PRINCIPLES OF DISTRIBUTED TEST SYNTHESIS BASED ON TRUE-CONCURRENCY MODELS

Claude Jard
IRISA/CNRS
Campus de Beaulieu
F-35042 RENNES Cedex
France
Claude.Jard@irisa.fr

Abstract Automatic synthesis of test cases for conformance testing has been principally developed with the objective of generating sequential test cases. In the distributed system context, it is worth extending the synthesis techniques to the generation of multiple testers. We base our work on our experience in using model-checking techniques, as successfully implemented in the TGV tool. Continuing the works of A. Ulrich and H. König, we propose to use a true-concurrency model based on graph unfolding. The article presents the principles of a complete chain of synthesis, starting from the definition of test purposes and ending with a projection onto a set of testers.

Key words: Test, Distributed systems, Synthesis, True-concurrency models, Inter-operability

1. INTRODUCTION

Algorithms for automatic test synthesis have been proposed both in the academic world, and in industry. However, the use of these tools reaches a limit when testing distributed systems. This is because they are dedicated to the synthesis of sequential test cases (represented by event sequences or finite automata). Such synthesis is not always well-suited to test systems containing parallel activities. It is also known that a state representation of a specification with parallelism often suffers from a combinatorial explosion.

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35497-2_31

I. Schieferdecker et al. (eds.), *Testing of Communicating Systems XIV*
© IFIP International Federation for Information Processing 2002

The interest in generating distributed test cases was recognised a few years ago, as demonstrated by the inclusion of concurrent constructs in the TTCN standard. We retain three main motivations to synthesise distributed test cases:

- It can be naturally imposed by the test architecture under consideration. Let us consider a system geographically scattered on a network. The idea is to design a set of testers, each tester being located at the communicating entity to be checked, and communicating with the other testers to co-ordinate the test activity and the production of diagnosis.
- It allows more compact and clear test cases to be obtained. This is the case when the system under test produces concurrent observable events: a sequential representation would require all possible interleavings to be computed. This rapidly suffers from a combinatorial explosion as the concurrency increases.
- In certain cases parallel testing is needed to check particular behaviours. For example, one often considers for controllability reasons that the testers must wait for the system stabilisation before injecting new interactions. Under this assumption, it was shown by [2] that a distributed test case can position the system under test into states which are not reachable by a sequential test. More generally, the situation will also occur in the context of real-time testing.

One can distinguish two main approaches to synthesising distributed test cases:

- The generation of sequential test cases, followed by their automated distribution. The idea is to produce a set of communicating testers which behave like the sequential test (i.e. in the sense of trace equivalence). The advantage of this approach is that it requires no more than the current state of the art; it can even be used on hand-written test cases. The major drawback is that it does not take into account the intrinsic parallelism of the system under test. In general, one does not know how to distinguish between parallelism and interleaving; in practice this leads to useless synchronisation between the local testers.
- The re-examination of the synthesis, retaining the parallelism information contained in the formal specification during the construction of the test cases. We discuss this extreme approach in the paper. The main difficulty is the use of a true-concurrency model in which causality and concurrency are explicitly represented, in place of the usual automata or transition system models. This kind of model has been mainly developed by theoreticians and has not yet been fully exploited. The synthesis of distributed test cases appears to be an interesting context to use the explicit parallelism included in the model.

The question of the automatic synthesis of distributed testers is relatively recent. It has appeared gradually from the notion of multiple, then distributed interfaces. For example, in [18], the system under test is modelled by a single finite state machine with several distributed interfaces. A test generation method is sketched, based on the idea of synchronisable test suites. In [19], multiple testers are generated by considering only particular synchronous behaviours of the parallel specification. Co-ordination of the testers makes the assumption that the communications between entities of the system under test are observable. The idea of using true-concurrency models in the case of asynchronous systems came from two research groups separately (one in Korea, driven by M. Kim, the other in Germany, driven by A. Ulrich and H. König). In Kim's approach [20,21], they adopt a specific model, which consists in computing particular concurrent paths from a communicating finite state machines view. The introduction of event duration makes the computation easier. It is not clear however to know the algorithmic complexity of the method and how it scales up in a real testing methodology (abstraction, selection, ...). We chose to follow the Ulrich and König's approach [22], mathematically based on theoretical and algorithmic results on Petri nets. The partial order semantics of Petri nets and its implementation in the "unfolding" algorithmic has been developed for many years, but rather confined in the theoretical computer science community. We think it is enough sound and advanced to be applied in several domains... like distributed testing. In [22], the unfolding of "behaviour machines" is used to propose a "partial order transition cover" as a general heuristics to select partial order test cases, which could be later projected on parallel testers. In the same vein, [23] has tempted to avoid the use of Petri nets and to directly generate the partial orders (event structures) in the context of asynchronous communication.

