Monitoring and Testing with Case Observer Automata: an Industry Report

Anders Hessel
Xware AB and Mälardalen University
Xware AB, Sveavägen 151, SE-113 46, Stockholm, Sweden
IDT, Mälardalen University, P.O. Box 883, SE-721 23, Västerås, Sweden.
Email: anders.hessel@xware.se and anders.hessel@mdh.se

***Abstract*—In a highly configurable system, new or changed configurations may have to be tested in a running environment. To elicit high level information from such system, a monitoring solution must be able to: sort out the interesting data, connect the related data, draw conclusions about the data, and report the conclusion. The conclusions can be most helpful to verify that the new configuration has the desired functionality. To continuously monitor requirements and try to find specific erroneous pattern is helpful during testing but also after a deployment.**

In this industry report we present the Case Observer Automata modeling language (COAml) that aims to solve these issues. The COAml works with workflows where related data are put into cases, which can be seen as instances of the same observation pattern. In a case, parallel activities can be traced with synchronization points and timers. The Case Observer Language can be integrated into all forms of tracing/monitoring such as business rule monitoring, coverage tracing, static program analysis, log analysis, etc. We are currently integrating the language in the xTrade Alarm Server at Xware, and there are plans to reintegrate the COAml into UPPAAL CoVer.

***Keywords*-formal methods; observers; real-time; system surveillance; model verification; workflow monitoring; business monitoring; coverage critera, model-based testing, YAWL**

I. Introduction

A. Abstract and Combine Information

Xware[1] makes a product called xTrade Business Hub, which is a modular system for business communication. It is highly configurable and when a business integration is done with xTrade (mainly by configuration of xTrade system(s)) there is a need to test and "debug" the resulting system functionality.

Even if xTrade has built-in log possibilities it can be cumbersome to find and read logs to troubleshoot new configurations. To automate test cases with confirmation of an intended message path is possible but labor intensive, especially for a deployed system.

The xTrade Alarm Server is a monitoring and reporting tool, which can filter logs and system activities from a system into events. It can also combine already filtered events, i.e., combining triggered "alarms" in combination rules.

Our challenge is to *raise the abstraction level for the combination rules* into general observer automata separated in case instances. We want to be able to follow several workflows simultaneously. The workflows may be distinct business cases.

As a simple example we may want to detect if a computer takes too long time to come alive again after being shutdown. For this we need to filter out *up* and *down* signals for the computer. We also need a timer to expire when the stipulated time is due. As we do not want to make event filters for each computer, we add the name of the computer as an output parameter. At the combination rule we now have to put signals about the same computer in the same case.

Our motivations for doing this are:

1) to utilize the alarms as verdicts for test cases and
2) to be able to model requirements for which we can make observations.

B. Motivation of Workflows in Observers

Workflow models are particularly helpful when we have ordering constraints that must be satisfied. Let e_i denote an event, say that we receive a sequence of observations $(e_1, e_{21}, e_{22}, e_{31}, e_{23}, e_{32}, e_4)$ is this correct or not? What if we got $(e_1, e_{31}, e_{21}, e_{22}, e_{23}, e_{32}, e_4)$? Things clear up a lot when we are able to express the condition for checking the sequence as a workflow with parallel subflows see equation 1.

$$e_1 \left\{ \begin{matrix} e_{21}, e_{22}, e_{23} \\ e_{31}, e_{32} \end{matrix} \right\} e_4 \tag{1}$$

Now it is clear that the sequences e_{21}, e_{22}, e_{23} and e_{31}, e_{32} can be done in parallel, but must not start before e_1 and they must both end before e_4.

C. Contributions and Challanges

In this paper we present ongoing work with a language that we call Case Observer Automata modeling language, COAml, which has some features that we believe will make it easier to specify high level automata or workflows for business surveillance from an observational perspective. As

