Inter-Workflow Support

Ronny S. Mans[1,3], Nick C. Russell[2], Wil M.P. van der Aalst[1], Arnold J. Moleman[3], Piet J.M. Bakker[3]

[1] Department of Information Systems, Eindhoven University of Technology, P.O. Box 513, NL-5600 MB, Eindhoven, The Netherlands.
{r.s.mans,w.m.p.v.d.aalst}@tue.nl
[2] Carba-Tec Pty Ltd, 128 Ingleston Rd, Wakerley QLD 4154, Australia.
nrussell@carbatec.com.au
[3] Department of Quality Assurance and Process Innovation, Academic Medical Center, University of Amsterdam, P.O. Box 2260, NL-1100 DD, Amsterdam, The Netherlands. {a.j.moleman,p.j.bakker}@amc.uva.nl

Abstract. Processes concerning the diagnosis and treatment of patients can be characterized as weakly-connected interacting light-weight workflows coping with different levels of granularity. Moreover, for each individual patient a doctor proceeds in a step-by-step way deciding about the next steps to be taken. Classical workflow notations fall short in supporting these patient processes as they primarily support monolithic processes in which it is assumed that a workflow process can be modeled by specifying the life-cycle of a single case in isolation. In this paper, we present an extension of the Proclets framework which allows for dividing complex entangled processes into simple fragments. Additionally, increased emphasis is placed on interaction related aspects such that fragment instances for individual patients can cooperate in any desired way. Finally, we describe an architecture in which inter-workflow support facilities can be added to existing WfMSs.

1 Introduction

In healthcare organizations, such as hospitals, many complex, non-trivial processes are performed which are lengthy in duration. Actually, a lot of these processes involve patient processes which are concerned with the diagnosis and treatment of patients. For reasons of patient safety and to ensure the quality of services delivered to patients, for these patient processes it is crucial that tasks are performed in the right order and in due time. Moreover, it needs to be taken into account that tasks may be performed at different medical departments. In order for providing support and monitoring for these processes, Workflow Management Systems (WfMSs) present an attractive vehicle to this end. Based on process definitions, WfMSs are able to manage the flow of work in these processes such that individual workitems are done at the right time by the proper person [6, 14, 26, 24].

However, a number of difficulties commonly arise when hospitals attempt to automate these patient processes. This is due to the fact that a patient process

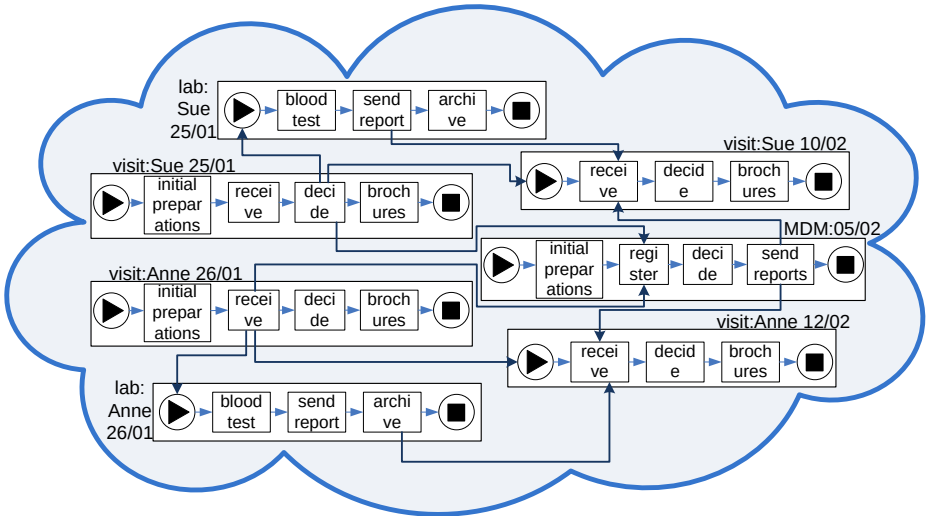


Fig. 1. Interacting workflow fragments.

for an individual patient typically consists of a number of workflow fragments that may cope with different levels of granularity and run at their own speed. Moreover, the doctor proceeds in a step-by-step way when deciding about the steps to be taken next. These two aspects can be illustrated by the example that is depicted schematically in Figure 1.

Figure 1 shows the possible patient process of two patients. For patient "Sue" the process starts with the first visit to the outpatient clinic (the process instance which is named "visit:Sue 25/01"). For the first visit, first some initial preparations are necessary (task "initial preparations"). After this, the results of previous tests are received (task "receive"). Then, a doctor decides about the next step(s) that need to be taken (task "decide") followed by brochures that are provided by a nurse (task "brochures"). As indicated by the outgoing arcs from the "decide" task, a second visit is necessary ("visit:Sue 10/02"), a lab test needs to be taken ("lab:Sue 25/01"), and Sue's case needs to be discussed during a multidisciplinary meeting ("MDM:05/02"). For the lab test that needs to be taken, first a blood test is taken (task "blood test"). Subsequently, a report describing the result of the lab test is sent ("send report"), and the report is archived (task "archive"). Note that as the report is required as input for the second visit, there is an arc leading from the "send report" task to the "receive" task of the second visit. At the multidisciplinary meeting, multiple patients are discussed individually. First, some initial preparations need to be taken for the meeting (task "initial preparations") after which patients can be registered for the meeting (task "registration"). Then at the meeting itself, for the registered patients, a decision is made about the next steps that need to be taken (task "decide"). Finally, reports are sent out (task "send reports"). Note that Sue is

registered for the meeting and that the resulting report is necessary as input for the second visit.

For patient "Anne" a similar process is followed. Note that for both patients many more options may be possible. For example, for Sue an MRI or CT test may be necessary, which is either initiated at the second visit (task "decide") or during the multidisciplinary meeting (task "decide"). Alternatively, the result of the lab test for Sue may be necessary as input for the multidisciplinary meeting instead of the second visit. Also note that Anne and Sue are discussed at the same multidisciplinary meeting. So, process instance "MDM 05/02" operates at a different level of granularity (a group of patients) than the other instances shown in Figure 1 (a single patient).

As illustrated by the example, the entire patient process should be seen as a *cloud of standardized workflow fragments for which the ultimate selection of these fragments and the interactions between them is patient specific.* Moreover, these fragments may cope with different levels of granularity. Current workflow languages require that the complete workflow is described as one monolithic overarching workflow [25]. As a consequence, using current workflow languages, *it is hard to describe such a cloud of loosely coupled workflow fragments and all the possible interactions that may occur between these fragments.*

In particular, for providing the required support, so called Proclets are an interesting means in order to model these kind of processes and have them executed by a WfMS. Proclets are *lightweight interacting processes* that can be used to divide complex entangled processes into simple fragments and, in doing so, place increased emphasis on interaction-related aspects of workflows. Proclets aim to address the following problems that existing workflow approaches are currently facing:

- Models need to be *artificially flattened* and are unable to account for the mix of *different perspectives and granularities* that coexist in real-life processes.
- Cases need to be *straightjacketed into a monolithic workflow* while it is more natural to see processes as intertwined loosely-coupled processes
- *One-to-many* and *many-to-many* relationships that exist between entities in a workflow can not be captured.
- It is difficult to model interactions between processes.

Proclets were one of the first modeling languages to acknowledge above mentioned problems and are part of the Proclet framework which has been described in detail by van der Aalst et al. [4, 5]. Based on our healthcare experiences, an extension of the Proclet framework will be presented in this paper. In particular, via so-called "interaction points", "internal interactions", and "external interactions", at design time, possible interactions between Proclet classes can be modeled without the need to define complex pre- and postconditions. Next, at run-time they allow users to select interactions between future and existing Proclet instances. In this way, users can define the next steps that need to be taken in the process of treating a patient (e.g. a lab test, a next visit of a patient). Note that although the framework has been extended based on experiences from the healthcare domain, this does not mean that its application is limited to the

healthcare domain only. Any environment where processes are fragmented, interaction is important, and tasks which are done at different levels of granularity, can potentially be supported.

In order to illustrate that the framework can also be applied in other environments, we next list some real world examples where interactions between instances, different levels of aggregation, and relations between entities play an important role:

- The reviewing of papers for a conference: For a conference the goal is to select the best papers from all papers that have been submitted. Before issuing a call for papers, several people are invited to take part in the program committee. For the papers that have been submitted, the program committee selects multiple reviewers per paper. A reviewer may be asked to review multiple papers. Finally, a decision is made by the program committee and the authors are informed whether their paper has been rejected or accepted.
- Processing of insurance claims: some claims may refer to the same accident. Even though claims may start out as separate instances, at some point it may be desirable to merge all related claims so that a uniform decision can be reached.
- In software development: software modules are composed of submodules, which in turn may be composed of sub-submodules and so on. Considerations at higher or lower levels of aggregation may influence other levels. For example, the discovery of a design flaw at a higher level may impact modules at lower levels or at the same level at which the flaw has been discovered.

As indicated earlier, the Proclet framework allows for executing patient processes in a WfMS. So, an additional contribution of this paper is that we investigate the *general problem of how WfMSs can be fully extended with facilities for inter-workflow support*. So, our focus is not on simply extending the functionality of a particular WfMS.

The remainder of this paper is organized as follows. The Proclet framework together with our contributions will be discussed in detail in Section 2. Afterwards, in Section 3, the design of a WfMS extended with inter-workflow support is presented. Section 4 discusses related work. Finally, Section 5 concludes the paper.

2 The Proclet Framework

In this section, we discuss how Proclets provide a framework for modeling and executing workflows. First, the most important concepts of the framework will be introduced in Section 2.1. Afterwards, in subsequent sections, particular aspects of the framework will be addressed.

2.1 Concepts

In this section, we discuss the main concepts of the Proclet framework. This will be done by two scenarios. One scenario is rather simple while the other one is

more complex. Additionally, these scenarios allow for showing the mechanisms that relate to the concepts that are introduced.