The rest of the paper is organised as follows: First, we give an overview of a test synthesis method based on model-checking and sequential transition systems, as implemented in the TGV tool. We then propose to revisit the whole test-production chain using partial-order representations of the behaviours. This is presented in Section three, following the different steps of the methodology: the partial order view of a specification (the notion of "tile") and of a test purpose, the construction of an unfolding (the "puzzle game"), its partial order abstraction, and its final projection onto several testers. Particular attention is paid to the algorithmic complexity and its potential to scale up, in the perspective of developing a real prototype. Some indications of possible future developments are given in the conclusion.

2. SYNTHESIS BASED ON TRANSITION SYSTEMS

We mainly rely on our experience in conformance testing. The TGV tool (Test Generation using the Verification technology) [3,4], jointly developed by our group at Irisa and a group at Vérimag, is a real-size implementation of synthesis techniques based on transition systems. We thus begin by recalling the main principles.

2.1 Our example

Let us consider the small example depicted in Figure 1. This is a simple connection-disconnection protocol, modelled with two interacting finite automata communicating through one-bounded channels. The users can use the protocol by asking for a new connection (event **a**), or asking for the disconnection of a previously opened connection (event **b** from one side, event **c** from the other side). Each request is acknowledged by the local process after having performed the corresponding action (events **a**, **b** and **c**). The protocol manages a possible collision of disconnect messages by exchanging a disconnect confirmation, named d. The scenario presented in Figure 1 illustrates the three possible repeatable behaviours (connection is closed by the initiator, connection is closed by the other side, and collision). The message exchanges between processes are neither controllable, nor observable.

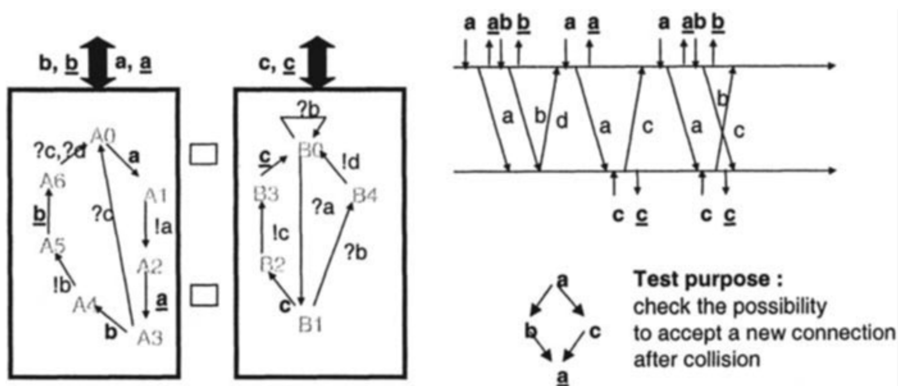

Figure 1. A small example of connection-disconnection protocol and a possible test purpose.

The test objective here is to check the possibility to accept a new connection after collision (collision management is known to be a fragile aspect of this kind of protocol). This can be naturally described by the partial

order given in Figure 1: **a** must precede **b** and **c**, which precede **a**. The description by communicating finite automata is just for illustration purpose. In the real use of TGV, models are described in higher-level languages like SDL, Lotos or UML [5,6,7], the associated compiler providing the state representation.

2.2 The state-graph representation

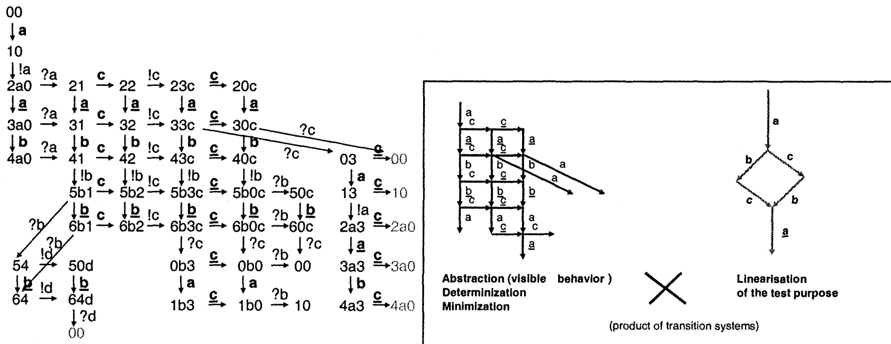

Figure 2. Finite automata, trace-equivalent to the communicating automata of Figure 1; its visible abstraction, guided by the automaton of the test purpose. Each state of the test purpose accepts all the events (the single loops are not represented). The sink state is labelled by *accept*.