[1] http://www.xware.se/

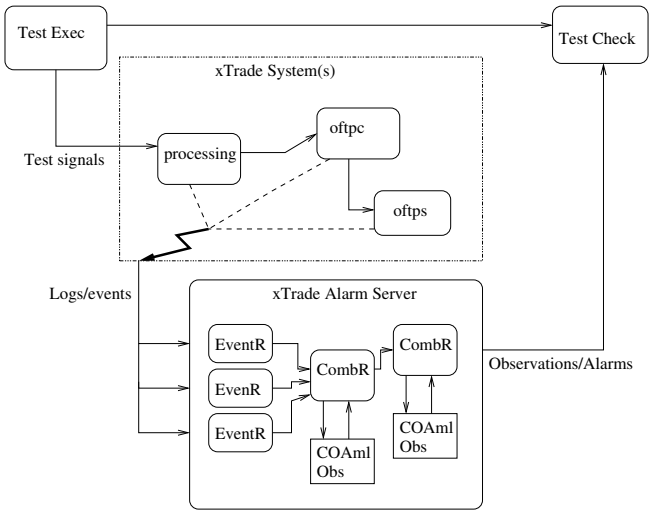

Figure 1. The test and monitoring environment

we can follow some of the internal activities in the system we are able to make a closer verification, i.e., we can make a kind of grey-box verification.

The contributions of the paper are:

- A method that make the verdict of a test case easier.
- A three step method for high level reasoning: filter, sort, and use automata.
- A new multi-purpose observer language with support for:
  - *Cases*; Collections of related states in cases.
  - *Multiple States*; Follows each of the possible paths which may result in multiple states. Thus if there is a path to a reporting point it will be found.
  - *Timers*; Timers can be set and removed for a case.
  - *Workflows*; Workflow concepts as multiple split of subflows and required synchronization points (merge of flows).
  - *State Removal*; Flushing of states according to matches.

II. Context

A. Test and Monitoring Setup

In figure 1 the test and monitoring setup for the an xTrade configuration is shown. The xTrade systems are stimulated with data. The figure includes a set of modules that are in use for the test case. A message that is passed through the system can be traced, as each of the modules that pass by the message will log its activity.

Below the xTrade system(s) the xTrade Alarm Server is depicted. The events traced from the system are filtered (with predicates on some fields) by *event rules* marked (EventR). In fact, part of the event rule filtering are done in the client to avoid unnecessary data traffic. If an event rule triggers, it selects parameters and sends out the result to its listeners which can be a *backend* or a *combination rule*. This procedure is common for all rules that triggers.

There are a variety of backends whose function is to send out a message or in another way inform the environment about the happening. A combination rule combines different notifications, i.e., triggered event rules or triggered combination rules.

In the figure we show automata combination rules with a COAml observer attached. Each observer accepts a language that is determined by the context of its combination rule. The notifications from other rules are sent to the observers as *trap*s. The inputs for a combination rule is defined by connecting *port*s from other rules. All report possibilities in the observer becomes a port of the combination rule.

OFTP is a communication protocol supported be xTrade. In the figure test signals are sent to an internal processing step that passes the message to an OFTP client that connect to an OFTP server on another xTrade instance. The different modules "logs" to the xTrade Alarm Server some of their activities. If the path taken by the message is correct the xTrade Alarm Server triggers the intended rule(s) and the verdict is easy for the oracle module *Test Check*.

B. The Environment of the Observer

We will use the term *monitoring application* to denote the immediate environment of the observer as we want to be as general as possible for our description or the COAML language.

Our observer framework implementation is a general purpose library whose API can be used in other products. We first developed the langugage to define coverage criteria and to be effective for test case generation.

The monitoring application defines the "run-time" interface for the observer work with, by (programatically) registering the possible traps. The observer specifications can then refer to registered traps and their parameters. The configuration done by the user is thus done towards the monitoring application. This makes it possible to have domain specific help in the monitor applications without loosing the generality of the COAML language.

III. COAML Description

In this section we will describe the COAML language. COAML is an observational language that is designed to be able to report when something interesting happens. The reports can be gathered and saved for, e.g., coverage detection, where a part of a coverage criterion has been fulfilled. The reports can also as in the xTrade Alarm Server case be directly sent to a recipient.

A running COAML system will be called an *observer* for simplicity. Basically an observer receives traps of a happening in the system/trace/code/model that it is observing. Observers also receive timeouts from the monitoring application, but we will consider timeouts only as a special

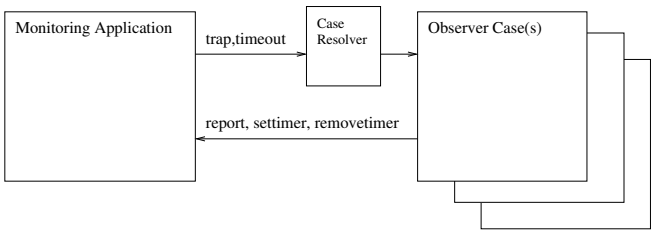

Figure 2. The environment in which the observer works.

type of traps. The observer change state when it receives at trap. The observer also as a side effect communicates its reports to the monitoring application. If the monitoring application is tracing coverage to facilitate test case selection then the states would be the unsatisfied information and the reports would be the fact that a coverage item was satisfied.

