Implementation of Workflow Systems using Reference Nets
—
Security and Operability Aspects

Thomas Jacob[a], Olaf Kummer[b], Daniel Moldt[c], and Ulrich Ultes-Nitsche[d]

[a]Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, e-mail: mail@ThomasJacob.de
[b]CoreMedia AG, Erste Brunnenstr. 1, 20459 Hamburg, e-mail: olaf.kummer@coremedia.com
[c]Fachbereich Informatik, Universität Hamburg, Vogt-Kölln-Str. 30, 22527 Hamburg, Germany, e-mail: moldt@informatik.uni-hamburg.de
[d]Department of Electronics and Computer Science, University of Southampton, Southampton, SO17 1BJ, United Kingdom, e-mail: uun@ecs.soton.ac.uk

Abstract. We present in this paper a generic approach to designing and running workflows in a flexible way. Our language of choice is reference nets, a version of high-level Petri nets that allow creation of new net instances as well as communication between net components in an object-based way.
We discuss the general approach to workflow design as well as the specifics of reference-net-specified workflow: their flexibilities and, related to this, security aspects of reference-net workflows. Reference nets allow the implementation of even elaborate security concepts for workflow systems in an elegant and simple way.

1 Introduction

To support complex business processes, workflow systems are promoted by various institutions and IT companies. Their aim is to use information technology to manage the different tasks that must be completed in order to complete a business process successfully. In addition, workflow systems must deal with incomplete tasks, missed deadlines, etc.

Various approaches have been proposed for describing workflows. We present in this paper a Petri-net-based approach, using reference nets [6]. Reference nets allow object-based design of a workflow, and the run-time environment Renew (REference NEt Workshop) [6] can be used as the basis for the workflow engine executing reference-net workflows. Some of the modelling concepts have been introduced in [10]. However, support by a tool was missing.

We discuss how the Renew tool has been extended to create a workflow engine [3]: A *task symbol* has been introduced to reference nets to allow for a more compact representation of workflows. The task symbol does not at all change

the abilities of reference nets as they can be easily compiled to a three-transition reference-net structure. In fact, tasks only occur on the graphical level of the Renew editor, but are resolved into the aforementioned reference-net structure in the internal representation of reference-net workflows. The workflow engine, as an extension to the Renew simulation engine, can handle tasks and the way how users select and perform tasks.

In addition, we present how the system, supporting the general concepts of reference nets, can be used to implement elaborate access controls to tasks and data. To each task, we propose the implementation of a task rule, which is simply a Java method that evaluates to type `boolean`, deciding whether or not access to a task will be granted (see [3] for details). This will enable us, for instance, to implement simple role-based access controls as well as sophisticated content-dependent ones [8] such as "need-to-know" access controls [1]. Having defined a generic approach, we can integrate any available information in the access control decision procedure.

Our paper is structured as follows: After a brief introduction of reference nets using an example taken from [6] in the next section, we introduce workflow specifications in the third section. Section 4 contains a discussion of the workflow engine, followed by Section 5 on workflow security. An example in section 6 illustrates some features for our workflow engine.

2 Reference Nets

Reference nets are high-level Petri nets that implement the nets-within-nets concept (see [9]) and incorporate Java annotations. In [6] a formal definition is given. The Java annotations control enabling a transition and can create side effects when a transition is fired. Firing a transition can also create a new instance of a subnet in such a way that a reference to the new net instance will be put as a token into a place. A net instance can communicate with the subnets it has created if it possesses a reference to the subnets. Communication occurs through synchronous channels associated with transitions. A net instance can also communicate with nets whose subnet it is. Subsequently, we introduce the different concepts present in reference nets using an example taken from [6]. We assume the reader's familiarity with the general notions of place/transition nets (P/T nets) and coloured Petri nets.

We start with the simple P/T net depicted in Figure 1. It describes two people who either work at home or in the office, earn money when communicating with one another while both being in the office and spend money by commuting to and from the office.

If we move in this example from P/T nets to coloured Petri nets [4], we can fold the two subnets representing `A` and `B` into a single coloured Petri net, which is presented in Figure 2.

To introduce the first concept, synchronous channels (see [5] and [2]), in this paper we extend our example slightly: We separate the "money storage" part

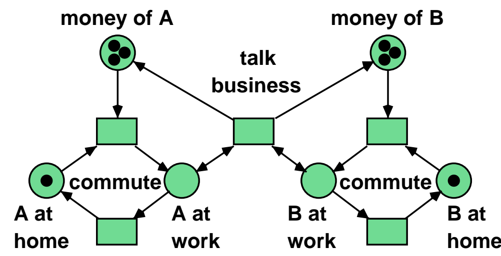

Fig. 1. P/T net example: Co-operation of two business people.

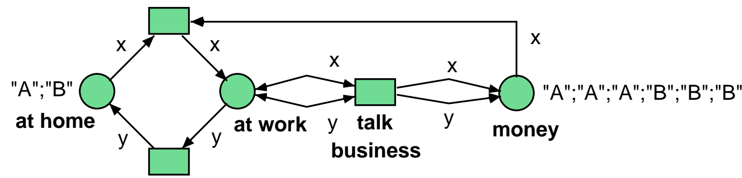

Fig. 2. Coloured Petri net version of the example.

from the "working life" part in our net by creating a simple bank account model. The model is shown in Figure 3.