The exhaustive simulation of the protocol, starting in the initial state A0B0 with empty channels, and keeping track of the global states reached by the simulator gives the graph depicted in Figure 2. Notice that cycles are built when reaching a previously generated state. This state graph captures all the possible traces of the protocol (the interleaving is computed in the case of concurrent events). From this graph, one can compute a trace-equivalent automaton restricted to the alphabet of visible events. A part of this graph can be selected using the test purpose in order to keep only the traces it accepts (leading to the sink state PASS).

2.3 Test synthesis

The resulting graph representing all the visible traces of the specification, consistent with the test purpose, defines a set of possible test cases (up to the inversion of interactions). In general, to reduce the complexity, one extracts only one test case using some heuristics. For example, we consider that a test case must be (globally) controllable, which means that there is no choice

between an emission from the tester and another interaction. In our example, the resulting test case is given in Figure 3. To be complete, we must mention that the test case is augmented with verdicts (PASS in the final state, FAIL for possible receptions that are not in the graph, INCONCLUSIVE when the reception is not on a path selected by the test purpose) and timers (to prevent the test from dead-, live- and output-locks). The generated test cases are guaranteed to be safe (they cannot reject a conformant implementation in the sense of *ioco* conformance [8]). The method is also complete in the sense that it is able to reject any non-conformant implementation (assuming the provision of a corresponding test purpose, and some fairness assumptions about the implementation).

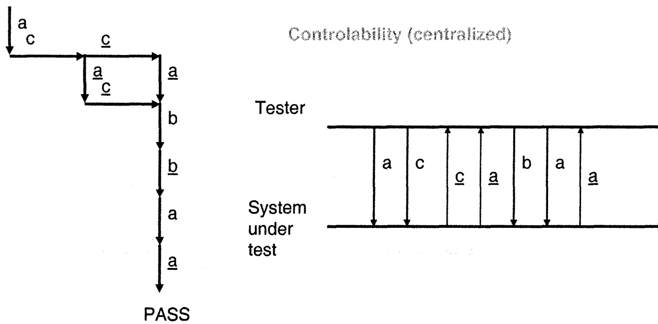

Figure 3. Resulting test case under controllability assumption.

In the complete version of TGV, these different graphs are not built in sequence. Their construction is performed on-the-fly during the synthesis of the test case through the use of APIs at different levels [10], as depicted in Figure 4. The algorithms are mainly based on adaptations of Tarjan's algorithm [9], computing the strongly connected components of a graph via a depth-first search. The complexity is linear in the size of the state graph (though the graph itself may be exponential in the size of the model, notably when the concurrency is significant). Determinisation remains exponential, but is applied to the graph of visible behaviours, which is much smaller than the state graph.

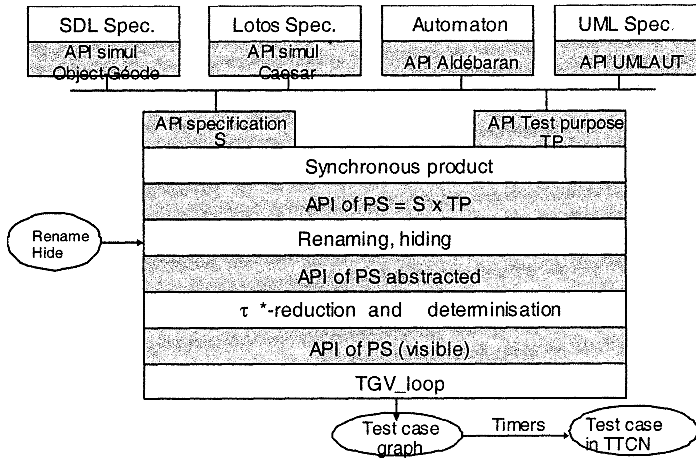

Figure 4. The on-the-fly organisation of TGV.

3. PARTIAL ORDER VIEW OF THE SYSTEM

3.1 Tile systems

In this section we introduce our mathematical framework. Tiles correspond to partial transitions and a system is defined as a collection of tiles.