A. Case Abstraction

One of the features in the COAML is the ability to separate the activity of the observer into *cases*, i.e., when receiving a trap not the full observer state is working but only the relevant case(s). This is made possible by a case identifier and a case resolving procedure that will find the right case(s) and pass the trap to them. In figure 2 an observer is depicted with its environment. The *case resolver* decides to which observer case(s) the trap shall be sent.

For a case identifier with one field, we could have a trap with a parameter, e.g., named *var*, that is matched with the field of the case identifier. Say we trace definition-use pairs for a variable, then we could have the variable as the case identifier, thus the observer activities are split by the variable used.

The real benefit for the modeler is that there is no need to handle traps that is not related to the case and thus the model can be kept simple. In the xTrade Alarm Server there are also functions to force some specific instances.

Unresolvable traps: A trap that fails to fully specify the case identifier, e.g., it specifies one of two fields of the case identifier, will be sent to all cases that matches the existing field. The observer assumes that this is intentional. It must then be the case that each field in the case identifier uniquely specifies the case and thus only one case is selected. Moreover the first received trap must be specifying the full case identifier. We have an idea of an algorithm that would solve underspecified traps in general. So far we have not had the chance to implement and validate the algorithm.

B. Observer State

In the observer we have a set of *items* (markers in Petri Nets terminology, or "threads"), each item is placed at a location and has an evaluation for set of parameters (specific to the location). Such set of items and a set of timeouts defines a state of a case instance. The state of the full observer is a combination of the case instances.

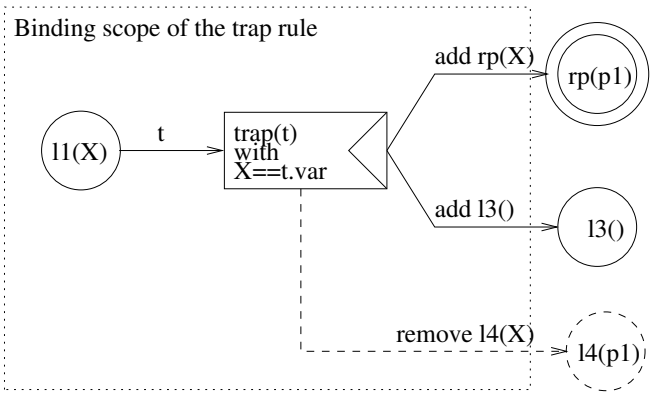

Figure 3. A graphical representation of a rule.

C. Observer Transitions

Transitions of individual states (items) in a case instance are done according to *rule*s. There are two types of rules, triggered rules and reduce rules. Triggered rules fire when a trap (or timeout) is received. Reduce rules fire automatically when it is possible. When a trap has been received and the trap rules have been exercised, reduce rules may be triggered. The set of reduce rules keep on firing until a fixpoint is reached, i.e., no reduce rule can be fired. Both rule types have conditions and actions.

In a transition the *condition*s also has the function of variable binding evaluation. A special type of "condition" is the *assign*, where a new variable can be assigned a value. There is always one value that is assigned although there can be several solutions. In the actions new items can be created and timeouts can be *set* or *remove*d.

1) Trap Rules: When a trap is received and delegated to a case instance, the items that are in a location which has an outgoing rule that trigger on the trap is examined. The conditions on the rules are evaluated, and if an item cannot satisfy any rule it stays in its location. If for an item a rule is triggered (i.e., the trap matches and the conditions if the rule is satisfied) by at least one rule the original item is removed. A triggered rule executes its action part, which may include actions to create new items, i.e., successors to the item. Observe that more than one rule can be triggered and each rule can *add* more than one successor. In the case where a rule replaces one item with several new, the COAML construction can be used as a workflow split.

In figure 3 a graphical representation of a trap rule is shown. The rule triggers when a trap *t* is received and there is an item in location *l1*, with the condition that the variable named *var* of the trap shall be equal to the first parameter of *l1* which is assigned the name *X* for use in the rule. There are three actions, add an item with the same parameter *X* to the reporting location *rp*, add an item to *l3* that do not have any parameter and remove, if it exists, an item in location *l4* which parameter equals *X*.