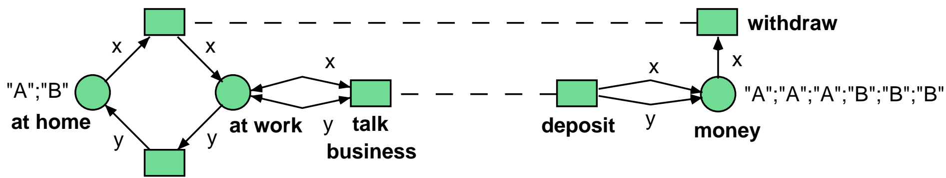

Fig. 3. Separation of "working life" and "money storage".

The model with separate account and person specifications will not provide the functionality of our initial simple model without the definition of some means of co-ordination between the two separate nets. The co-ordination concept we use is communication via parameterised synchronous channels. In this concept, one transition "owns" a channel, which is labelled with `:channelName(parameters)`. Other transitions can be synchronised with transitions using the channel by labelling them with `netName:channelName(parameters)`. If the channel is used for synchronisation within the same net, `netName` will be `this`. Otherwise `netName` is the name of the net, which contains the transition owning the channel. A transition owning a channel is enabled if and only if it is enabled in the usual sense

as well as all transitions elsewhere whose label refers to the channel are enabled in the usual sense. This includes taking into account the value bindings of variables that can occur as channel parameters. A transition using but not owning a channel will be enabled whenever the transition owning the channel is enabled. Enabled transitions sharing a channel will fire at the same time, i.e. in a synchronous fashion. Figure 4 shows the net from Figure 3 extended by channels `deposit(_,_)` and `withdraw(_)` to co-ordinate the two subnets.

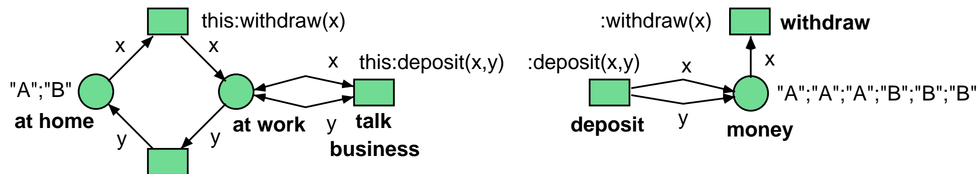

Fig. 4. Introducing synchronous channels.

By using a channel twice in a transition, we can enforce firing another transition two times as a kind of atomic event. Figure 5 shows this concept in a net, which is behaviourally equivalent to the one in Figure 4.

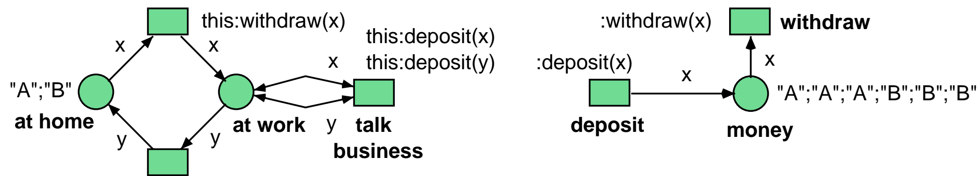

Fig. 5. Multiple channels at one transition.

Instead of dealing with the multiplicity of similarly named tokens (e.g. "`A`") to represent `A`'s credit balance, we will be dealing with integer values. This idea is introduced in Figure 6.

Having introduced synchronous channels will allow us to create instances of subnets and to communicate with subnets in an object-like way. Using this concept enables us to use different instances of the same `account` net to handle the accounts in our example separately. A new instance of a subnet is created by using the construct `name:new netName` where `new` is a keyword and `name` and `netName` are a reference to the new instance of the net and the name of the net, respectively. Figure 7 shows the `account` subnet and Figure 8 the (main) `person` subnet of our final specification.

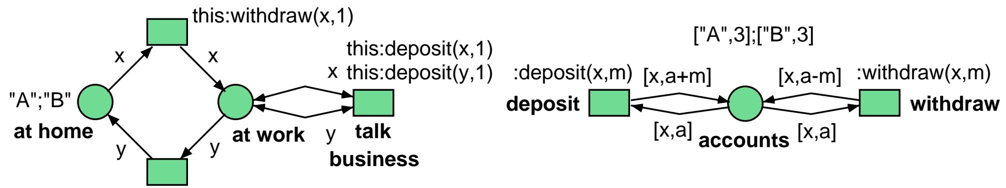

Fig. 6. Introducing integers.

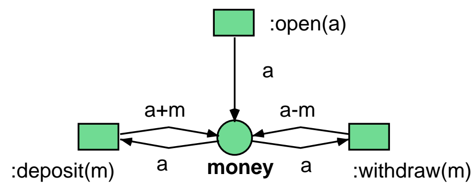

Fig. 7. Description of an account.