Let **V** be a finite set of variables. Each variable $v \in \mathbf{V}$ takes its values in some finite domain D_v. For $V \subseteq \mathbf{V}$, we set $X_V = \prod_{v \in V} D_v$. Elements of X_V are denoted by x_V and are called *V-states*, or local states. For $v \in \mathbf{V}$, we denote by $v(x_V)$ the value of the variable v in state x_V. We shall consider local transitions relating local states, very much in the same way as transitions relate states in standard automata. These local transitions will be referred to as tiles in the sequel. Formally, a *tile* is a 4-tuple $\tau = < V, \bar{x}_V, \alpha, x_V >$, where $V \subseteq \mathbf{V}$ is a subset of variables, and $(\bar{x}_V, \alpha, x_V)$ is a local transition, relating the previous V-state $\bar{x}_V \in D_V$, and performing event α where α ranges over some set A of possible event labels. For τ a tile, we shall sometimes denote by V_τ its set of variables. A system is a triple

$\Sigma =< V, X_0, \mathrm{T} >$, where $V \subseteq \mathbf{V}$ is a finite set of variables, X_0 is a set of initial states, and T is a finite set of tiles and $V = \bigcup_{\tau \in \mathrm{T}} V_\tau$.

Figure 5 shows the tile system of our example, as it is entered in our prototype.

```
% Variables
var A : 0..6 init 0;
    B : 0..4 init 0;
    M : (O,a,b) init O;
    N : (O,c,d) init O;

% Tiles
?A  pre A(0) label ?A post A(1);
?b  pre B(0) M(b) label ?b post B(0) M(0);
!a  pre A(1) M(0) label !a post A(2) M(a);
?a  pre B(0) M(a) label ?a post B(1) M(0);
!A  pre A(2) label !A post A(3);
?C  pre B(1) label ?C post B(2);
?B  pre A(3) label ?B post A(4);
!c  pre B(2) N(0) label !c post B(3) N(c);
?c  pre A(3) N(c) label ?c post A(0) N(0);
```

Figure 5. The tile system of the example of Figure 1 in textual form.

The interleaved sequence of states and events $x_0, \alpha_1, x_1, \alpha_2, x_2, \ldots, \alpha_k, x_k, \ldots$ is a *run* of system Σ if $x_0 \in X_0$ and,

1. for each $k > 0$, there exists $\tau =< V_\tau, \bar{x}_{V\tau}, \alpha, x_{V\tau} > \in \mathrm{T}$ such that, $\forall v \in V_\tau : v(x_{k-1}) = v(\bar{x}_{V\tau}), \alpha_k = \alpha, v(x_k) = v(x_{V\tau})$, and,

2. $\forall v \notin V_\tau : v(x_{k-1}) = v(x_k)$.

Since tiles define local transitions, it may be the case that two successive tiles of a given run involve disjoint sets of variables, i.e. modify different local states. In this case, exchanging the order of the tiles yields to an equivalent run. This is why we will adopt a partial ordering of tiles instead of considering the different runs.

3.2 Construction of the unfolding

Given a run, the sequence of successive tiles forms a graph, by superimposing the pre-condition of a tile $v(\overline{x}_{v\tau})$ onto an equivalent condition in the existing graph (like a puzzle game). This graph contains two types of nodes: the conditions (the different values of the variables used in pre and post conditions of the tiles), and the events of the tiles. Given two nodes *n* and *n'* (condition or event), we say that *n causes n'*, written $n \leq n'$, if either $n = n'$ or there is a path of arrows from *n* to *n'*. We say that *n* and *n'* are in *conflict*, written $n \# n'$, if there is a condition *m*, different from *n* and *n'*, from which one can reach *n* and *n'*, exiting *m* by different arrows. Finally we say that *n* and *n'* are *concurrent* if neither $n \leq n'$, nor $n' \leq n$, nor $n \# n'$ hold. A *co-set* is a set of concurrent nodes. From a tile system $\Sigma = < V, X_0, \mathrm{T} >$, the basic algorithm for the construction of the graph is the following:

Puzzle := X_0 *;*
repeat
if there exists a tile $\tau \in \mathrm{T}$ *such that* $\overline{x}_{v\tau}$ *is a co-set of Puzzle then append* τ *to Puzzle*
forever

Figure 6 shows the graph obtained after 8 steps of the above algorithm.

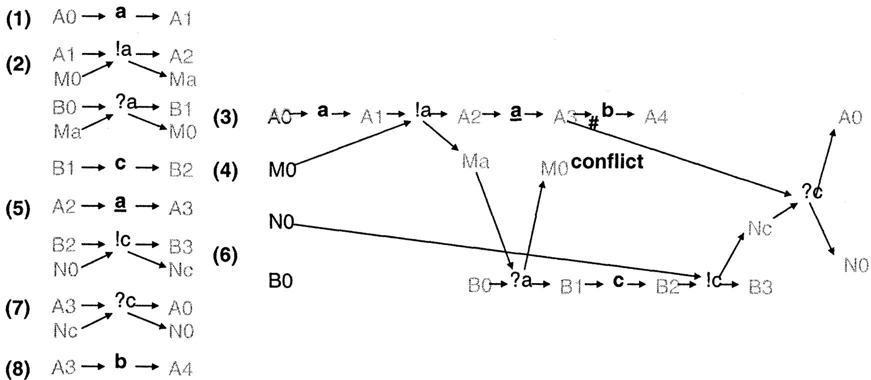

Figure 6. Result of the application of these 8 tiles in sequence. Conflict between tiles 7 and 8 is pointed out by the branching from condition A3.