First Scenario Before introducing the framework, we first present the first scenario which is shown in Figure 2. In this scenario, we schematically depicted the process that needs to be followed by patient "Sue". Currently, Sue is in the process of having a first visit (fragment "visit:Sue 25/01"). As indicated by the outgoing arcs from the "decide" task, the doctor decides during the "decide" task that a next visit is necessary (fragment "visit:T2") and that a lab test needs to be taken (fragment "lab:T1"). As the last two fragments need to be created in the future, the fragments for them are visualized with a dotted rectangle around them. Additionally, their instance identifier starts with a "T". As a subsequent action of creating an instance for the lab test, the result of the lab test needs to be used as input for the second visit. This is indicated by the arc leading from the input condition of the "lab" fragment to the "send report" task and the arc leading from the "send report" task to the "receive" task of the second visit.

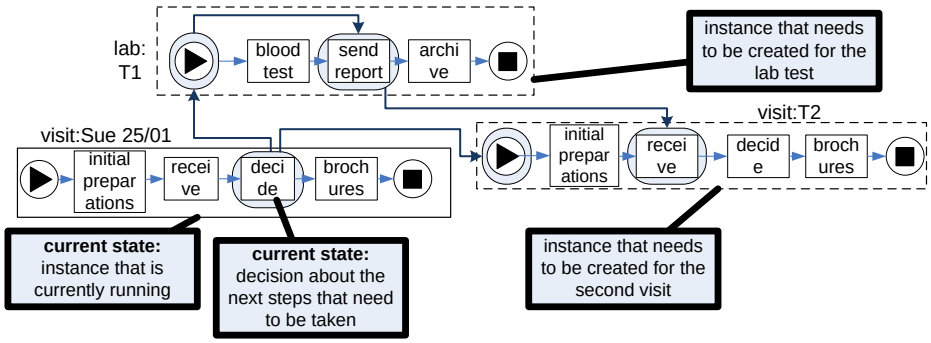


Fig. 2. The first scenario. In this scenario, for patient "Sue" it is decided during the first visit that a lab test is required and that a second visit is required.

Based on the scenario discussed above, we start introducing the Proclet framework. That is, the framework is centered around the notion of ***Proclets***. There is a distinction between a Proclet class and a Proclet instance. A *Proclet class* can best be seen as a process definition which describes which tasks need to be executed and in which order. For a Proclet class, instances can be created and destroyed. One instance is called a *Proclet instance*. For the definition of a Proclet class, a selection can be made between multiple graphical languages. In this paper, we use a graphical language based on the YAWL language [7]. However, other languages, such as Petri Nets [1] or EPCs [2], can also be used. With regard to the selection of a graphical language, some limitations apply. First of all, Proclet instances need to have a state and they need to support the notion of a task. Second, a Proclet class needs to be sound, i.e., satisfy basic

correctness requirements such as absence of deadlocks, proper termination, etc. [3].

In order to have interactions and collaboration among Proclets, *interaction points*, *channels*, *ports*, and *performatives* are important. The meaning of them will be discussed below. Additionally, we describe how a Proclet class and instances of it are defined.

- A Proclet class has a ***unique name***. In the same way, an instance of a Proclet class has an unique identifier.
- Proclet instances interact with each other via ***channels***. A channel can be used to send a ***performative*** to an individual Proclet instance or to a group of Proclet instances.
- A performative is a specific kind of message with several attributes which is exchanged between one or more Proclets. Two important attributes are the "sender" and "set of receivers" attributes. The *sender* attribute contains the identifier of the Proclet instance creating the performative. The *set of receivers* attribute contains the identifiers of the Proclet instances receiving the performative, i.e. a list of recipients. Additional attributes will be discussed in more detail later (Section 2.4).
- A Proclet class has ***ports***. Performatives are sent and received via these ports in order for a Proclet instance to be able to interact with other Proclet instances. A port is either an incoming or an outgoing port. Each outgoing port is connected with exactly one incoming port. We call such a connection, an ***external interaction***. Furthermore, every port is connected to one ***interaction point***. An interaction point represents a specific point in the Proclet class at which interactions with other Proclet classes may take place, i.e. via the associated ports performatives may be sent and received. An interaction point may be linked to an input condition and a task.
- Moreover, a port has two attributes.
  First, the *cardinality* specifies the number of recipients of performatives exchanged via the port. An $*$ denotes an arbitrary number of recipients, + at least one recipient, 1 precisely one recipient, and ? denotes no or just one recipient. Note that by definition an input port has cardinality 1.
  Second, the *multiplicity* specifies the number of performatives exchanged via the port during the lifetime of an instance of the class. In a similar fashion to the cardinality, an $*$ denotes that an arbitrary number of performatives are exchanged, + at least one, 1 precisely one, and ? denotes that either one or no performatives are exchanged.
- For an interaction point having only incoming ports, it may be desired that the receiving of an individual performative is followed by the subsequent sending of a performative later in the process (e.g. the creation of a lab test needs to be followed by the execution of a task in the same process which sends the result of the test to the desired Proclet instance). Therefore, an interaction point with only incoming ports may be connected with an interaction point which has only outgoing ports. Such a connection is called an ***internal interaction***.

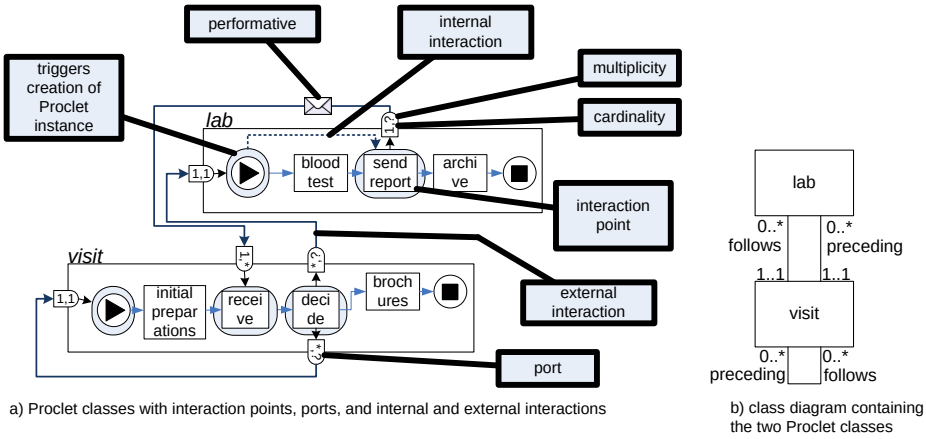


Fig. 3. Based on the first scenario, the concepts of the Proclet framework that are introduced so far are illustrated. That is, two Proclet classes are modeled. Moreover, for them, interaction points, channels, ports, and performatives are indicated.

The above mentioned concepts are illustrated in Figure 3. Based on the first scenario, in Figure 3a, two Proclet classes are shown together with their interaction points, ports, and external interactions. In Figure 3b, a class diagram is shown containing the two Proclet classes. First, as can be seen in Figure 3a, the "visit" Proclet class models all the tasks related to a visit of the patient, whereas the "lab" Proclet class does the same for a lab test. The "decide" step of the "visit" Proclet class has an interaction point with two outgoing ports. One outgoing port is leading to the interaction point that belongs to the input condition of the "lab" Proclet class. Sending a performative to the incoming port of this interaction point results in the creation of an instance of the "lab" Proclet class. Similarly, sending a performative via the second outgoing port of the "decide" task results in the creation of an instance of the "visit" Proclet class. As indicated by cardinality * for the two outgoing ports of the "decide" task only multiple instances of the "lab" Proclet class and multiple instances of the "visit" Proclet class may be initiated. The multiplicity of the two ports is ? which means that it is optional to send a performative in order to create an instance of the "lab" and "visit" Proclet class. Finally, performatives can be send from the "send report" task to the "receive" task modeling that the result of a lab test may be used as input for a patient visit. The cardinality 1 and multiplicity ? of the outgoing port of the "send report" task indicate that it is optional to send a performative to one "visit" Proclet instance. In a similar fashion, the cardinality 1 and multiplicity * of the incoming port of the "receive" task indicate that it is optional that performatives are received from the "send report" task.

However, although performatives can be sent to multiple receivers, there is still the issue that it needs to be *controlled to which specific Proclet instance or*

instances a performative is sent. For example, for Figure 3, when an instance exists of the "lab" Proclet class and the "send report" task is executed, it is still the question to which "visit" Proclet instance a performative is sent. In particular, if we want to achieve the behavior for "Sue" which is defined during the execution of the "decide" task in the first scenario (which is visualized in Figure 2), the following is required for the two Proclet models shown in Figure 3

- One Proclet instance exists for the first visit which has "visit:Sue 25/01" as instance identifier.
- A performative is sent from the "decide" task of the first visit to the initial condition of the "visit" Proclet such that one instance is created for the second visit of "Sue".
- A performative is sent from the "decide" task of the first visit to the initial condition of the "lab" Proclet such that one instance is created for the desired lab test for "Sue".
- The creation of an instance for the lab test should be followed by the execution of the "send report" task such that a performative is sent from that task to the "receive" task of the second visit.

Next to that, for the above mentioned interactions it is important that it is known whether they already have taken place, i.e. the state of them needs to be known. For example, in order for the "receive" task of the second visit to take place it is important to know whether the performative from the "send report" task has already been received.

Entities and Interaction Graphs

In order to be able to precisely specify the interactions that need to take place for "Sue" and their current state, we need to introduce two additional concepts. The fist concept is called an *entity.* An entity is an object that exists next to existing and future Proclet instances. Examples of an entity are a patient, a claim, or a software product that needs to be developed. So, "Sue" can be an entity. For an entity, tasks in multiple Proclet instances need to be performed. In order for these tasks to be performed in the desired order, specific interactions are required between existing and future Proclet instances. Note that this also may involve a sequence of interactions among multiple Proclet instances.

