Protocol-safe Workflow Support for Santa Claus

Rick van Rein

University of Twente, Dept. of Computer Science
Postbus 217, 7500 AE Enschede, the Netherlands, vanrein@cs.utwente.nl

Abstract. Practical software analysis techniques exploit a form a process description, mostly in some flavour of state diagram. Unlike typing information, these process structures are usually not passed down to the implementation level, and neither are they exploited in any form of consistency check. It is our belief that the information in most designs suffices to perform all sorts of consistency checks.
This workshop paper studies a simple case where workflow processes interact with 'actual' objects at the implementation level, and demonstrates how useful protocol checking can be in making and keeping these processes consistent with each other.

Keywords: Protocol checking, object orientation, workflow processes, model checking, program verification, Paul.

Introduction

During object oriented analysis, objects are partially modelled with some state diagram formalism [Har87] [Cor97]. This facilitates understanding of the order in which objects may participate in operations. With a few exceptions [SGW94] [Kri94] including our own work on the protocol checker Paul [RF99], it is not customary to exploit this form of process information in consistency checks.

During workflow analysis, business processes are captured in workflow charts. It is common for workflow systems to perform consistency checks, exploiting formal process notions such as Petri nets [AH97] or Spin [Hol97].

Workflow processes often act as clients of a set of implementation objects, and if the process knowledge of both notions is brought together, it is possible to check whether they are compatible. We perform a process check, by translating workflow processes into our protocol assuring universal language Paul. Although several other process aspects are interesting to consider, we limit our current presentation to checking protocols, or invocation orders.

For readability of this presentation, we introduce a graphical notation to describe workflow processs. This notation is inspired on Petri nets, where fat bars indicate 'places' where an object (or 'token') rests between actions (drawn as rounded rectangles). We annotate each place with the type of object held there, and its state at that point. An object that only flows out of an action is created in that action, an object that only enters the action is deleted; So in

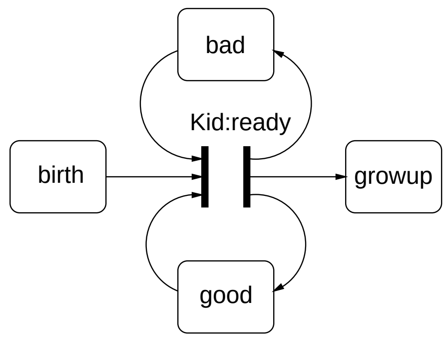

Fig. 1. The Kid workflow.

Fig. 1, the birth action creates an object of class Kid in state ready, and growup consumes such an object.

In addition to standard Petri net concepts, we allow communication between our workflows. This is drawn as a place with only incoming arrows (let a workflow take over the incoming objects) or a place with only outgoing arrows (take over the outgoing objects from workflows). The places in Fig. 1 are drawn separately to indicate that a Kid in state ready may be taken over by other workflows.

A workflow model can in general refer to multiple objects, and this matches well with the notion of multiple *import roles* in Paul. To perform protocol checking, we map each workflow process onto a class in Paul, with one import role for each object accessed; The imported protocol for an object is the projection of the workflow process on that object.

Implementation objects are assumed to already be described in terms of Paul objects. It is not hard [RF99] to generate Paul classes from state diagrams. We assume that different workflows each access a different *export role* of a Paul class.

The last modelling step is to associate the workflow classes' imported roles with the corresponding implementation objects' exported roles and let Paul search for protocol inconsistencies. For each discovered protocol error, Paul generates a helpful error message, including the trace that led to the error. It has been our experience that this feedback is very helpful in debugging the protocol aspects of software.

The remainder of this paper walks through a case study demonstrating this process of workflow/implementation verification.

1 Business Process Analysis

Case description. We have performed a case study on a non-profit organisation headed by a certain *Santa Claus*, located at the North Pole. Santa employs hundreds of blue-collar workers named *elves*. His market comprises of kids, who are classified as either good or bad . These kids send in an annual wish list, but their wishes are only granted when their classification reads good. The wishes usually are requests for toys, which are constructed in-house by Santa's elves. For simplicity, we assume that each wish list requests only a single toy.