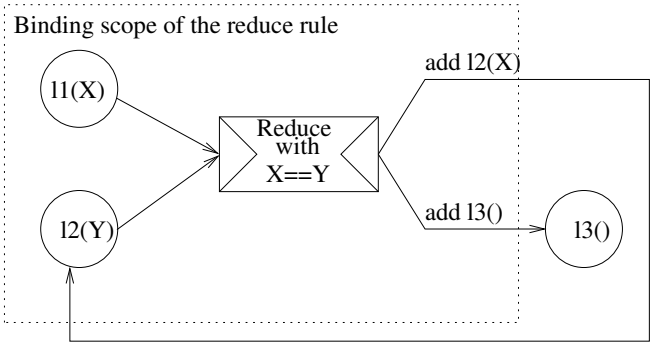

Figure 4. A graphical representation of a reduce rule.

D. Reduce Rules

Reduce rules take no input, instead they are triggered whenever possible after traps have been received and trap rules have been fired. There is a clear two phase division. No reduce rules are allowed to fire before all effects of a trap have been carried out. Reduce rules are tested in the order they appear. If a reduce rule fires then the reduce rules are tested again for the new set of items.

In figure 4 a graphical representation of a reduce rule is shown. The rule triggers when there are two items in the case (situated in location *l1* and *l2*) with equal parameters. There are two actions in this case, (i) to add the same *l2(Y)* back again (*X* equals *Y*), and (ii) to add an item in the location *l3* without parameter. A reduce rule that uses at least two items corresponds to a workflow join. Even if there is an immediately split (by adding multiple items) the reduce rule is still a synchronization point.

E. Some Keywords

To exemplify the rules of the language we use a textual syntax (as supported by our parser). There are some keywords to notice; the **rule** keyword starts a rule statement, the **trap** and **timeout** keywords followed by the id define the trap or timeout that triggers the rule, the **with** keyword starts a sequence of conditions, the **action** keyword starts a sequence of actions. In an action sequence the **add** keyword followed by a parameterized location adds a new item to the case, the **settimer** keyword followed by the timer id (with the length of the time until the timer expires as parameter) starts a timer and the **removetimer** keyword followed by the timer id removes a timer.

The **begincase** keyword starts a rule that is triggered only when the case instance is created. The **true** keyword can be used as a condition when no other conditions are needed. The **endcase** keyword destroys the case instance. If there are any items left when the endcase is reached the observer can be configured to report it as a problem.

We use a prefix "**ID.**" to address values in the case identifier and a prefix "**trap.**" to address values in the received trap. Assign is denoted as "**:=**" and equality "**==**".

IV. Examples

In the examples we will only show the transitions (rules) of an observer specification. There are also definitions of the locations (*node*s) with their parameter types. Additional the observer can take parameters which are bound to a set of predefined evaluations, i.e., the variables can have multiple values.

A. Simple Examples

Definition-use pair observer: In the definition-use pair example below we use two traps *def* and *use*. We assume that the traps have at least two parameters named *edge* and *var*. Further we assume that there is a case identity that has one field named *var*. Incoming traps *def* and *use* are directed to the case where their *var* field is matching the *var* field of the case identity, i.e., if the definition or use is concerning a variable x the case for that variable will be used.

Line 1 simply declare that the *nodef* location without parameters is the initial item added at instance creation. The rule that starts at line 2 accept *def* traps from items in location *nodef*. The variable *E* is free and assigned the value of the *edge* parameter in the received *def* trap. The edge (id) is remembered as a parameter of the *hasdef* location.

The rule that starts in line 5 is triggered by a trap *use* in a *hasdef* location. The variables *E2* and *V* are free and assinged the *edge* parameter of the trap and the field *var* of the case identifier *ID* respectively. We put back the item we started with (*hasdef(E)*). The *dupair* location is a report point (or satisfied coverage item) where the variable and the edge pair is reported.

```
1 begincase add nodef();
2 rule nodef() trap def
3         with E := trap.edge
4         action add hasdef(E);
5 rule hasdef(E) trap use
6         with E2 := trap.edge, V := ID.var
7         action add hasdef(E),
8                add dupair(V, E, E2);
9 rule hasdef(E) trap def
10        with E2 := trap.edge
11        action add hasdef(E2);
```

The rule started in line 9 simply switches the parameter of the *hasdef* location to the last edge where the variable was defined. We note that there is a way to include fields of the case identity in the report. If in- and out-of-scope traps are availabe we could report uninitialized use etc.