In order to store for an entity the interactions that need to take place between existing and future Proclet instances and their state, we introduce a so-called *interaction graph.* An interaction graph belongs to a specific entity and consists of *interaction nodes* and *interaction arcs.* An *interaction node* refers to an interaction point of a Proclet instance for which *one or more internal or external interactions will take place*, i.e. an instance of an interaction point. So, an interaction node is a triple of which the first value refers to the identifier of the Proclet class, the second value refers to the identifier of the Proclet instance, and the third value refers to the identifier of the interaction point for which one or more

interactions take place. On its turn, an *interaction arc* refers to an interaction, either internal or external, that needs to occur between two interaction points of a Proclet instance. In that way, the direction of the arc in the graph is the same as the direction of the arc for the associated internal or external interaction.

In Figure 4, for the first scenario, the corresponding interaction graph is given for entity "Sue". First, in Figure 4a, the instances for the fist visit, second visit, and lab test are shown. However, they are now modeled using terminology of the Proclet framework, i.e. using interaction points, ports, and so on. For example, the "decide" task of the first visit has two outgoing ports illustrating the performatives that will be sent in order to create an instance for the lab and the second visit. Also, for the first visit (Proclet instance with identifier "visit:Sue 25/01"), currently the "decide" task is executed. As a result, the "initial preparations" and "receive" task are already executed which is indicated by the check marks.

As a result of executing the "decide" task for the first visit, which necessitates interactions with existing and future Proclet instances, an interaction graph is created for entity "Sue". The graph is shown in Figure 4b. There are five interaction nodes and four interaction arcs. Note that by dotted arcs, nodes of the interaction graph are linked with their corresponding interaction points in a Proclet instance. Additionally, via dotted arcs, arcs of the interaction graph are linked with their corresponding internal or external interactions. The meaning of each arc for the entity "Sue" is as follows:

- (visit,Sue 25/01,decide) → (lab,T1,create): from the "decide" task of the "visit" Proclet class with instance identifier "Sue 25/01", a performative is sent in order to create an instance of the lab Proclet class. As the lab instance still needs to be created a temporary instance identifier is used for it (i.e. T1). Note that the arc refers to an external interaction. For presentation reasons, input and output ports are not shown in an interaction graph. Instead, for an external interaction, the respective interaction nodes are immediately connected via an arc.
- (visit,Sue 25/01,decide) → (visit,T2,create): similar as for the previous arc. This time an instance of the "visit" Proclet class needs to be created which represents the second visit. Note that also for the second visit a temporary instance identifier is used (i.e. "T2").
- (lab,T1,create) → (lab,T1,send report): the creation of an instance for the "lab" Proclet class needs to result in a subsequent interaction. This is represented by an internal interaction for which no performatives will be sent. Note that the subsequent interaction is the sending of a performative, starting from the "send report" task of the same instance to the "receive" task of the second visit.
- (lab,T1,send report) → (visit,T2,receive): from the "send report" task of the "lab" Proclet instance, a performative needs to be sent which is received by the "receive" task of the future "visit" Proclet instance for the second visit which has temporary instance identifier "T2".

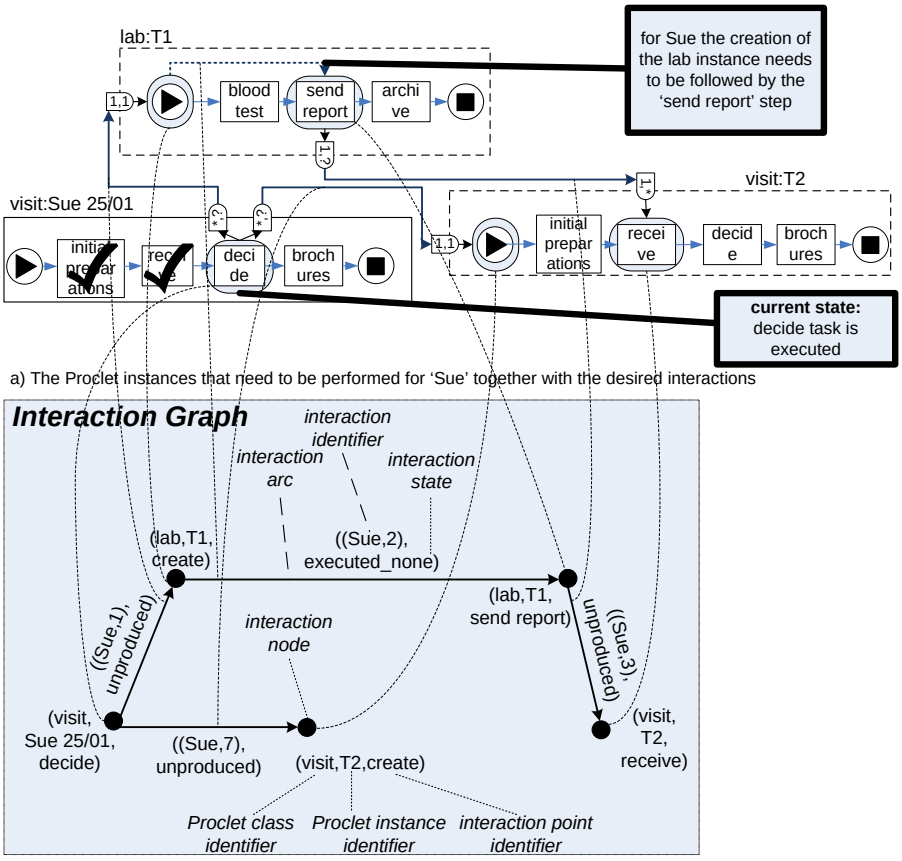


Fig. 4. For entity "Sue", using the Proclet terminology introduced so far, it is shown how the existing and future Proclet instances need to interact (Figure a). Additionally, for entity "Sue" the associated interaction graph is shown (Figure b).

Obviously, the interaction graph of entity "Sue" saves all the interactions that need to take place between future and existing Proclet instances. In other words, the entity "Sue" is the linking pin between the three Proclet instances.

As indicated before, in an interaction graph we also save the state of the interactions for an entity. Therefore, every arc in the interaction graph has an *interaction identifier* and an *interaction state*.

The *interaction identifier* is an identifier of which the first value relates to the entity itself and of which the second value relates to a unique identifier for the interaction. These interaction identifiers allow for keeping track of the state of external interactions for entities, i.e. performatives that are exchanged. Additionally, in order to realize the latter, an additional attribute is added to

a performative called *set of interaction identifiers*. For the interaction arcs for which a performative is sent, the associated interaction identifier is added to this set. More details will be provided later.

Next, the *interaction state* of an arc stores the specific state of an interaction for the respective entity. For example, for an external interaction, has already a performative been sent or received. For an internal interaction, is the task that is linked to the interaction point already executed or not. In total, for an arc referring to an external interaction we distinguish four different states and for an arc referring to an internal interaction we also distinguish four different states. Below, for the first scenario, tasks will be executed for different Proclet instances. In that way, it can be seen which performatives are exchanged. Moreover, it allows for explaining how the arcs of an interaction graph are updated and which states we distinguish. In this way, the "mechanisms" of an interaction graph can be illustrated and the "mechanisms" of an interaction point, internal interaction, and external interaction.

Executing the First Scenario

For the first scenario, visualized in Figure 2, subsequently tasks for the first visit, lab test, and second visit will be performed. Similarly, as in Figure 4 we show the current state of the Proclet instances and the current state of the interaction graph for entity "Sue". Tasks that are completed are indicated by a check mark.

Step 1:

In Figure 4, the "decide" task of the first visit is currently executed and an interaction graph is created. In Figure 5, the next step is visualized. That is, for the "decide" task that is executed, in the interaction graph it can be seen that two interactions need to take place. So, two performatives need to be sent in order to create an instance of the "lab" Proclet class and an instance of the "visit" Proclet class. As an example, we see the performative that is sent to create an instance of the "visit" Proclet class. The sender of the performative is the Proclet instance that is currently executed ("visit:25/01"). The receiver of the performative is the instance of the "visit" Proclet class which has temporary identifier "T2" as it still needs to be initiated. As interaction identifier we see that "(Sue,7)" is added.

As a result, in the interaction graph for entity "Sue", we see that the interaction state of the arc leading from the "(visit,Sue 25/01,decide)" node to the "(lab,T1,create)" node has been changed to "sent". The same can be observed for the arc leading from the "(visit,Sue 25/01,decide)" node to the "(visit,T2,create)" node. This is due to the fact that a performative has been sent for that interaction. That is, the sender and receiver of the performative match with the sender and receiver of the associated interaction arc in the graph. Also, the performative contains the interaction identifier of the arc for which it is sent.

Step 2:

The next step is shown in Figure 6. The receival of the two performatives has resulted in the creation of an instance for the "lab" Proclet class and the "visit" Proclet class as well. Instead of instance identifier "T1" the instance of the "lab"

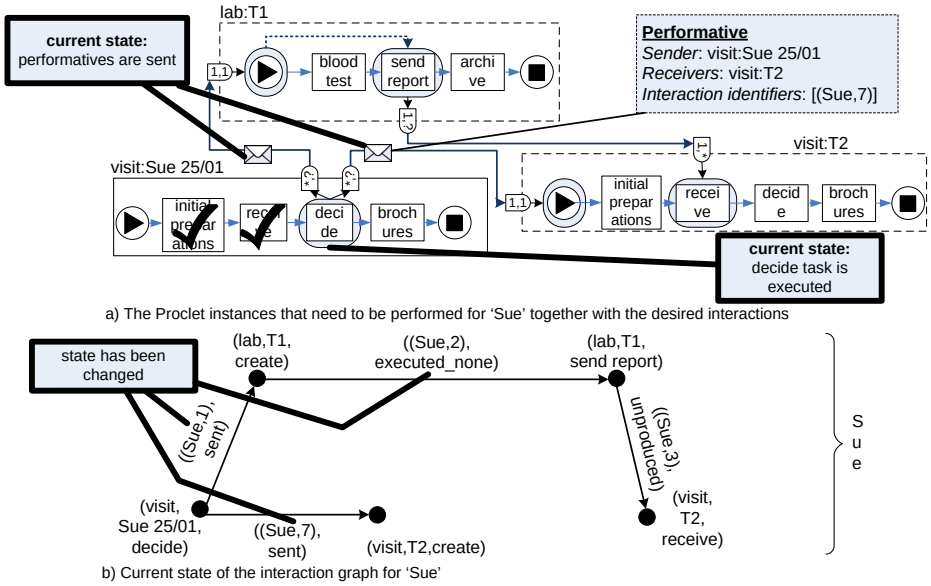


Fig. 5. As a result of executing the "decide" task, two performatives are sent. As a consequence, the interaction graph is updated.