This graph is generally infinite (in the case of infinite behaviour), it has no circuits, every condition has at most one input node, every node has a finite number of predecessors in the graph, and no node is in self-conflict. It is in fact an occurrence net in the framework of Petri nets. We can use the

corresponding terminology. A *cut* is a set of conditions c satisfying the following two properties: c is a co-set, and c is maximal (it is not properly included in any other co-set). A *configuration* is a set of nodes κ satisfying the two following properties: κ is causally closed (if $n \in \kappa$ and $n' < n$, then $n' \in \kappa$) and conflict-free (no two nodes of κ are in conflict). Furthermore, we require for convenience that all maximal nodes (if any) of configurations shall be conditions. Finite configurations and cuts are closely related. In particular, given a finite configuration κ the set of conditions *Cut(κ)* is a reachable global state, which we denote *GS(κ)*. The basic algorithm will eventually produce any reachable global state under only the fairness assumption that every tile candidate to be added is eventually chosen to extend the puzzle (the correctness proof follows from the definitions and from the results of [11]).

It appears that the unfolding is of fractal nature, and can be reduced to a finite generator part, called a finite complete prefix. A prefix *U* of the unfolding is complete if for every reachable global state *S* there exists a configuration *C* in *U* such that $GS(C) = S$ and for every tile τ enabled by *S* there exists a configuration $C \cup \{e\}$ such that $e \notin C$ and *e* is the event of τ. A complete prefix contains as much information as the unfolding, in the sense that we can construct the unfolding from it as the least fix-point of a concatenation operation on patterns defined by maximal configurations of the prefix . In order to construct such a prefix, the question is to locate the events (called the *cut-off events*) from which the extension in the unfolding can be stopped. We will denote *[e]* the set of predecessors of *e* (the set of events *e'* such that $e' \leq e$). An event *e* of the prefix is a *cut-off event* (with respect to a particular order $\prec$) if the prefix contains an event *e'* such that $GS([e]) = GS([e'])$, and $[e'] \prec [e]$. The algorithm to construct a finite complete prefix is the following:

```
Finite_Puzzle := X0;
cut_off := {};
repeat
   Select a tile τ such that X̄vτ is a co-set of Finite_Puzzle;
   live := τ exists and X̄vτ ∩ cut_off={};
   if live then append τ to Finite_Puzzle;
                if ∃ u ≺ t : GS(u)=GS(t)  then cut_off := cut_off ∪ Xvτ
until not live
```

The correctness of the algorithm requires that the partial order $\prec$ be correctly chosen. In [12], it is proved that $\prec$ must be *adequate*, that is defined as an order that is well-founded, which refines the set inclusion and which is preserved by finite extensions. The size of the prefix also depends

on this order, but it is possible to guarantee that the prefix is never larger than the global reachability graph (states + transitions).

> The running time of the algorithm is $O\left(\left(\frac{|C|}{\varphi}\right)^{\varphi}\right)$, where C is the set of conditions of the prefix, and φ denotes the maximal size of the pre-conditions of the tiles in the original system.

Figure 7 shows the complete prefix of our example as computed by the Esparza-Römer-Vogler's unfolding algorithm (available through the "Model-Checking Kit" of the Technical University of Münich [26]). The slowest part of the algorithm is locating the possible conditions that can be covered by a new tile. This is implemented by coding the concurrency relation and providing a method of maintaining it. This deteriorates as the size of the prefix increases, since the amount of memory needed to store the concurrency relation may be quadratic in the number of conditions in the already built part of the prefix. A recent improvement proposed in [13] structures the set of events in order to speed up the search in practice, not by trying the events one by one, but several at once, merging the common parts of the work.

The unfolding can be generated from this prefix by considering three maximal configurations ended by the sets {A0,M0,N0,B0}, {A4,M0,N0,B1} and {A2,Ma,N0,B0} respectively, shown as dashed lines in Figure 7. From these maximal nodes in the prefix, the unfolding can be continued by gluing the pattern starting from a similar cut in the prefix.