Due to the growth of the world population, Santa decides to automate much of the administration by installing a workflow system. The workflow in this business is described with a number of interacting workflow processes.

Kid workflow. The Kid workflow in Fig. 1 shows that a kid is born, may be marked as good or bad with judging actions, and an action growup terminates the workflow of this kid in this system. In the mean time, this kid can be taken over by any other workflow process at any moment, as shown with the places.

Present workflow. The Present workflow in Fig. 2 shows that the processing of presents starts with a wish list coming into the system with action listinput. The judge action combines such a list with a kid, and dependent of whether the kid is classified as good or bad, either the kid is put back in ready state and the list will be discarded, or the kid will be output in accepted state and the list in order state. We see that the wrap action can continue as soon as an object of class toy becomes available in state made and the list in state order. Finally, there is a deliver action in which the presents are dropped off at the good kid.

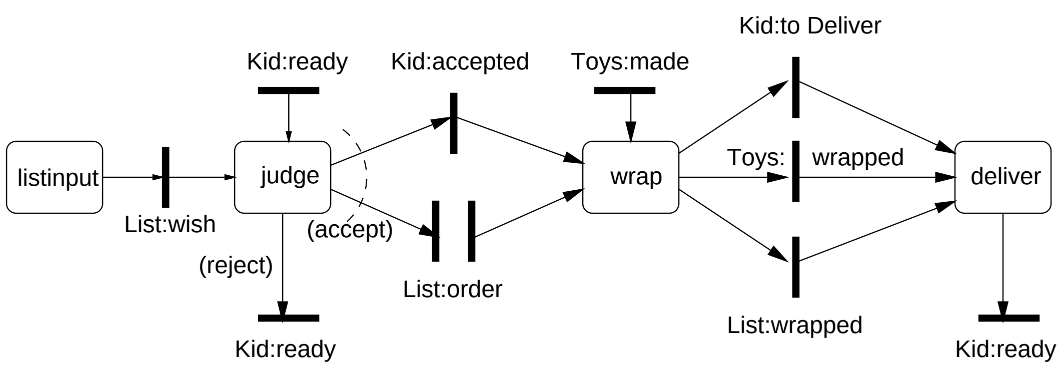

Fig. 2. The Present workflow.

Protocol checking. To be able to perform protocol checks on these specifications, we describe them in Paul, our Protocol Assuring Universal Language.

The object model of Paul explicitly describes the protocols imported by each class. So, we can map each workflow onto a class in Paul with an import role for each variable that it uses. Without specifying it here, we assume a reasonable binding of workflow actions (rounded rectangles) to import role invocations. For the aforementioned two workflows, we write the following code fragments, modelling the Kid workflow as a 'manual' process, where each action is called from the external world, and modelling the Present workflow as an 'automatic' process, resulting from the listinput action:

```
class KidFlow: birth.(good+bad)*.growup is
  imports kid: (good+bad)*
  . . .
  method good () is
    invoke kid.good ()
  end
  method bad () is
    invoke kid.bad ()
  end
  method birth () is
    // Nothing due to static binding
  end
  method growup () is
    // Nothing due to static binding
  end
end

class PresentFlow: listinput is
  imports kid: judge.accept.deliver+judge.reject
  imports list: reject+accept.wrap.deliver
  imports toys: wrap.deliver+decline
  . . .
  method listinput () is
    // listinput process: no action on implementation objects
    if kid.judge () then // nondeterministic (non-process) choice
      // (reject) branch in workflow
      invoke toys.decline ();
      invoke kid.reject ();
      invoke list.reject ();
    else
      // (accept) branch in workflow
      invoke kid.accept ();
      invoke list.accept ();
      // wrap process
      invoke toys.wrap ();
      invoke list.wrap ();
      // deliver process
      invoke list.deliver ();
      invoke kid.deliver ();
      invoke toys.deliver ();
    end
  end
end
```

The notation, which is equivalent to state diagrams, uses a . for sequential composition, a + for choice (by the server if the same starting action occurs in both sides) and * for looping.