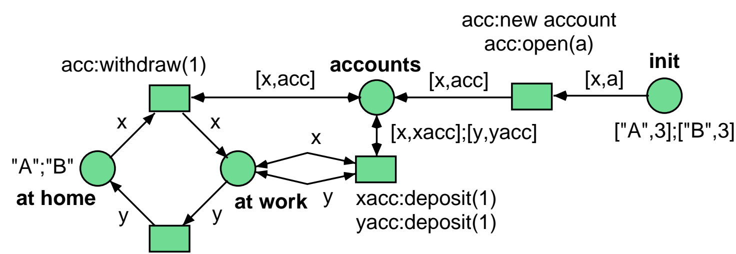

Fig. 8. Description of `person`.

Initially, two instances of net `account` will be created and references to the two instances will be stored as part of the tokens in place `accounts`. Communication with the net instances occurs as before via named, parameterised synchronous channels.

Presenting Figures 7 and 8 completes our brief overview of the capabilities of reference nets. It should be mentioned, finally, that reference nets allow for Java annotations (objects and methods) as net inscriptions, resulting in a flexible modelling language for workflows. The reference net tool Renew (REference NEt Workshop), extended in a suitable way, will then provide us with a runtime environment for workflow processes.

3 Workflow Specifications using Reference Nets

The aim of a workflow specification is to unambigously describe the *control flow* as well as the *tasks* that together implement a business process. Tasks are the atomic work items that need to be completed in order to successfully complete the business process. Control flow is the co-ordination process that controls the order of tasks to be performed, including dependencies between the tasks and their possibly various outcomes.

Petri nets, and reference nets in particular, are a very natural approach to describe the control flow in a workflow: Their strength is indeed the description of the flow of information through the net's structure and, by the firing rules of transitions, to describe dependencies between various events. Even though it is fairly evident that tasks will be somehow related to transitions, as transitions are the actions in a Petri net specification, the atomicity of transitions is slightly too strict for mapping tasks onto transitions directly. We will clarify this statement subsequently, which will lead to the definition of a task object, which we then will map onto a net structure formed by three transitions.

Before discussing its formalisation, we first have to clarify what the basic ingredients of a task are in a workflow. In crude terms: a task is an atomic unit of work that will be completed by an individual. Its support can range from completely manual (no support at all) to complete automation (no involvement of an individual but only software). The result of a task can be either positive or negative. In the positive case, the task will be completed, usually creating some kind of output which will then be provided to control the workflow and subsequent tasks. In the negative case, the task will be terminated unsuccessfully, a case in which the state of the workflow has to be reset to the one prior to the attempt of performing the task. So, even though a task is atomic within a workflow, it has some structure from the modelling point of view.

We will therefore model a task using three transitions representing *entering* the task as well as either *success* or *failure* of completing a task. Firing the entry transition will create an instance of a task-related object, which could include instantiation of a sub-net, creating e.g. a sub-workflow, instantiation of a software object needed to deal with the task, and extension of a to-do list of work items. When the task is completed, its result will tell whether the

success or failure transition has to fire. There are no time constraints imposed on the duration between entering and leaving the task. We hence have inherently solved an additional requirement for modelling tasks: performing tasks requires time. The lower part of Figure 9 shows the basic structure of a task construct in our reference-net-based modelling technique. Please note that the topmost entry places are chosen arbitrarily as well as the output places at the bottom of Figure 9.