Proclet class has now identifier "lab:25/01". As a consequence, in the interaction graph for "Sue", the interaction nodes referring to the "lab" instance have been updated with the new instance identifier. Moreover, the interaction state of the arc leading from the "(visit,Sue 25/01,decide)" node to the "(lab,25/01,create)" node has been changed to "consumed". That is, a performative has been received for that interaction arc which resulted in the creation of an instance of a Proclet class, i.e. the performative can be considered as "consumed" as its receival led to a certain action. Additionally, it can be seen that the interaction state of the arc from the "(lab,25/01,create)" node to the "(lab,25/01,send report)" node has been changed to "executed source". As an instance of the "lab" Proclet class has been created, for this internal interaction the source interaction point has been executed.

For the instance of the "visit" Proclet class that has been created, similar remarks can be made. As can be seen in the graph, the interaction state of the associated arc has been changed to "consumed" too. Moreover, instead of instance identifier "T2", the instance of the "visit" Proclet class has now instance identifier "Sue 10/02".

Step 3:

As a next step in the scenario, the "blood test" task of the "lab" Proclet instance has been performed. As a result, the "send report" task may be performed. The result of performing the task can be seen in Figure 7. For, the arc in the interaction graph leading from the "(lab,25/01,send report)" node to the

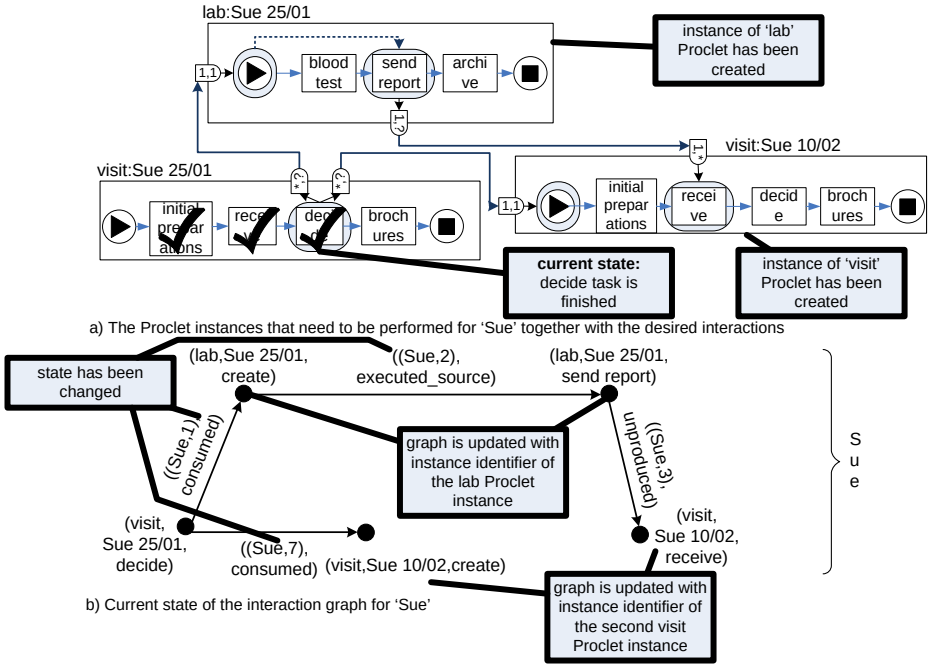


Fig. 6. As a result of receiving the two performatives, instances for the "lab" and "visit" Proclet classes are created. As a consequence, the interaction graph is updated.

"(visit,Sue 10/02,receive)" node, a performative is sent to the "receive" task of the "visit" Proclet instance with instance identifier "Sue 10/02". As a result, the interaction state of the arc has been updated to "sent". Moreover, the performative that is sent is visualized in the figure. That is, the sender and receiver of the performative match with the sender ("lab:Sue 25/01") and receiver ("visit:Sue 10/02") of the associated interaction arc in the graph. Also, the performative contains the interaction identifier of the arc for which it is sent ("(Sue,3)").

Furthermore, it can be seen that the interaction state of the arc from the "(lab,Sue 25/01,create)" node to the "(lab,Sue 25/01,send report)" node has been changed to "executed both". Due to the execution of the "send report" task both the source and destination interaction node of this internal interaction have been executed now. In that case, the interaction state is updated to "executed both".

In case the arc would not have state "executed source", i.e. the interaction point which is connected to the input condition is not executed, the "send report" task may not be executed. This is due to the fact that the meaning of an internal interaction is that first the source interaction point is executed, which is linked with either a task or an input condition, and then the task which belongs to the destination interaction point is executed.

Step 4:

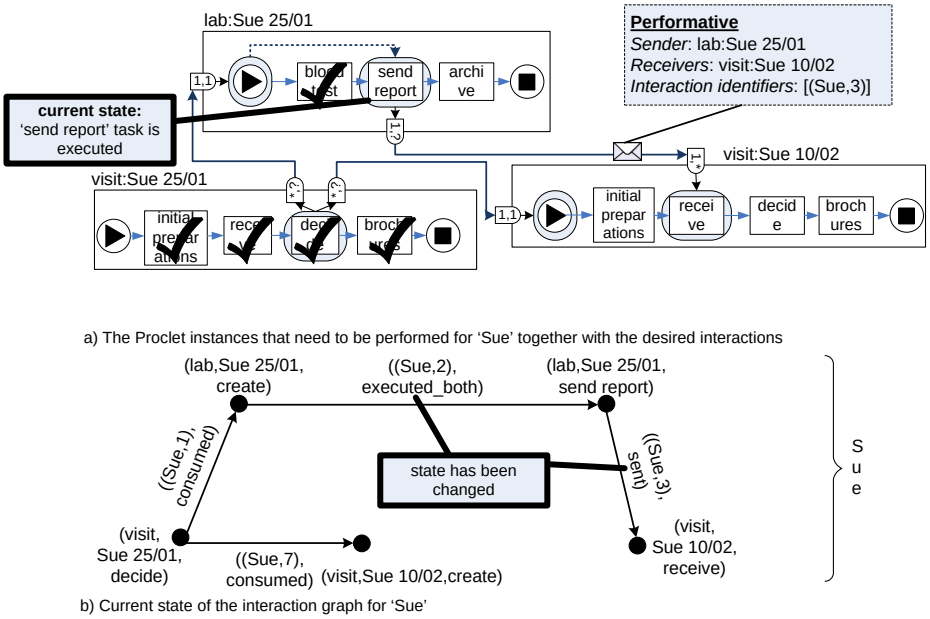


Fig. 7. The "send report" task of the "lab" Proclet instance is executed. This results into a performative that is send to the "receive" task of the "visit" Proclet instance for the second visit. As a consequence, the interaction graph is updated.

The last step of the first scenario is that we start executing steps for the second visit. As can be seen in Figure 8, the "receive" task is currently executed. For this task, we find in the interaction graph, the arc leading from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node. This indicates that for completing this task it is required that a performative is received which contains "(Sue,3)" as interaction identifier. As this performative has been sent as result of executing the "send report" task, the "receive" task may be completed. Consequently, the interaction state of the arc is updated to "consumed" indicating that the required performative was available and that it has been consumed in order to complete the task. Note that if the performative would not have been available yet, it is not possible to complete the task. So, although it is possible to complete the task according to the process definition, it still needs to wait till all required performatives are received. However, an exception to this rule is possible. This will be discussed later in Section 2.2.

Second Scenario Before, we have considered a simple scenario for which we have shown some of the mechanisms of an interaction graph. Moreover, we only considered steps that are done for an individual patient. Now, as a follow-up we consider a more complex scenario in which we demonstrate that the framework can also deal with Proclet classes that operate at different levels of granularity.

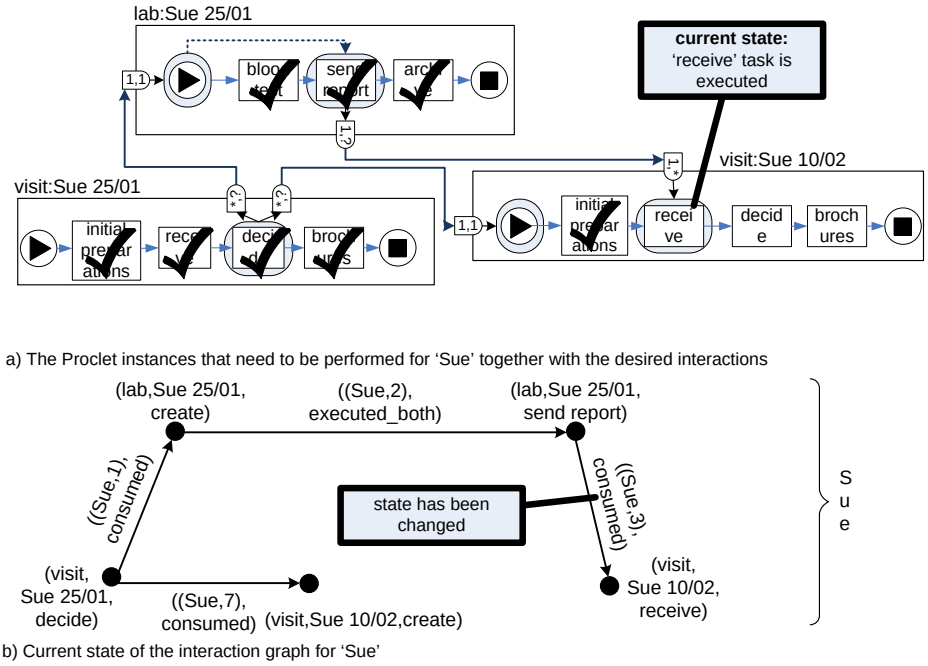


Fig. 8. The "receive" task of the "visit" Proclet instance is executed. As a performative has been sent which contains "(Sue,3)" as interaction identifier, the task may be completed. Subsequently, the interaction graph is updated.

