwood, 1989.

[ROSA98] ROSATEA International Workshop on the Role of Software Architecture in Testing and Analysis, July 1998, Marsala, Italy.