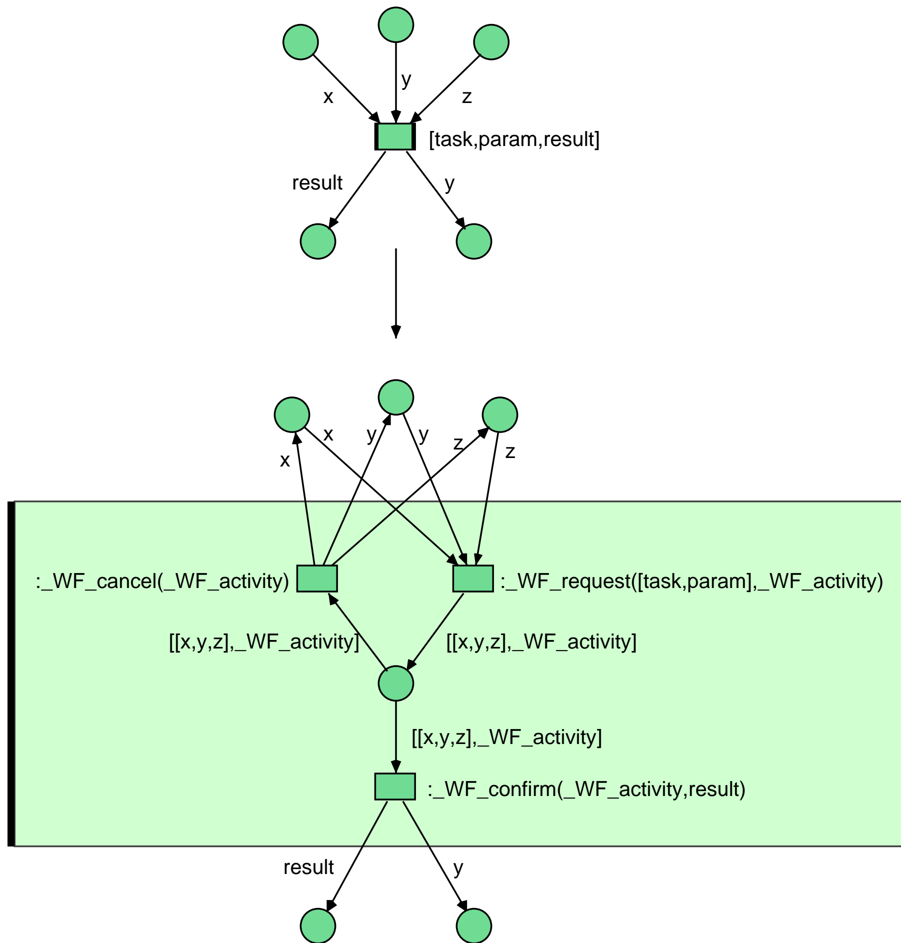

Fig. 9. Specification of a workflow task.

`_WF_request([task,param],_WF_activity)` is the transition that will be fired to invoke the task. In case of an unsuccessful task termination, transition `_WF_cancel(_WF_activity)` will fire. Note that `_WF_cancel(_WF_activity)`

will put back all tokens initially taken from the entry places when firing transition `_WF_request([task,param],_WF_activity)`. This restores the workflow state prior to entering the task. In case of a successful task completion, firing transition `_WF_confirm(_WF_activity,result)` will make the result of the task available to subsequent tasks. It can also be used for control flow purposes.

The subnet in the gray box will always have exactly the same structure. What will vary are the input and output places to the task. For the convenience of the presentation as well as the specification, we introduce therefore a new symbol that represents a task specification. As depicted, we represent the subnet shown in the lower part of Figure 9 by using the *task symbol*, which is presented in its upper part.

4 The Runtime Environment

The runtime environment is basically provided by the tool *REference NEt Workshop* (Renew). Renew provides a full simulation environment for reference nets, including the invocation of Java methods that occur as reference-net inscriptions and the handling of concurrent net processes. In particular, it handles the synchronisation of firing transitions based on their channel inscriptions. To make Renew incorporate a complete workflow engine, `Tasks`, `WorkItems`, and `Activities` as well as their storage and handling have been implemented in additional packages to the Renew system.

4.1 The Specification Language

The Renew system has been extended to deal with task symbols. They will be compiled down to reference-net structures as presented in Figure 9 for an example task. The compilation process was integrated fairly easily into Renew because of Renew's general structure: Between the level of the graphical editor, which has been extended by the task symbol, and the level of the simulator, Renew contains an additional layer called the *shadow level*. The shadow level uses an internal representation of the net drawn on the net-editor level. The translation process from the editor to the shadow level could therefore be extended by a compiler for task symbols that replaces each task symbol by the three-transition sub-net it represents (Figure 9). On the shadow level we end up with only reference nets without task symbols, which then can be fed into the simulation engine as usual. To add task-specific functionality to the simulation engine, it has been extended by a workflow engine.

4.2 The Workflow Engine

The workflow engine comprises as one of its main features an interface to workflow users: Users will be able to access their agendas, i.e. task lists, using the workflow engine. These task lists will be filtered so that a user sees only tasks she/he is authorised to work on (see next section). Users can then select a task

from their task list to deal with it and inform the system about the successful or unsuccessful completion (termination) of a task. The entire handling of tasks lies within the packages that have been added to Renew's simulation engine. This includes returning unfinished (unsuccessfully terminated) tasks to task lists, restoring state information in case of unsuccessful termination of tasks, dealing with automatic tasks (tasks that can be performed without direct user involvement), and so on. It should be noted that the additional packages provide interfaces to functionality already present in Renew, controlling, for instance, the firing of transitions according to the state of tasks. They also comprise a task database for the efficient storage of tasks and their inclusion to the aforementioned task lists.

The workflow engine fulfills therefore two tasks: the control of workflow processes via the Renew simulation engine and the management of tasks, users, and the processing of tasks by users. As mentioned above, tasks may only be processed by a user if she/he has the authority to do so. The handling of access rights is done by rules added to tasks, which form the major part of the workflow security system.

5 Workflow Security

In this section we will give a short overview of security issues in the context of workflow systems. The first subsection covers general aspects while the second one covers how access control has been integrated in our workflow concept and workflow engine.

5.1 Overview

The discussion of security in workflow systems can basically be reduced to discussing access control. In all other aspects, such as confidentiality of messages, authentication, etc., workflow systems are no different from other applications. The access-control aspect of workflow systems deals with issues like: "Who is responsible for performing a task and therefore needs access to data and software supporting working on the task?" Also delegation of work and related access rights are important issues within a workflow system. We will discuss in this section how reference-net-based workflow specification supports an elaborate access-control mechanism.