As stated in the introduction, we assume a Paul description for the implementation objects that are referred to in the workflows. The objects that we model here are Kid, List and Toy. Some of these export more than one role, and to ensure that the interleaving of the roles matches the implementation's abilities, there is also a *class protocol* which details how roles may get blocked while awaiting messages over other roles. We write the following code fragments in Paul:

```
class Kid: (good.(judge.accept.deliver)*+bad.(judge.reject)*)* is
  exports behaviour: (good+bad)*
  exports presents: (judge.accept.deliver+judge.reject)*
  ...
end
class Toy: order.make.(wrap.deliver+decline) is
  exports process: wrap.deliver+decline
  ...
end
class List: reject+accept.wrap is
  exports process: reject+accept.wrap
  ...
end
```

The last thing to do is tell Paul how the import roles of the workflow classes must be related to the export roles of the implementation classes; We use — to associate qualified roles of the form *classname*:*rolename*

```
KidFlow:kid       — Kid:behaviour
PresentFlow:kid   — Kid:presents
PresentFlow:list  — List:process
PresentFlow:toys  — Toy:process
```

Given these specifications, Paul is able to perform protocol checking, and we are quickly informed of our first mistake:

> Error: The "association PresentFlow:list → List:process" relation is not protocol safe: ⟨accept.wrap.deliver...⟩

This indicates that Paul found an inconsistency in the design; The trace between angular brackets leads to a protocol error. Apperantly, an error can occur in relation to an action deliver after the actions accept and wrap have taken place. And indeed, the Present workflow invokes the action deliver as a final action to a list, while the Paul specification of the associated List:process role does not support it.

This is a good example of how Paul works: It checks the consistency of a design with respect to protocols. Our experience is that Paul finds conceptual flaws in designs, just like type checkers do, but focussed on another class of conceptual errors. The counter-examples generated by Paul help locating the flaw.

The only thing reported by Paul is an inconsistency between parts of the design; It is now up to an intelligent being to see which part in the inconsistency is the 'wrong' part. In this example, it seems reasonable to discard a list when toys have been wrapped (and labelled with the kid's name), so we remove the place List:wrapped from the Present workflow in Fig. 2, *and thus* also the deliver action from the PresentFlow:list import protocol and its invocation in the listinput method body.

The checks performed by Paul inform us of yet another flaw in the design:

Error: The "export from Toy" relation is not protocol safe: ⟨decline(1)...⟩

The counter-example mentions a problem when the first export role (the (1) annotation) invokes the decline action. Because this error is related to an export role, it must indicate that the class protocol and export protocols do not co-operate well. And indeed, looking at the example we find that the Toy class protocol enforces starting with a make and an order operation, which are not called by any export class!

This is the kind of error where an implementation (represented by a class protocol) does not implement offered behaviour (represented by export protocols) in a protocol correct way. In examples less trivial than this one, it is *very* convenient to have this checked.

In this case, we decide that the order and make action are indeed something that the Toy should be aware of, and we decide to add a new export role to the Toy class:

exports production: order.make

Toy workflow. Now, if we run Paul again, it complains about the dangling Toy:production role; We clearly must access it from some workflow. We choose to design an additional Toy workflow, see Fig. 3. This workflow awaits a List in order state (meaning it is accepted by the Present workflow) and then produces a toy for it. After the addition of this workflow, Paul accepts the design. This means that, regarding action ordering, there are no protocol errors in the design, and Santa is ready to start working with it.

2 Business Process Reengineering

In the previous section we showed how a simple tool such as Paul helps to ensure protocol consistency in a design. One useful way to apply this principle is for making parts of the design more dynamic within the constraints imposed by the consistency.

The system that we designed for Santa Claus is rather inefficient. The turn-around cycle from wish list to toy delivery is about a month, and there are

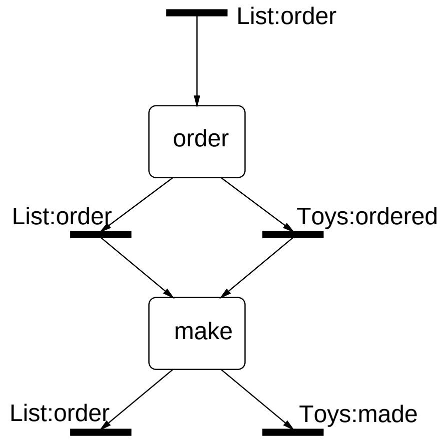

Fig. 3. The Toy workflow.

strict delivery deadlines. Furthermore, all wishes come in at the same moment, resulting in unemployment for the elves during eleven months a year. Since Santa resides on the North Pole (unlike his Dutch collegue Sinterklaas, who lives in sunny Spain) this is waisted time that can be put at better use.