Therefore, for the second scenario we deal with a Proclet class which operates at the level of an *individual patient* whereas another Proclet class operates at the level of a *group of patients*.

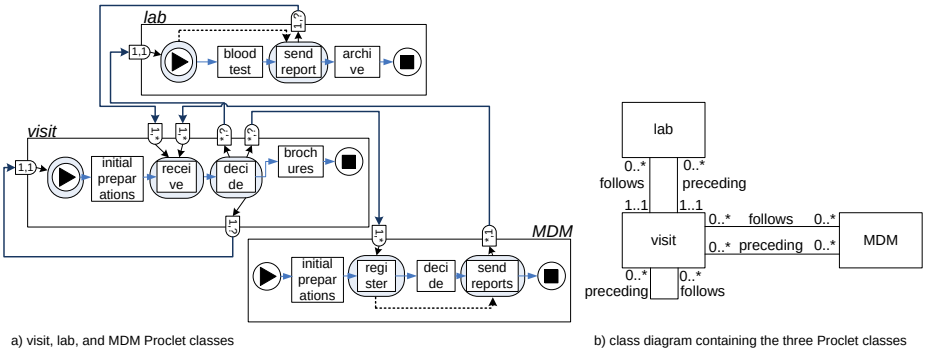


Fig. 9. The Proclet classes that are used for the second scenario.

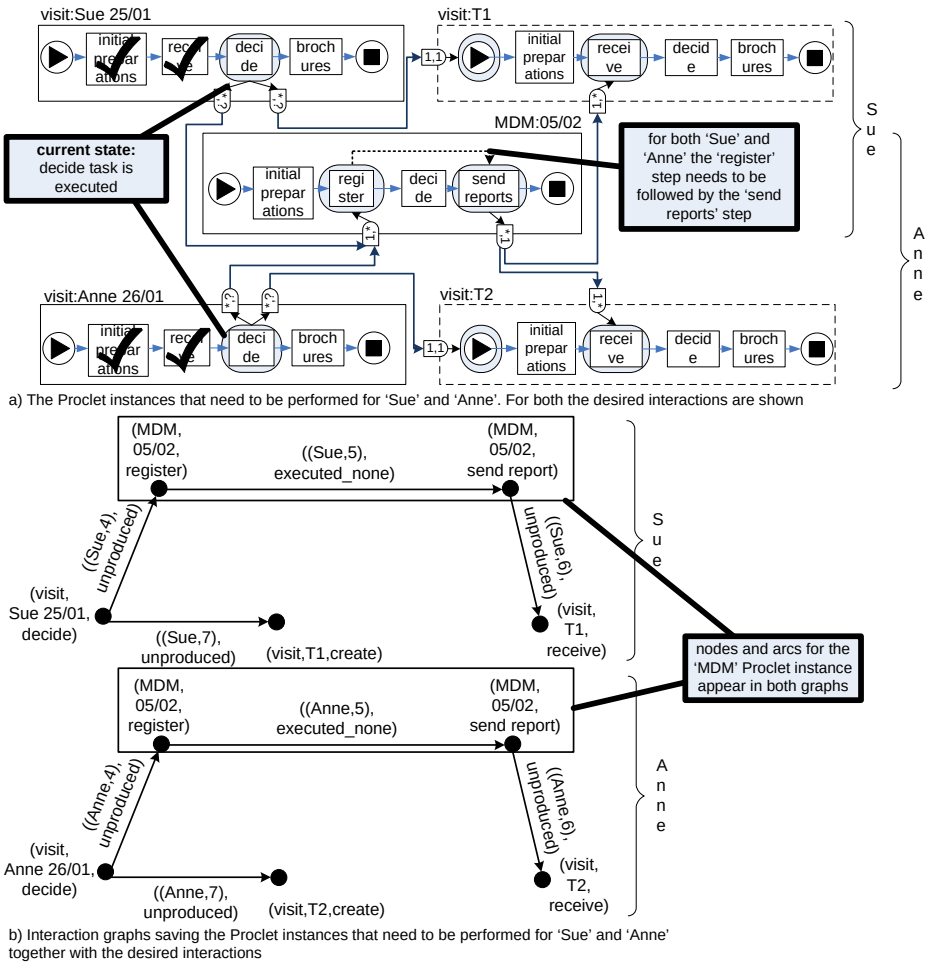
In Figure 9, we see three Proclet classes. The "visit" and "lab" Proclet classes have already been discussed in the first scenario. The "MDM" Proclet class is concerned with a weekly meeting in which gynecological oncology doctors discuss the medical status of multiple patients. For this meeting, multiple patients may be registered (task "register"). This can be seen by multiplicity * of the associated incoming port which indicates that multiple performatives may be received. During the "decide" task, the patients that are registered are discussed. Finally, for each patient that is discussed, a report may be sent out (task "send reports"). This is also represented by cardinality * of the associated outgoing port which indicates that a performative may be multicasted to multiple "visit" Proclet instances.

Note that there is an internal interaction defined from the "register" task to the "send reports" task. This internal interaction has as meaning that for every patient that is registered, it can be decided that its subsequent report needs to be sent to a specific Proclet instance (e.g. the second visit of the patient).

Obviously, the "MDM" Proclet class operates at another level of granularity, i.e. a group of patients, than the other Proclet classes. This can also be seen in the Figure 9b which shows a class diagram containing the Proclet classes.

The scenario that will be executed is visualized in Figure 10 and deals with two different patients. Here, for both "Sue" and "Anne" tasks from multiple Proclet instances will be performed. In particular, for "Sue" during execution of the "decide" task at the first visit (Proclet instance "visit:Sue 25/01") it is decided that a next visit is required (Proclet class "visit" with temporary instance identifier "visit:T1"). Moreover, "Sue" needs to be discussed during the multidisciplinary meeting for which already an instance is existing with identifier "MDM:05/02". Afterwards, the report needs to be used as input for the second visit.

For "Anne" exactly the same is decided during execution of the "decide" task. So, she is also discussed during the multidisciplinary meeting for which already an instance is existing with identifier "MDM:05/02". However, for her the instance that is existing for the first visit has identifier "Anne 26/01" and the instance for the second visit has temporary identifier "T2". As a result of executing the "decide" task for "Sue", which necessitates interactions with existing and future Proclet instances, an entity is created called "Sue" and for which subsequently an interaction graph has been created. For "Anne" exactly the same is done during the execution of the "decide" task for her but now an entity is created called "Anne" and a separate interaction graph has been created. For both entities the corresponding interaction graphs are shown in Figure 10. These interaction graphs are very similar to the first scenario. However, instead of a lab test, now for both patients an interaction with the "register" task of the "MDM:05/02" Proclet instance is required. This is followed by the execution of the "send reports" task after which a performative is send to the "receive" task of the second visit for both "Sue" and "Anne". Note that as no performatives have been sent yet, each arc in the interaction graph has either interaction state "unproduced" or "executed none". Moreover, it is important to mention that as



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. For both the desired interactions are shown

b) Interaction graphs saving the Proclet instances that need to be performed for 'Sue' and 'Anne' together with the desired interactions

Fig. 10. For both "Sue" and "Anne" it is shown how existing and future Proclet instances need to interact (Figure a). Additionally, for both, the interaction graph that is created during the execution of the "decide" task is shown (Figure b).

both patients are discussed during the multidisciplinary meeting, similar interaction nodes and similar interaction arcs for the "MDM:05/02" Proclet instance appear in the graphs of both "Sue" and "Anne". Below, for both patients, tasks will be executed for different Proclet instances. In this way, the impact of having the same interaction node in multiple graphs can be illustrated.

Step 1:

In Figure 11, the result of executing the "decide" task for both "Sue" and "Anne" is shown. Performatives are sent in order to create for both an instance of the "visit" Proclet class. Additionally, performatives are sent in order to register both of them for the multidisciplinary meeting. Note that as for "Sue" and

"Anne" their "decide" task is executed in different Proclet instances, the sending of performatives and updating the interaction graph for them occurs completely independently from each other.