We use in our approach an extension to the role-based approach.[1] In role-based access controls, users are assigned roles. Roles are simply named collections of users, where a single user can appear in multiple roles. There is also a sub-role relation that holds true, if all users in one role (the sub-role) are also contained in another role (the super-role). For the ease of discussion, we will view users subsequently as singleton roles. They are the only "roles" than can log-in to the system.

[1] Role-based access controls are sometimes also called group-based.

In the *purely* role-based approach, rights to perform specific tasks (and therefore access rights to the relevant software) are assigned to roles. A role inherits access rights from its super-roles. This is all that is possible in a purely role-based access control. To extend it slightly by supporting delegation, to perform a single task (and using the relevant software) a role may pass on its own access rights temporarily to another role. This temporary extension of a role's access rights will expire as soon as the task is completed (either successfully or unsuccessfully).

Even though delegation of access rights makes role-based access controls in a workflow environment more flexible, it is still a quite static concept. Very frequently, access permissions to data or software will vary over time; i.e. in different workflow states, access rights be assigned differently. So more flexible access-controls have emerged. The idea of *context-dependent* access controls, for instance, takes into account that access to particular data will normally only be important to perform one task in a workflow, but not another task. So state information about the workflow is combined with role information to compute access rights. This context-dependent concept, where the context is the workflow state, can be applied to workflows in environments in which state-dependent protection of data is legally required, for instance in clinical trials.

In the approach we present in this paper we go even one step further and allow *all* available information to be part of the access control decision, i.e. we keep the role concept but combine it with information about the workflow's state, data-base information, states of running programs, states of other workflow instances, etc. The aim is to be as flexible as possible. Obviously the utmost flexibility will never be needed. So, in practice, restrictions will apply. However, in different environments, in which workflow systems can support daily work, security requirements differ. Our approach can therefore be adapted to the different environments by simply not using all its possibilities. In that sense, the approach we present in this paper is a generic one.

5.2 Access Control in a Reference-Net Workflow

As mentioned in the previous sub-section, the aim of access controls in workflows is to define who is and who is not allowed to perform a task and access relevant data for this task. The access control system is the implementation of such an access control policy. In the modelling technique we are using, we will assign an *access rule* to each task. The rule will be evaluated to check the authority of an attempt to perform a task.

The first thing to note is that by moving the access control to the task level, our access control is a context-dependent one: Aiming to perform a task will only be possible if the role trying to do so has the authority for performing the task *and* if the workflow is in a state such that the task is active. However, an access rule for a task is in principle a method of type `boolean` that checks the authority of a request to perform the task. It may use any information available in its environment, including remote information if it has network access. This offers the greatest flexibility possible.

Our approach, as a generic one, is open to where the access control is implemented in the workflow. There are basically two options: in the control flow or in the task inscription.

- (**Control flow:**) We can specify the workflow in such a way that, by using control input places to tasks, tasks will only be enabled to be performed by a role, if all other requirements constraining the access to a task are satisfied (i.e. the rule is satisfied). The satisfaction or dissatisfaction of these requirements will be reflected by the availability or unavailability of tokens in a task's control input place necessary to enable the task.
- (**Task inscription:**) As reference net's allow for the inscription of transitions with Java code, we can implement the access control method to a task by an inscription of the entry transition of the task. The task will only be enabled when its entry transition is enabled, and the entry transition can only be enabled by being in the correct workflow state plus the evaluation of the access control method to `true`.

Obviously we can mix the two concepts as we like in a reference-net workflow specification, controlling one task using the control-flow scheme and another task by a task inscription. However, in practice, combining the two approaches in one specification is probably not a good idea as it will decrease the readability of the specification and probably increase the likelihood of introducing errors.

6 An Example Workflow

This section presents an example of a workflow for a mobile-phone shop. It contains several aspects of the workflow engine. In this contribution we include only the pure nets which have been executed by our workflow engine. The environment, the data description, and some other auxilary classes are not shown here.

Figure 10 represents the main net of a mobile-phone shop. Figure 11 contains the check of some customer data and Figure 12 contains the delivery of the goods as well as the check if a lower bound for the amount of goods has been reached.

The main net starts with a transition labeled with `manual` (that is the `new customer` transition): A new order form is created. Then the `EnterOrderData`-task is displayed at all salesmen displays. After its completion a new instance of the form exists.

The larger places in the net are used to represent the existing orders for a customer. The actual order now lies in place `unchecked orders`.