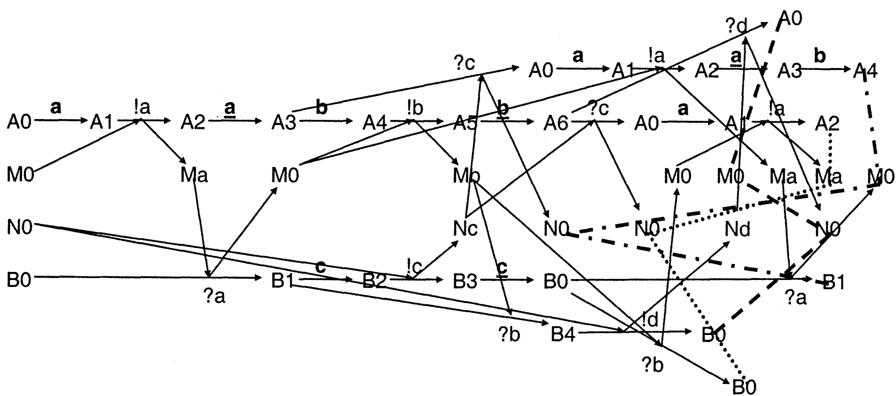

Figure 7. Complete finite prefix of our example.

3.3 Guiding by test purposes

Test purposes are a very interesting feature when dealing with large specifications. Their role is to mark out a relevant part of the specification in which a test must be found. This concept, present since the beginning in the ISO methodology, is rich enough to continue to arouse discussions in a broader community [14,15]. In TGV, a test purpose is given by a finite automaton with sink states labelled by *accept* or *refuse*. A transition of the specification is triggered if there exists a similar transition in the test purpose. It thus allows some transitions to be cut in the state graph representation. The accept state will become the PASS state in the final test case.

This point of view can be easily ported to partial order models, by considering that test purposes are particular tile systems, with two terminal specific tiles having accept or refuse as post-conditions. From the two tile systems (the specification and the test purpose), one can derive a new tile system, coding the product. The principle is as follows: for each tile of the test purpose, let us consider a tile of the specification with a similar event label and build a new tile by making the conjunction of pre-conditions and the conjunction of post-conditions. This can increase the number of tiles. The unfolding is then carried out on this new tile system. An alternative could be to perform the product on-the-fly during the construction of the unfolding, instead of pre-computing the new tiles. This would imply to keep the maximal configurations in the prefix in order to be able to continue the unfolding if required by the test purpose. Placing a tile containing an accept or refuse condition is considered as a cut event in the algorithm. Figure 8 shows the tile system of the test purpose chosen in our example.

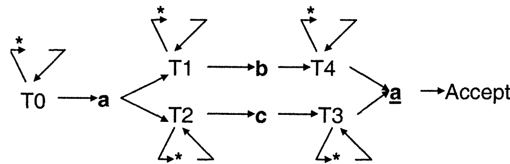

Figure 8. A partial ordered test purpose. "*" means any event label of the specification, except the outgoing events.

3.4 Abstraction

At this step, the complete prefix contains all the information needed to generate a test case (no further unfolding is needed, since all the realisation

of the test purpose has been considered in the new tile system, resulting from the product of the specification with the test purpose). We consider that the relevant semantics for a test case is the partial order of its events. It can be extracted from the prefix by considering the observable events only and the paths in the graph linking them. The graph of the test case is the partial order defined as follows: the nodes are the events of the observable tiles, and a node n precedes a node n' if and only if there exists a path of arrows from n to n'. This operation is linear in the size of the prefix (computation of a sub-order). To be rigorous, we must distinguish the situation in which two nodes labelled with concurrent events have a common predecessor, from the case where the events are in conflict. The latter situation will be resolved in the projection phase, while the first will generate a local choice.

Some more pruning can be done on the graph. First, the only interesting paths are those leading to *Accept*. The others can be deleted, while keeping the branching information on the relevant paths in order to be able to set possible INCONCLUSIVE verdicts. Second, we generally require that test cases are controllable. In our framework, we consider *local controllability* only, which means that a reception in the tile system, occurring on a local process cannot have the same direct predecessor as another event of this process. Figure 9 shows the partial order of the abstraction in our example and the extracted sub-graph.

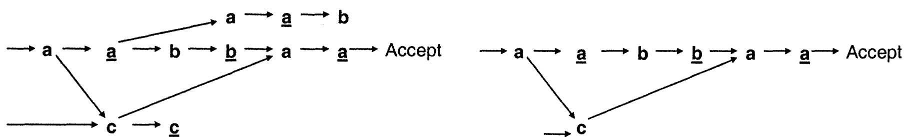

Figure 9. Abstraction and selection of a test graph.