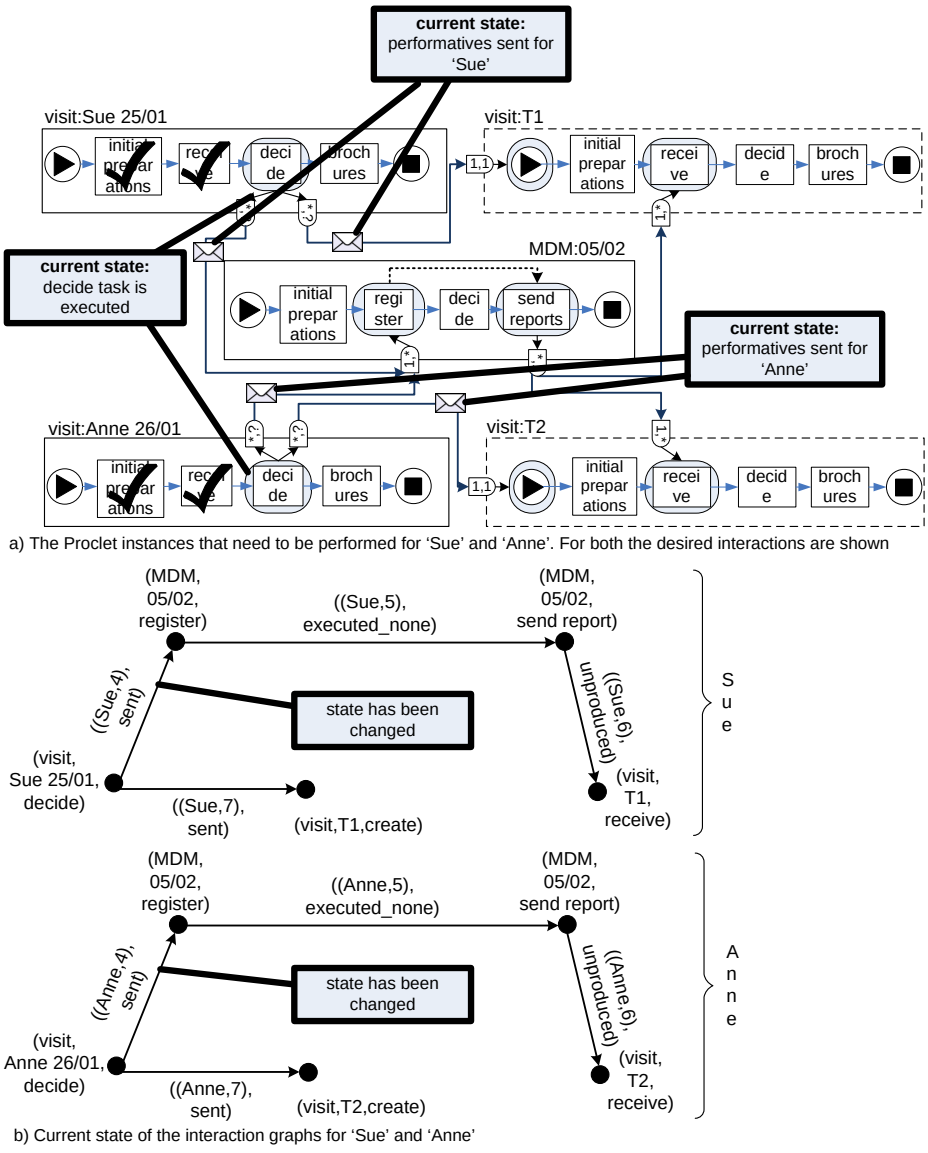


Fig. 11. As a result of executing the "decide" task for both "Sue" and "Anne" in total four performatives are sent. As a consequence, both interaction graphs are updated.

Subsequently, both interaction graphs are updated as expected. So, all the outgoing arcs of an interaction node that refer to a "decide" task that has been executed, have received the "sent" state.

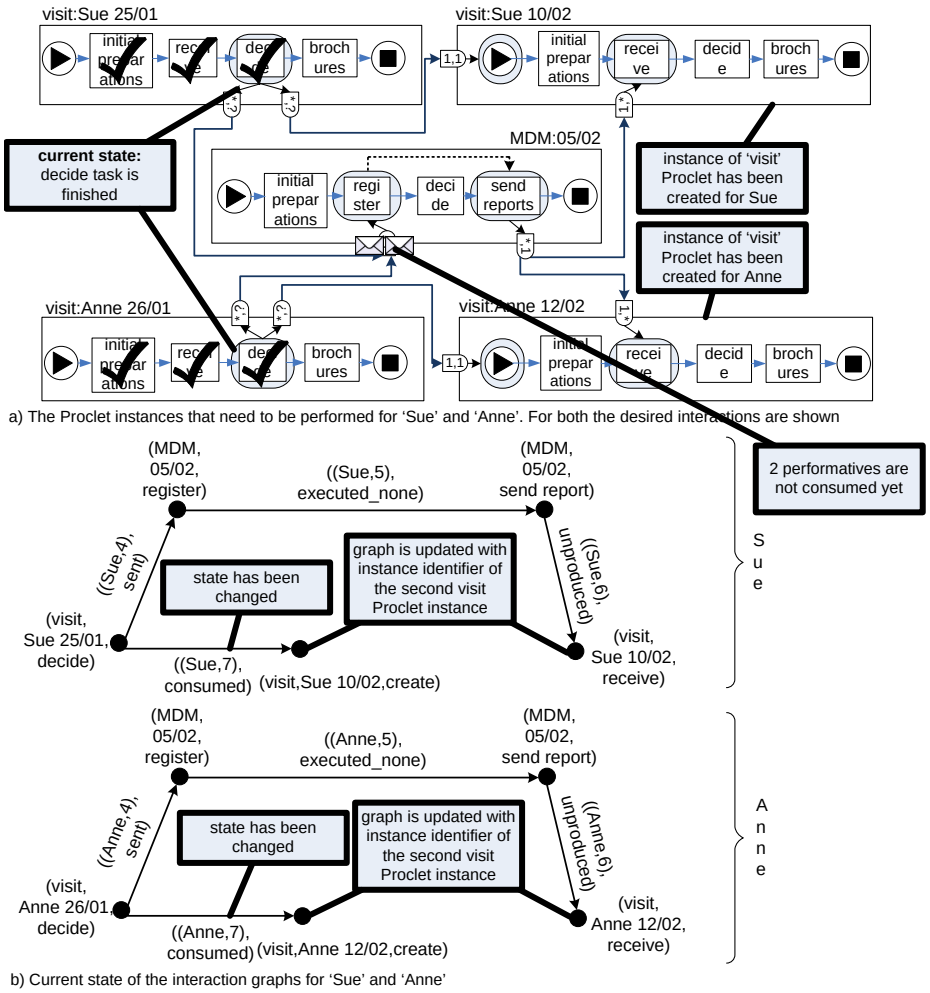


Fig. 12. Both for "Sue" and "Anne" an instance of the "visit" Proclet class is created. For the "receive" task of the "MDM" proclet instance, the relevant performatives are not consumed yet.

Step 2:

The next step is shown in Figure 12. As a consequence of receiving the required performatives, an instance of the "visit" Proclet class with instance identifier "Sue 10/02" has been initiated for "Sue" and an instance has been

created for "Anne" with instance identifier "Anne 12/02". Note that in the interaction graphs, the related interaction nodes and the related interaction arcs for them are updated accordingly, i.e. they have state "consumed". However, for the "register" task, the performatives have not been consumed yet as the preceding "initial preparations" task is still not executed. Therefore, the interaction arcs corresponding to these performatives still have state "sent".

Step 3:

Subsequently, for the "MDM" Proclet instance we perform the "initial preparations" task. Afterwards, the "register" task may be executed. The result can be see in Figure 13. For the "register" task of the "MDM" Proclet with instance identifier "05/02" we find in the interaction graph of both "Sue" and "Anne" an arc leading to the "(MDM,05/02,register)" node. So, in order to complete the task, for entity "Sue" a performative should be received with "(Sue,4)" as interaction identifier and for entity "Anne" a performative should be received with "(Anne,4)" as interaction identifier. As can be checked in Figure 12, these performatives have been sent. So, the task may be executed. For the receival of the performative which contains interaction identifier "(Sue,4)", the associated interaction arc with the same identifier is updated to "consumed" in the interaction graph of "Sue". For the performative which contains interaction identifier "(Anne,4)", the same is done but then for the interaction graph of "Anne".

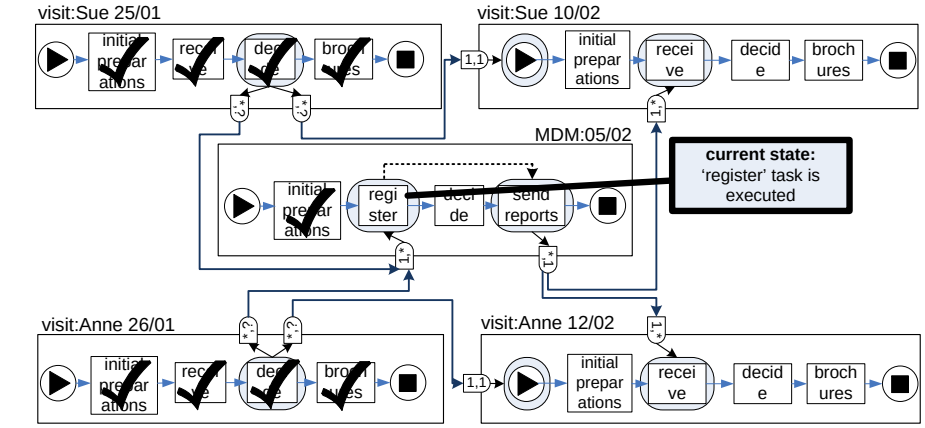
For the "register" task, both for "Sue" and "Anne" an internal interaction is defined for which the "(MDM,05/02,register)" interaction node is the source node. As this task will be completed, both for the interaction graphs of entities "Sue" and "Rose", the state of the corresponding outgoing interaction arc of the node is set to "executed source". Obviously, in order to perform a task it may be required to inspect and update multiple interaction graphs.

Note that if we abstract from the interaction identifier of an arc, then the internal interaction arc from the "(MDM,05/02,register)" node to the "(MDM,05/02,send report)" node is the same in both the interaction graph of "Sue" and "Anne". For these arcs it is important to see that the state is always the same and always changes simultaneously. This is due to the fact that both the tail and the head of these arcs refer to the same interaction node.