For this order a `WorkflowTask` for the `check of the order` (a net instance of Figure 11) is instanciated. `:start(activity,order)` allows the access to the order within the net instance of Figure 11. Within this net a `WorkItem` will be offered to the office workers in both cases (payment by credit card or bank). Depending on the result of the check a check of the address may be performed. The main net in Figure 10 now will continue according to the result. In the

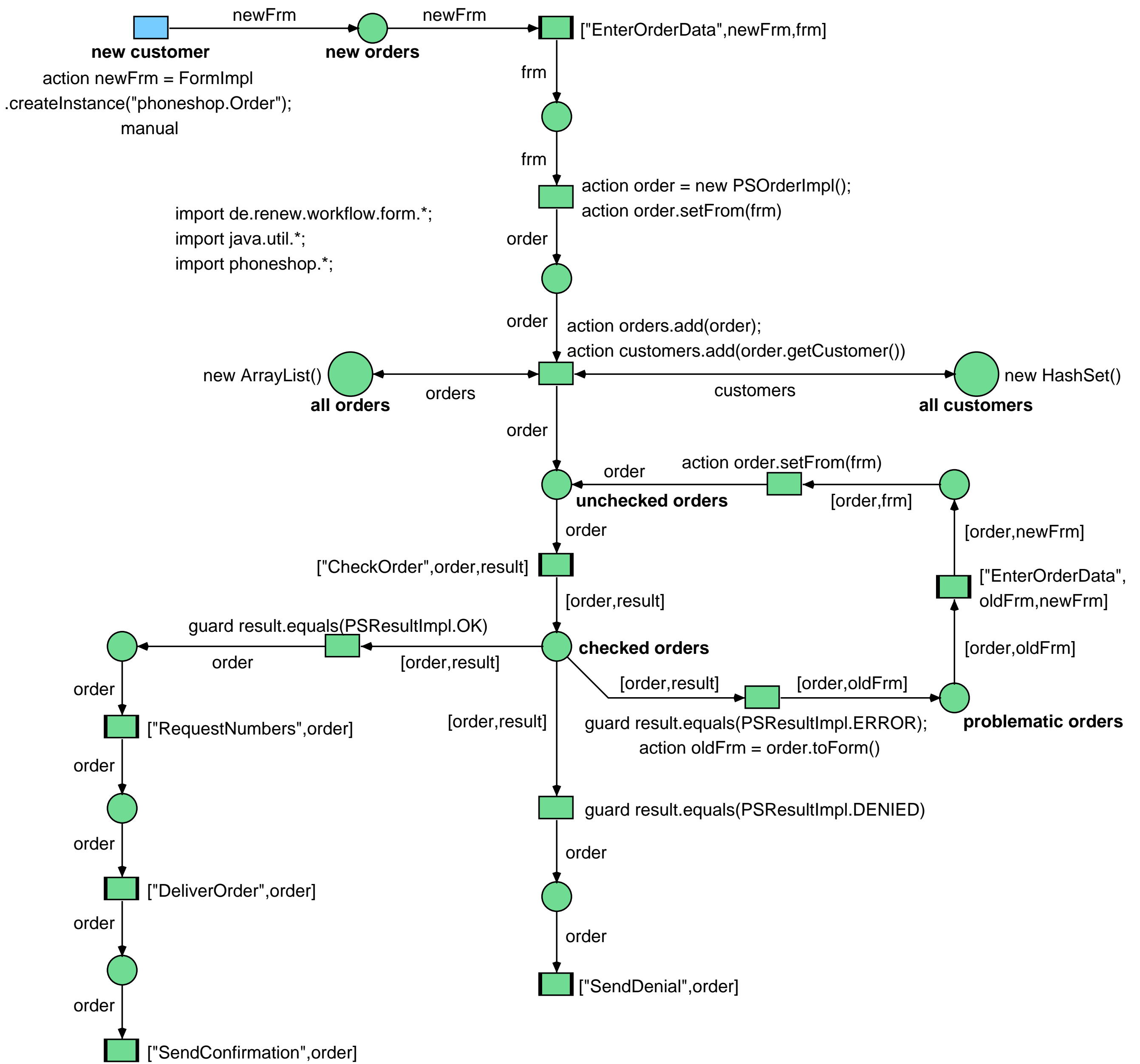

Fig. 10. Main net: Mobile-phone Shop

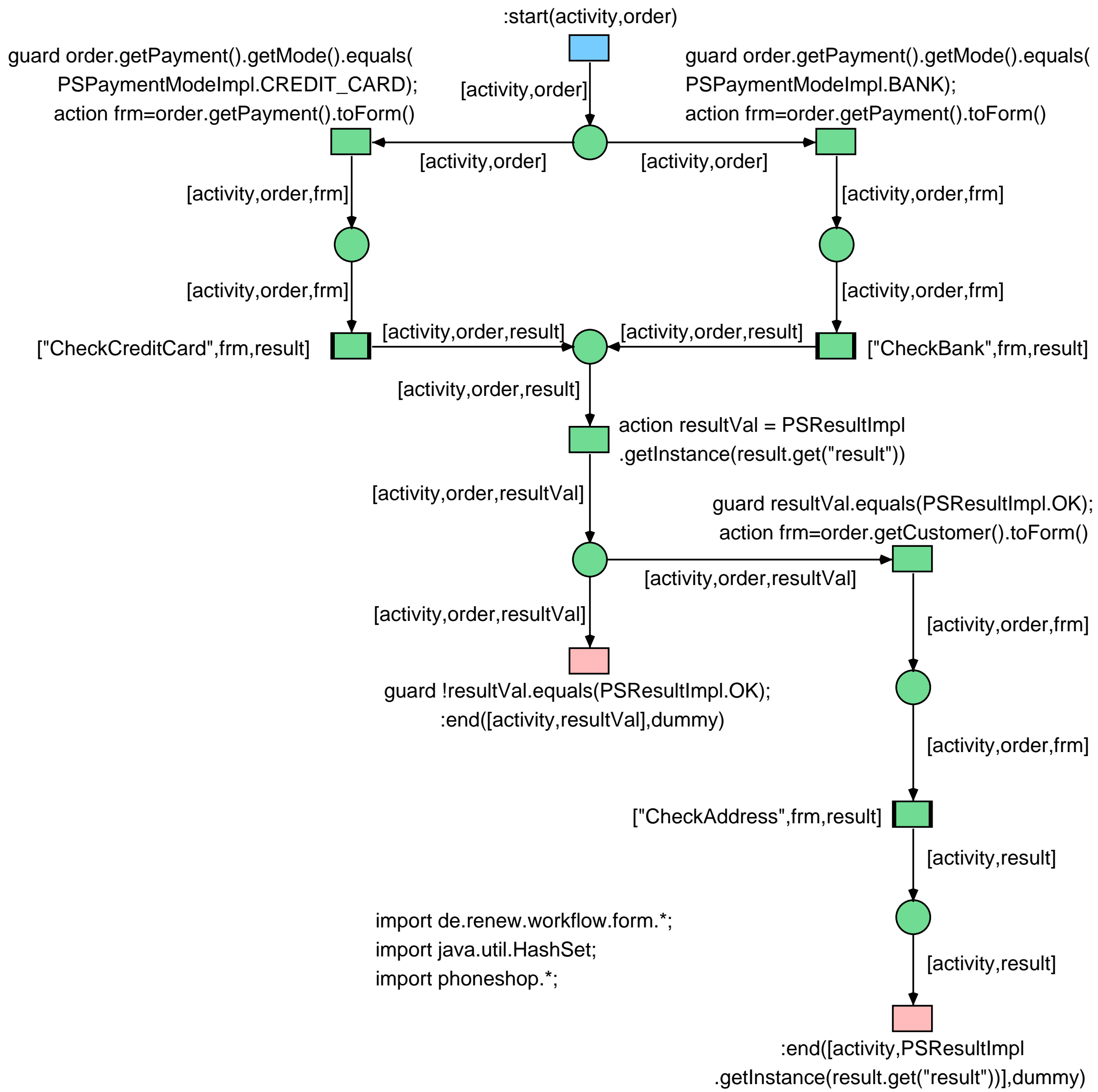

Fig. 11. Check order limit

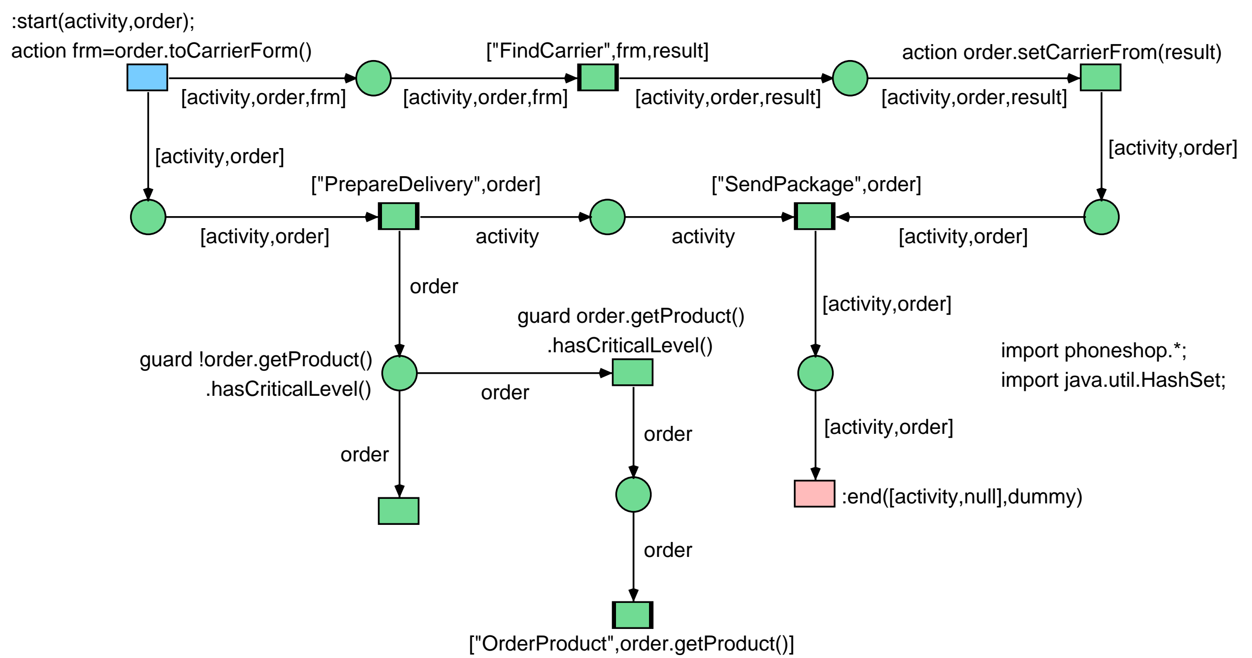

Fig. 12. Shipping of goods

positive case it will continue with the request for phone numbers and then with the sub-workflow in Figure 12 where the shipping is handled.