3.5 Projection

The last step is the projection of the test graph onto the different testers. The principle is the following:

- the events of the testers are the mirror images of the events of the test graph (receptions from the point of view of the specification are the emissions of the testers, and vice versa),
- the graph of events of a tester is the projection of the test graph, keeping only the local events (another sub-order construction). At this step, it is advisable to build the transitive reduction of the local graph, in order to avoid redundant synchronisation (this is of cubic complexity, but the algorithm is applied generally on small graphs),

- a direct link between two events located on different testers is implemented by exchanging a synchronisation message (this particular message is emitted after the occurrence of the first event, and received by the other tester before performing the second event).

This is standard way to distribute the behaviour of an automaton. By construction, the partial order defined by the test graph is preserved by projection. Thus, all the traces are preserved too [16,17]. The result of the projection for our example is shown on Figure 10.

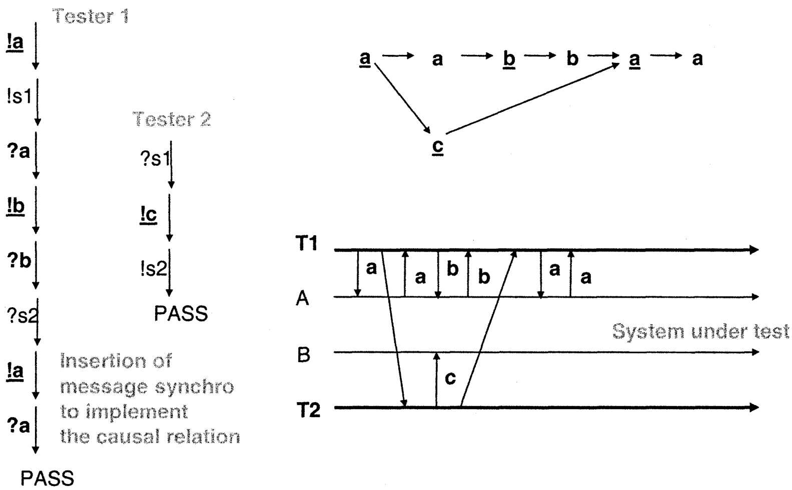

Figure 10. The resulting testers in our example.

It is interesting to compare this distributed test case with the sequential one of Figure 3. The automatic distribution of the test case of Figure 3 would produce much more synchronisation. Of course, the diamond (a,c) in the centralised test case is represented by real parallelism in the distributed test case. Furthermore, we can see that interaction !b (and ?b) is concurrent with !c in the distributed case. This situation seems impossible to infer from the sequential test case.

4. CONCLUSIONS AND PERSPECTIVES

Pursuing the approach initiated by Ulrich and König, we propose a complete chain of test synthesis based on a true-concurrency model. This is done by revisiting the TGV methodology based on test purposes and of

graph manipulation (product, abstraction, projection, controllability). The theoretical basis of our proposal relies on sound and scalable algorithmic, based on the construction of prefixes of unfoldings. The main perspective is to continue the implementation of these ideas. We have also several research directions to explore:

- The use of UML as modelling language. Beyond its popularity, there is a real challenge to deal with the partial order semantics of UML, as given in the action semantics currently specified.
- The required algorithms seem to be implementable on-the-fly, like in TGV. But the situation is much more complex on unfoldings than in simple transition systems.
- There are specific questions of controllability in a distributed context [24], which deserve further study.
- Finally, it is tempting to refine the standard conformance relation based on sequential traces to a kind of partial order inclusion. This could be achieved by considering a distributed observation of the communication between the entities under test [25], i.e. by instrumenting the implementation by a vector clock mechanism.

5. BIBLIOGRAPHY

[1] ISO/IEC 9646 IT-OSI, OSI Conformance Testing Methodology and Framework

[2] M. Törö. Decision on Tester Configuration for Multiparty Testing. Proc. of the 12th int. Workshop on Testing of Communicating Systems. Budapest, Hungary, 1999.

[3] JC. Fernandez, C. Jard, T. Jéron and C. Viho. Using on-the-fly verification techniques for the generation of test suites, Conference on Computer-Aided Verification (CAV '96), New Brunswick, New Jersey, USA, Alur, A. and Henzinger, T. editors, Springer, LNCS 1102, july 1996.

[4] JC. Fernandez, C. Jard, T. Jéron and C. Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology, Science of Computer Programming, Groote, J.-F. and Rem, M. editors}, Elsevier Science, number 29, pages 123-146, 1997

[5] L. Doldi, V. Encontre, JC. Fernandez, T. Jéron, S. Le Bricquir, N. Texier and M. Phalippou. Assessment of Automatic Generation Methods of Conformance Test Suites in an Industrial Context, IFIP TC6 9th International Workshop on Testing of Communicating Systems, Baumgarten, B. and Burkhardt, H.-J. and Giessler, A., Chapman & Hall, september 1996.