Therefore, we redesign our Santa support system so that stocks of toys can be built up during the year. To do this, we creatively revise the Toy workflow to the one in Fig. 4, assuming some action order to order toys before a wish list requests them. After being made, toys are stocked until they can be paired with a wish list, with the select action; Note how this action does not change the state of either List or Toy, so that we need not model its influence on these objects in Paul.

The result is a new workflow system which is more efficient, and which exploits the unchanged implementing objects as the original system. This is the kind of thing that business process reengineering aims for. The added value of Paul to business process reengineering is that it disapproves of incorrect designs and provides sufficient feedback to repair them. This is particularly of interest because business process reengingeers are not always technically skilled people. The consistency checks make the processes in the Santa system reconfigurable without loss of integrity.

3 Related Research

The major reason for not checking object process consistency lies in the lack of a semantics for popular methods [Cor97]. Work done in that direction [LMM99] suggests that state chart models are also very complicated, which makes them less suitable for formal verification.

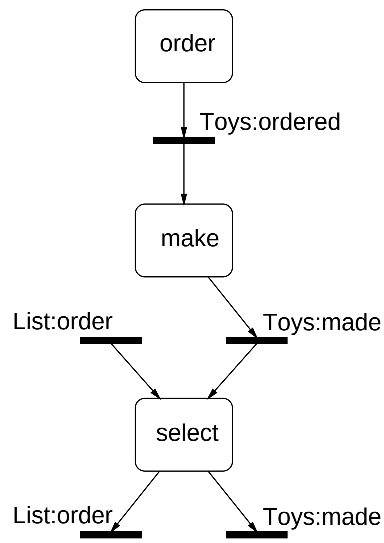

Fig. 4. Reengineered Toy workflow.

Some methods exist with more precise semantics, and with a simpler process model that still seems to suffice. Although ROOM [SGW94] and its tool support provide consistency checks on process models, it aims at (embedded) realtime systems and is therefore unlikely to connect to workflow systems. ROOM describes systems with known numbers of instances, and although Paul 1.0 suffers from the same problem, we are working on extensions with creational primitives.

This work is based on our work on Paul. An implementation of Paul and the full sources for this case study can be downloaded from ftp://ftp.cs.utwente.nl/pub/doc/Quantum/Paul. For an online tutorial on Paul, please visit http://www.cs.utwente.nl/~vanrein/paul/learn/tract.

Finally, our current work includes a graphical notation for *life cycles*, a process construct meant to express both workflow and implementation-object behaviour in a way close to state-diagrams. We intend to extend the protocol approach checking described here for these processes as well.

4 Conclusions

We have demonstrated our protocol checker Paul in a workflow setting. It served well to ensure consistency between workflow diagrams and implementing objects, also in a setting where the workflow parts of the design are made dynamically changeable. We believe this to demonstrate the usefulness of representing process information in both a design *and* its implementation.

This work has been performed in the scope of the Quantum project, in which Compuware's UNIFACE lab and the University of Twente cooperate.

References

[AH97] W.M.P. van der Aalst and K. van Hee. *Workflow management: modellen, methoden en systemen*. Academic Service, 1997.

[Cor97] Rational Software Corporation. *UML Semantics*. Rational Software Corporation, 1997.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[Hol97] G.J. Holzmann. The model checker Spin. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[Kri94] G. Kristen. *Object Orientation: The KISS Method: From Information Architecture to Information System*. Addison-Wesley, 1994.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a formal operational semantics of UML statechart diagrams. *Formal Methods for Open Object-Based Distributed Systems*, pages 331–347, 1999.

[RF99] R. van Rein and M. Fokkinga. Protocol assuring universal language. *Formal Methods for Open Object-Based Distributed Systems*, pages 241–258, 1999.

[SGW94] B. Selic, G. Geullekson, and P.T. Ward. *Real-time Object-Oriented Modeling*. John Wiley & Sons, Inc., 1994.