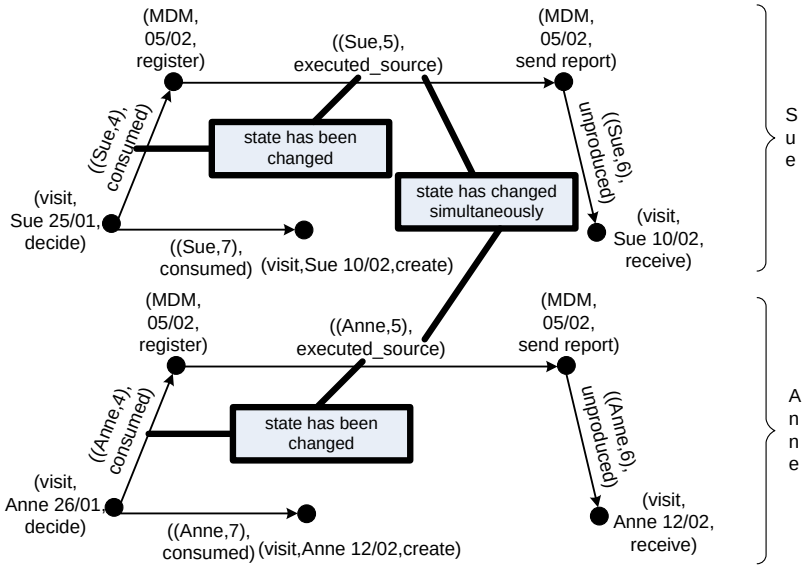
Step 4:

As a next step, the "send reports" task of the "MDM" Proclet instance is executed. First of all, as can be seen in Figure 14, for the "send reports" task, both for "Sue" and "Anne" an internal interaction is defined for which the "(MDM,05/02,send reports)" interaction node is the destination node. As we have seen earlier, the "register" task, which is the source of the two internal interactions, has already been executed, i.e. the state of the interaction arcs is "executed source". So, it is allowed to execute the task. Consequently, the state of the arcs is simultaneously updated to "executed both".

Note that if the task which belongs the source node of an internal interaction has not been executed yet, it is not allowed to execute the task which belongs to the destination node of the internal interaction. This is due to the fact that the meaning of an internal interaction is that first the task which corresponds



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. For both the desired interactions are shown



b) Current state of the interaction graphs for 'Sue' and 'Anne'

Fig. 13. The "register" task of the multidisciplinary meeting is performed. As performatives have been sent earlier which contain either "(Sue,4)" or "(Anne,4)" as interaction identifier, the task may be completed. Subsequently, the interaction graphs for both are updated.

to the source interaction node is executed, and then the task which belongs to the destination interaction node is executed.

Next, in the interaction graphs of both "Anne" and "Sue", the "(MDM,05/02,send report)" node has exactly one outgoing arc. Both arcs point to the "receive" task of the "visit" Proclet class. So, as can be checked in Figure 9, this means that both interaction arcs refer to the same port. However, the

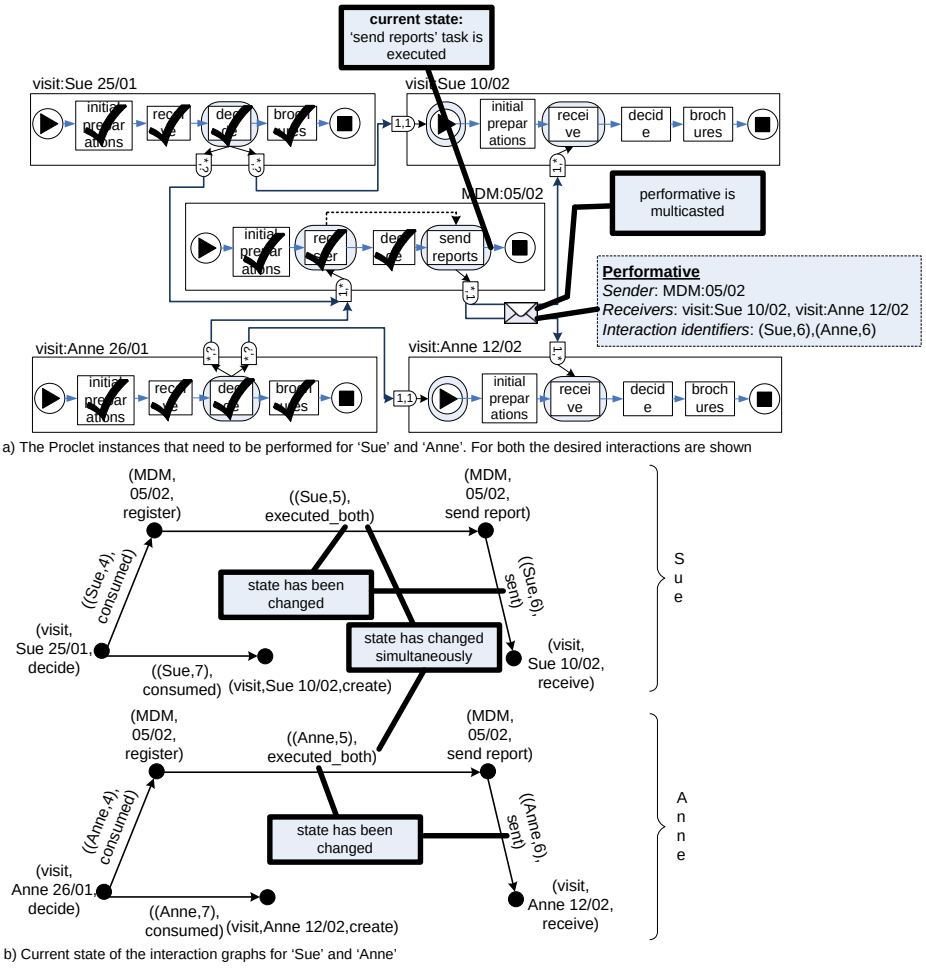
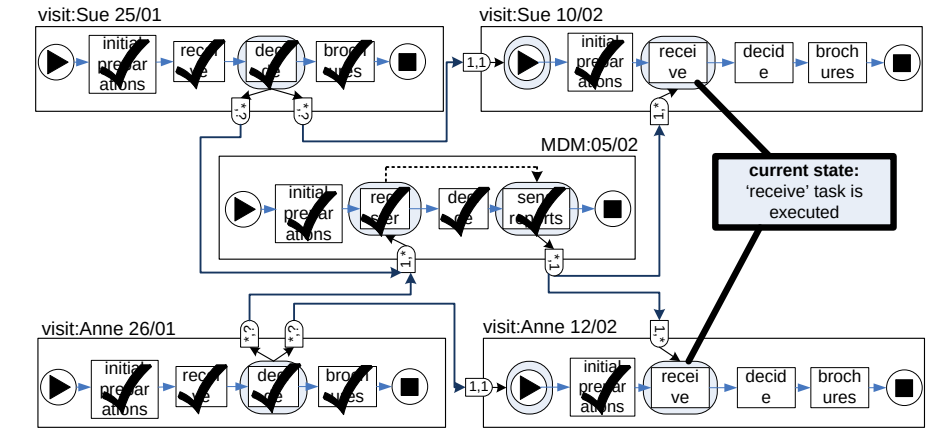


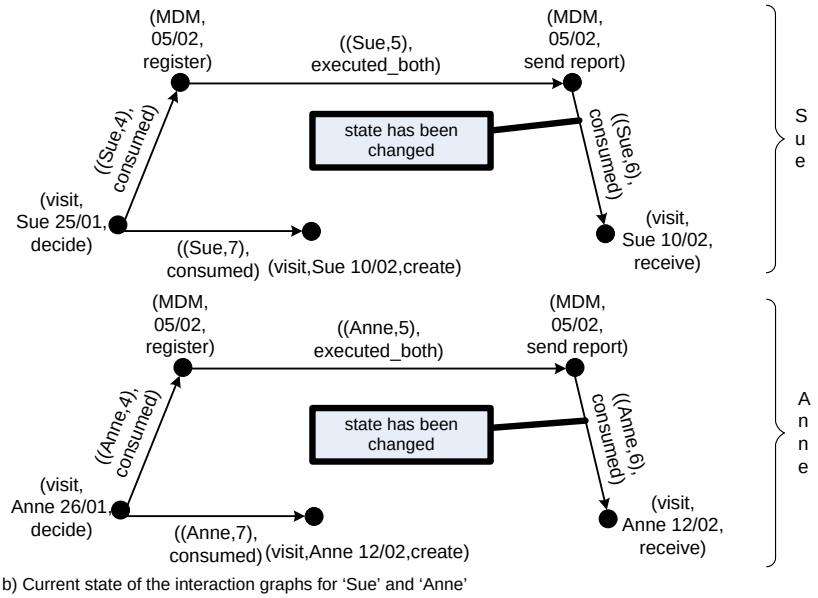
Fig. 14. The "send reports" task of the multidisciplinary meeting is performed. As a result, one performative is multicasted to the "visit:Sue 10/02" Proclet instance and the "visit:Anne 12/02" Proclet instance. Subsequently, the interaction graphs for both are updated.

instance identifiers are different. So, for "Sue" there needs to be a performative which is sent to the "(visit,Sue 10/02,receive)" Proclet instance with "(Sue,6)" as interaction identifier and for "Anne" there needs to be a performative which is sent to the "(visit,Anne 12/02,receive)" Proclet instance with "(Anne,6)" as interaction identifier. However, as the two potential performatives have the same sender and the "receive" task of the "visit" Proclet class as destination, only one performative will be created which is *multicasted* to the different receivers. So, as can be seen in Figure 14, there is a performative which has "MDM:05/02" as sender, "visit:Sue 10/02" and "visit:Anne 12/02" as receivers, and contains

"(Sue,6)" and "(Anne,6)" as interaction identifiers. Note that the corresponding interaction arcs are updated accordingly, i.e. the state is set to "sent".



a) The Proclet instances that need to be performed for 'Sue' and 'Anne'. For both the desired interactions are shown



b) Current state of the interaction graphs for 'Sue' and 'Anne'

Fig. 15. For both "Sue" and "Anne", the "receive" task is performed. As a performative has been sent which contains both "(Sue,6)" and "(Anne,6)" as interaction identifier, the task may be completed. Subsequently, the interaction graphs for both are updated.