This last net shows the possibility to finish a workflow for the upper level by calling the `:end(activity,dummy)` transition and still having active parts within the net. Once they are finished the whole net instance is removed from the system if there is no log function in the system. This is possible due to the automatic garbage collection of the tool Renew. The designer of the net has to ensure that the nets are designed in a way that allows to use this feature, however, we will not deepen the discussion here.

This example covers only the operability, however, security issues can be added as described in the sections above. The full modelling (or programming) power of the workflow engine can be seen when looking at the possibilities of the net concepts that can be used and the inscriptions which cover nearly all kind of Java code. One main attempt of our research is now to find the right restrictions and short cuts when designing workflow systems. Following our overall approach we aim at the integration of modelling and implementation. Without tool support this does not seem to be feasible.

7 Conclusion

We have discussed in this paper how reference nets can be used to specify workflows and how the REference NEt Workshop (Renew) tool [6] can be extended to form the corresponding workflow engine. The major additional concept introduced was that one of a *task*. We were able to reduce the concept of a workflow task to a three-transition reference-net structure. Such a structure has been integrated into the three-layer architecture of Renew [3]. By adding a component to handle tasks, tasks lists of user, etc., Renew was turned into a complete run-time environment for reference-net workflows. The basic mechanism of the workflow engine has successfully been used in the context of mobile devices (see [7]).

The concept of adding *rules* to tasks enabled us to develop a generic concept for access controls in the workflow engine. As these rules are simply Java methods of type `boolean`, we can flexibly implement any existing access control scheme into our formalism, including role-based and content-dependent access controls [1, 8], based on the implemented features (see [3]). The flexibility of the concept allows for including any other kind of information to check the authority of an access request to a task, such as date and time, states of other workflows, system parameters, and so on. It should be noted that, because of the flexibility of the approach, we only have created a generic concept for the design and implementation of secure workflows. The framework will have to be filled with design strategies that will again impose constraints on the flexibility of implementing task rules. Otherwise the entire approach can become too complex when specifying concrete workflows, which can lead to undetected introduction of security holes hidden behind a too complex mechanism. It will, together with applying our approach to case studies, be part of future work to define such security design strategies.

References

1. RALPH HOLBEIN. Secure Information Exchange in Organisations. Dissertation. Institute for Computer Science, University of Zürich, Switzerland, 1996. (Published by Shaker Verlag, Aachen, Germany, 1996.)
2. SØREN CHRISTENSEN and NIELS DAMGAARD HANSEN. Coloured Petri Nets Extended with Place Capacities, Test Arcs and Inhibitor Arcs in: Marsan, Marco Ajmone (Ed.). *14th Internation Conference on Application and Theory of Petri Nets*. LNCS 691, Berlin Heidelberg New York: Springer Verlag, pages 186–205, 1993
3. THOMAS JACOB. Implementierung einer sicheren und rollenbasierten Workflowmanagement-Komponente für ein Petrinetzwerkzeug. Diploma Thesis. Department for Informatics, University of Hamburg, Germany, 2002.
4. KURT JENSEN. Coloured Petri Nets. EATCS Monographs on Theoretical Computer Science. 3 Volumes, Berlin Heidelberg New York: Springer Verlag, 1992, 1994, and 1997.
5. OLAF KUMMER. A Petri Net View on Synchronous Channels. In: *Petri Net Newsletter* 56: 7–11, 1999.
6. OLAF KUMMER. *Referenznetze*. Dissertation. Department for Informatics, University of Hamburg, Germany, 2002.
7. STEFAN MÜLLER-WILKEN and WINFRIED LAMERSDORF. JBSA: An Infrastructure for Seamless Mobile Systems Integration. In: Claudia Linnhoff-Popien and Heinz-Gerd Hegering (Eds.): *Proc. 3rd IFIP/GI International Conference on Trends towards a Universal Service Market (USM 2000)*, Berlin Heidelberg New York: Springer Verlag, 164-175, 2000
8. ULRICH ULTES-NITSCHE and STEPHANIE TEUFEL. Secure Internet-Access to Medical Data. In: Informatik — Journal of the Swiss Chapter of the ACM. No. 1, February 2001. Special Issue on IS and the Transformation of Health Care, pages 23–26.
9. RÜDIGER VALK. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In: *19th Internation Conference on Application and Theory of Petri Nets*. LNCS 1420, Berlin Heidelberg New York: Springer Verlag, pages 1–25, 1998.
10. WIL M.P. VAN DER AALST, DANIEL MOLDT, RÜDIGER VALK, and FRANK WIENBERG. Enacting Interorganizational Workflows Using Nets in Nets. In: *Proceedings of the 1999 Workflow Management Conference "Workflow-base Applications"*. University of Münster, Germany, pages 117–136, 1999.