Downtime observer: In the next example we will show how to monitor the downtime of a service and report if it is too long. We assume that we have two traps *up* and *down* which both have at least one parameter that identifies the service (or computer) that is the subject of the trap, e.g., logs the up or down. The case identifier has a field name that identifies the service. We have in this example also a

timer called *dt* that is set to 5 minutes (the time unit is in seconds in this case) when we receive the knowledge of the service going down by the trap *down*.

In the rule at line 6 the service is up again. The timer is then removed and the case is closed. If on the other hand the timeout *dt* is received first three things are done (i) a new timer is started now with 10 minutes interval, (ii) the *waitup* node is put back, and (iii) we report that the service has been down too late by adding an *report* item. I.e., there will be a new report every 10 minutes.

```
1 begincase add init();
2 rule init() trap down
3         with true
4         action settimer dt(300),
5                add waitup();
6 rule waitup() trap up
7         with true
8         action removetimer dt,
9                endcase;
10 rule waitup() timeout dt
11        with C := ID.name
12        action settimer dt(600),
13               add waitup(),
14               add report(C);
```

B. Examples with Workflow and Reduction

A parallel workflow is shown below. At the initialization two parallel (sub)flows are started *start1* and *start2*. The flows are waiting for trap *t1* and trap *t2* respectively. Although the flows are short the principal should be clear. The *reduce* rule can be triggered first after each subflow is done, i.e. the subflows has to be in sync to progress.

```
1 begincase add start1(), add start2();
2 rule start1() trap t1
3         with true
4         action add done1();
5 rule start2() trap t2
6         with true
7         action add done2();
8 reduce done1(), done2()
9         with true
10        action endcase;
```

In our last example we assume that the observer has a parameter *V* bound to a set of three different values, thus the *add start3(V)* command generates three items The trap *t* has a parameter *var* and will be received three times with each value from *V*. After each trap the reduce rule at line 5 will be activated. The result is that a *done* location will be removed and the *collect* locations parameter will be decreased. After receiving the three values the reduce rule in line 8 will be triggered. For n values we have $n!$ number of legal sequences, thus the parallel expression power has been shown useful.

```
1 begincase add start3(V), add collect(3);
2 rule start3(X) trap t
3         with X == trap.var
4         action add done(X);
5 reduce done(X), collect(N)
6         with N > 0, M := N - 1
7         action add collect(M);
8 reduce collect(N)
9         with N == 0
10        action endcase;
```

V. Related Work

COAML[2] is a development of the Observer Language [1] used in UPPAAL CoVer [2], [3][3] that is an extension of UPPAAL[4]. This observer language makes it possible to express structural, data-flow, and projectional coverage criteria. There is also inspiration taken from YAWL [5] (Yet Another Workflow Language), which is a workflow language developed from (Colored) Petri Nets [6], [7]. YAWL seem to become an important language for workflow modeling. As an example there is work on transforming the Business Process Execution Language (BPEL) to YAWL[8].

A. Comparison of Concepts YAWL/Petri Nets, Observers, and Case Observers

We will refer to the coverage observers as Observers.

Places and Markers: Places in YAWL/Petri Nets are the same as locations in Observers/COAML. Markers in YAWL/Petri Nets are the same as items in COAML and unsatisfied coverage items in Observers. In YAWL/Petri Nets multiple identical markers is possible this is not the case in Observers/COAML.

Startup: In Observers each state change in the model can initiate a new item from a special state called **start**. In YAWL/Petri Nets there is one start state called **START** for the whole workflow. In COAML each new case instance run the actions in the *begincase* rule, and thus several items can exist when the first trap is received (which also can be a timeout).

Timers: YAWL has a time service and task timers, COAML has timers per case. Observers lack the concept of time.

Instances: YAWL has multiple instance tasks (atomic and compound), COAML has Cases. Directing observations is not an issue in YAWL as it is the YAWL instance task that is the object under observation.

Split and Join: YAWL has AND (all), OR (some), and XOR (one) splits, in COAML all these three are defined by the action part of trap and reduce rules. YAWL is more specific. In COAML multiple rules can trigger by the same trap which makes other kinds of behavior possible. YAWL has AND (all) and XOR (one) joins, but also the more

[2]http://www.hessel.nu/, (anders@hessel.nu)
[3]http://www.uppaal.org/CoVer/, (cover@hessel.nu)

complex OR join where all markers in (the OR split to OR join) workflow has to be collected. In COAML the reduce rules is the same as AND join, the XOR join is done by *add*ing the same location from the different subflows (which can only be one if the split is excluded OR by conditions).