[6] A. Kerbrat, C. Rodriguez, and Y. Lejeune. Interconnecting the ObjectGéode and CADP toolsets. In Proceedings of SDL forum'97. Elsevier Science (North Holland), 1997.

[7] T. Jéron, JM. Jézéquel and A. Le Guennec *Validation and Test Generation for Object-Oriented Distributed Software*. International Symposium on Software Engineering for Parallel and Distributed Systems (PDSE'98), Tokyo, Japan, April 1998.

[8] J. Tretmans, *Test Generation with Inputs, Outputs and Repetitive Quiescence*. Software, Concepts and Tools (1996) 17: 103-120.

[9] R. Tarjan. *Depth-first Search and Linear Graph Algorithms.* SIAM Journal Computing, 1(2):146-160. June 1972.

[10] T. Jéron and P. Morel. *Test Generation Derived from Model-Checking.* 11th intern. Conf. On Computer Aided Verification (CAV'99), Trento, Italy, July 1999. LNCS 1633, pp. 108-122.

[11] J. Engelfriet. *Branching Processes of Petri Nets.* Acta Informatica 28, pp. 575-591 (1991).

[12] J. Esparza, S. Römer. *An Unfolding Algorithm for Synchronous Products of Transition Systems.* Proc. Concur'99, Springer, LNCS 1664 (1999) 2-20.

[13] V. Khomenko and M. Koutny. *Towards an Efficient Algorithm for Unfolding Petri Nets.* Proc. Concur'2001. Springer, LNCS 2154 (2001): 366-380.

[14] Y. Ledru, L. Du Bousquet, P. Bontron, O. Maury, C. Oriat, and ML. Potet. Test Purposes: Adapting the Notion of Specification to Testing. To appear in the proc. of the IEEE Automated Software Engineering Conference (ASE'2001). San Diego, November 2001.

[15] RG. De Vries and J. Tretmans. *Towards Formal Test Purposes.* In Formal Approaches to Testing of Software (FATES), Aalbord, Denmark, August 2001.

[16] B. Caillaud, P. Caspi, A. Girault and C. Jard. *Distributing Automata for Asynchronous Network of Processors.* European Journal on Automated Systems (JESA), 31(3): 503-504. May 1997.

[17] C. Jard, T. Jéron, H. Khalouche and C. Viho. *Towards Automatic Distribution of Testers for Distributed Conformance Testing.* . Formal Description Techniques and Protocol Specification, Testing and Verification, 18, pp. 353-368. IFIP, Kluwer, November 1998.

[18] G. Luo, R. Dssouli, Gv. Bochmann, P. Venkaratan and A. Ghedamsi. *Test Generation with respect to Distributed Interfaces.* Computer Standards and Interfaces 16 (1994): 119-132.

[19] R. Castanet and O. Koné. *Deriving Co-ordinated Testers for Interoperability.* Protocol Test Systems, VI (C-19), O. Rafiq (Ed). Elsevier Science B.V. (North-Holland). 1994 IFIP.

[20] M. Kim, S.T. Chanson, S. Kang and J. Shin. *An Approach for Testing Asynchronous Communicating Systems.* Proc. of the 9th int. Workshop on Testing of Communicating Systems. Darmstadt, 1996, pp. 141-155.

[21] M. Kim, J. Shin, S.T. Chanson and S. Kang. *An Enhanced Model for Testing Asynchronous Communicating Systems.* Formal Description Techniques and Protocol Specification, Testing and Verification, 19, pp. 337-355. IFIP 1999. Beijing, China.

[22] A. Ulrich and H. König. *Specification-based Testing of Concurrent Systems.* Formal Description Techniques and Protocol Specification, Testing and Verification, 17. T. Mizuno, N. Shiratori, T. Higashino & A. Togashi (Eds.), 1997 IFIP. Published by Chapman & Hall.

[23] O. Henniger. On Test Case Generation from Asynchronously Communicating State Machines. Testing of Communicating Systems. Vol. 10. M. Kim, S. Kang & Al. (Eds). Chapman & Hall, pp. 255-271, September 1997.

[24] A. Ulrich and H. König. *Architectures for Testing Distributed Systems.* Proc. of the 12th int. Workshop on Testing of Communicating Systems. Budapest, Hungary, 1999, pp. 93-108.

[25] L. Cacciari and O. Rafiq. *Controllability and Observability in Distributed Testing.* Information and Software Technology 41 (1999): 767-780. Elsevier.

[26] http://wwwbrauer.informatik.tu-muenchen.de/gruppen/theorie/KIT/)