Observations: In COAML one trap can trigger many rules, whereas one YAWL task is just one task. If no condition is true for the trap COAML reports an non-conforming error. COAML is made to make observation and report higher order observations. YAWL is made to define a workflow.

Reset: Reset Petri Nets and YAWL may reset all markers from a subset of the places. In COAML the remove action can be used to delete items. To match multiple items in a location some parameters can be set to a wildcard.

VI. Application Areas and Future Research Collaborations

There are several application areas for the proposed language COAML. On the one hand it can be used to monitor an abstract system. On the other hand it can be used in quite intimate tracing, it all depends on which kind of traps the monitored system can produce.

The observers have an origin in the model-based testing world where they have been used to specify and to observe coverage criteria. In that case a test case generation engine generates a test suite that fulfills the given coverage criterion. The reporting of the observer is used as guidance for the test case generation. Another area that is currently targeted for the (case) observers is system surveillance, also including business monitoring. In fact, as the observers are meant from the start to be non-intrusive there is no difference between observing a running system and observe the development of a search branch for a model checker. The observers can also be used in a model checker as a verification property.

Other possible application areas are log browsing, which is also a form of pattern matching as well as (static/dynamic) code/model analysis, e.g., looking for uninitiated variables or race conditions. For instance a code analysis tool could use a configurable number of observers to find flaws in the code. Implementing a new flaw type could be done by defining a new observer automaton.

Currently the development is done in parallel for a commercial product and for an academical tool. As the APIs now are more well defined than before and we seek colaboration partners that have use of observer libraries. Our ambition is that the COAml language and our libraries shall be the number one choice when there is a need to build a tailored tool that needs observational capabilities.

VII. Summary

We have shown how the COAML is used and how it lets its users specify parallel sections with concurrent activities without raising the complexity of the observer. The case solving mechanism separates the unrelated traps. The related traps are sent to the same case. Within the cases there can be several interacting activities, suitable for surveillance of parallel workflows.

Acknowledgements

The author wish the thank Hi5 Mobility Program and Xware AB that have sponsored this research project. Many thanks also to the anonymous reviewers for improving the text.

References

[1] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson, "Specifying and generating test cases using observer automata," in *Proc. 4^{th} Int. Workshop on Formal Approaches to Testing of Software 2004 (FATES'04)*, ser. LNCS, J. Gabowski and B. Nielsen, Eds., vol. 3395. Springer–Verlag, 2005, pp. 125–139.

[2] A. Hessel and P. Pettersson, "CoVer - A Test Case Generation Tool for Real-Time Systems," in *Testing of Software and Communicating Systems: Work-in-Progress and Position Papers, Tool Demonstrations, and Tutorial Abstracts of TestCom/FATES 2007*, ser. LNCS, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds. Springer–Verlag, 2007, pp. 31–34.

[3] A. Hessel, K. G. Larsen, M. Mikuionis, B. Nielsen, P. Pettersson, and A. Skou, "Testing real-time systems using uppaal," in *Formal Methods and Testing*, ser. LNCS, R. M. Hierons, J. P. Bowen, and M. Harman, Eds., vol. 4949. Springer–Verlag, 2008, pp. 77–117.

[4] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a Nutshell," *Int. Journal on Software Tools for Technology Transfer*, vol. 1, no. 1–2, pp. 134–152, 1997.

[5] W. van der Aalst and A. ter Hofstede, "Yawl: Yet another workflow language," in *Information Systems*, vol. 30(4), 2005, pp. 245–275.

[6] K.Jensen, "Coloured petri nets: A high level language for system design and analysis," in *Advances in Petri Nets*, ser. LNCS, G. Rozenberg, Ed., vol. 483. Springer–Verlag, 1990.

[7] F. Gottschalk, M. H. Jansen-vullers, and H. M. W. Verbeek, "Protos2cpn: Using colored petri nets for configuring and testing business processes," in *In Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN*, 2006.

[8] A. Brogi and R. Popescu, "From bpel processes to yawl workflows," in *WEB SERVICES AND FORMAL METHODS*, ser. LNCS, M. Bravetti, M. Nez, and G. Zavattaro, Eds., vol. 4184. Springer–Verlag, 2006, pp. 107–122.