Step 4:

Finally, as last step for both "Sue" and "Anne" the "receive" task of their second visit is executed (Figure 15). As a performative exists which contains

the interaction identifiers for both of them, the "receive" tasks for the two may be executed. Subsequently, the interaction state of the corresponding arc in the two graphs are updated to "consumed". However, note that for both of them, the execution of the "receive" task and the subsequent update of the interaction graph occurs completely independently from each other.

Proclets Framework So Far

In Figure 16, the concepts that have been introduced so far for the Proclets framework are visualized. For an interaction point we have indicated that it represents a specific point in a Proclet class at which interactions with other Proclets may take place. However, based on internal interactions that can be defined and interaction graphs of entities that can be extended, the notion of an interaction point can be more refined. Therefore, a distinction is made into a configuration, an inbox, and an outbox interaction point. The meaning of them is as follows.

Inbox Interaction Point: For one or more entities, performatives may be received. In this way, an inbox interaction point is only connected to input ports. For each input port, an arbitrary number of performatives may be received. An inbox interaction point is either connected to a task or an input condition.

In Figure 16, inbox interaction points are marked with the abbreviation "IB". For example, the "receive" task in the "visit" Proclet class is an inbox interaction point as for an entity only performatives are received.

Outbox Interaction Point: For one or more entities, performatives may be sent to multiple receivers. So, an outbox interaction point is only connected to output ports. For each output port, an arbitrary number of performatives may be sent. Note that by definition an output port is only connected to a task.

In Figure 16, outbox interaction points are marked with the abbreviation "OB". For example, the "send reports" task of the "MDM" Proclet class is an outbox interaction point as for multiple entities a performative may be sent.

Configuration Interaction Point: A configuration interaction point is the same as an outbox interaction point. Additionally, when an instance of a task is executed for such an interaction point, it is allowed to extend the interaction graph for multiple entities. In case for an entity such a graph does not yet exist, it will be created otherwise it can be extended. For each entity for which the interaction graph is extended, a human actor can nominate interactions that need to take place between existing and future Proclet instances. In Section 2.3 more details are provided about the extension of an interaction graph and for which entities this may occur.

In Figure 16, configuration interaction points are marked with the abbreviation "CP". For example, the "decide" task of the "visit" Proclet class is a

configuration interaction point as for an entity multiple performatives may be sent and the associated interaction graph may be extended.

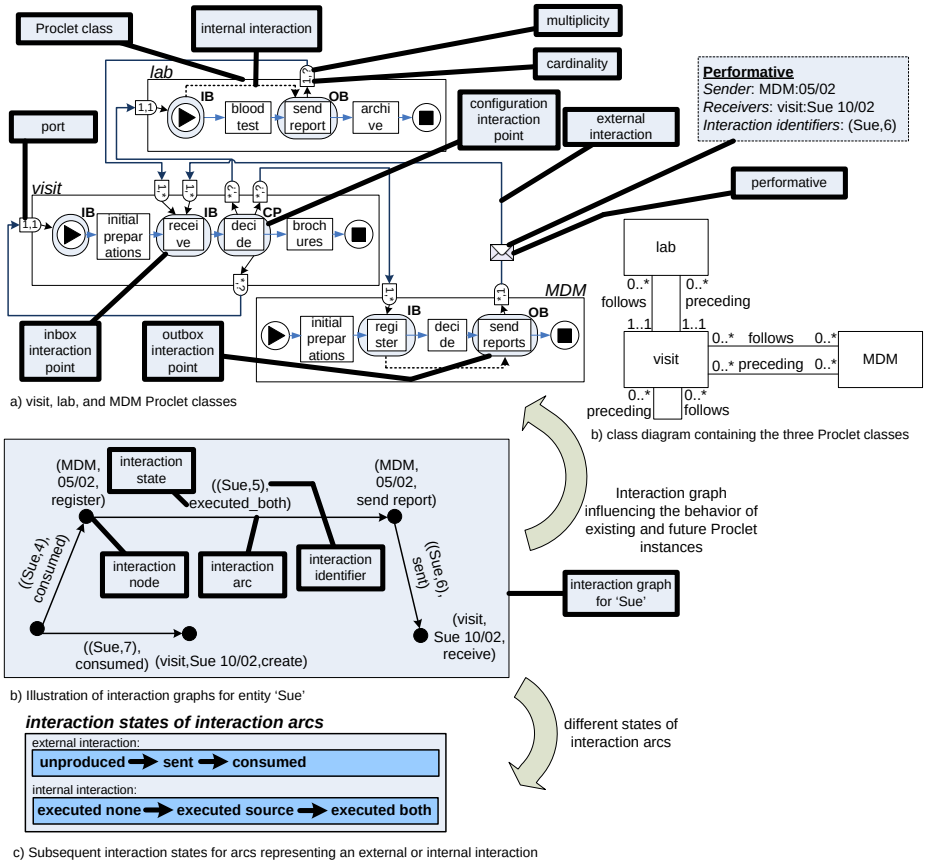


Fig. 16. Illustration of the concepts that have been introduced for the Proclets framework so far. Additionally, for interaction points, a distinction has been made into configuration (CP), inbox (IB), and outbox(OB) interaction points.

For an internal interaction this means that its source interaction point is always an inbox interaction point and its destination interaction point is always an outbox interaction point. For example, for the "MDM" Proclet class an internal interaction is defined which leads from the "register" inbox interaction point to the "send reports" outbox interaction point.

As indicated in Figure 16, interaction graphs that exist for entities, influence the behavior and interactions between existing and future Proclet instances. To this end, the interaction state of interaction arcs in these graphs are important.

For arcs referring to external interactions, the state transitions from "unproduced", "sent", to "consumed". The general meaning of each state is as follows.

unproduced: No performative has been produced yet for the interaction represented by the arc. Note that an interaction arc corresponds to a performative when the interaction identifier of the arc is contained in the "interaction identifiers" attribute of the performative.

sent: A performative has been produced for the interaction represented by the arc. In particular, the interaction identifier of the arc is contained in the "interaction identifiers" attribute of the performative that has been produced.

consumed: In case the head of the arc refers to an input condition of a Proclet class, state "consumed" is obtained when the corresponding performative has been 'consumed' in order to create an instance of a particular Proclet class. In case the head of the arc refers to a task instance, state "consumed" is obtained when the corresponding performative has been "consumed" in order to complete the task instance. Note that a task instance may be completed if for all interaction nodes that belong to the task instance, all the incoming arcs have state "sent". However, an exception to the latter is possible in case of an exception that is handled. This will be discussed in more detail in Section 2.2.

For arcs referring to internal interactions, the state transitions from "executed none", "executed single", to "executed both". The general meaning of each state for an arc is as follows.

executed none: if an arc has state "executed none" both the source and destination internation nodes have not been executed. That is, for a source interaction node which is linked to an input condition of a Proclet class, this means that no instance of that Proclet class has been created yet. For a source interaction node which is linked to a task instance, the task instance has not been executed yet. For the destination interaction point, which is always linked with a task instance, this also means that the task instance has not been completed yet.

executed single: For a source interaction node which is linked to an input condition of a Proclet class, this means that an instance of the Proclet class has been created. For a source interaction node which is linked to a task instance, this means that the task instance has been executed.

executed both: As a follow-up on the previous state, now the task instance which is linked to the destination interaction point, has been executed.

With regard to the state of an arc, it should be noted that the same interaction arc may be found in multiple interaction graphs (when abstracting from the interaction identifier). So, for these arcs, their tails refer to the same interaction node and their heads refer to the same interaction node. Consequently, the state of these arcs will always change simultaneously. For example, for the second scenario, we have seen in Figure 14, that the state of the arc from the "register" task to the "send report" task changed simultaneously from "executed source" to "executed both" in the interaction graphs of both "Sue" and "Anne".

2.2 Exception Handling

The interactions that are defined in the interaction graphs are nominated by a human actor. Therefore, these interactions need to occur. However, for them, several kinds of exceptions may occur in which they cannot take place anymore. In this section, we discuss the different situations in which an exception may occur and how they can be handled. First, exceptions that occur in the context of executing a task are discussed. Next, exceptions in the context of Proclet instances that are canceled or completed are elaborated upon. Note that the exceptions that may occur are discussed by referring to the two scenarios that have been presented earlier.

Execution of a Task

In order to illustrate an exception that may occur in the context of executing a task we refer back to the first scenario. However, now we assume the situation in which the "send report" task of the "lab" Proclet instance has not been executed. Also, we assume that we are currently executing the "receive" task of the Proclet instance for the second visit. Note that the latter task is linked to an inbox interaction point. This situation is depicted in Figure 17.

Time-Out Value

As no performative has been sent from the "send report" task of the "lab" instance to the "receive" task of the second visit, it is not allowed to complete the "receive" task, i.e., the state of the interaction arc for the respective interaction is still "unproduced". As a result, an exception occurred for the "visit" Proclet instance. We distinguish two different approaches in order to deal with such a situation. The first approach is to reserve more time in which to receive the missing performative. This can be supported by defining a *time-out* value for an inbox interaction point. The value defines how much time needs to be reserved in which to wait for missing performatives. Once the waiting time has lapsed, a human user is requested how to deal with the situation. Note that the time-out value can be mapped to any unit of time. For example, a value of "5" may belong to 5 minutes.

Another approach to deal with the situation is to force complete the task and thus not to wait for missing performatives. This situation is illustrated in Figure 17c. As the interaction with the "send report" task will not take place, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has received state "failed" in order to indicate that the interaction will not take place anymore.

Exception Interaction Point

However, as a follow-up on the approach to force complete the task instance it may be desired to use the result of the lab test as input for a third patient visit. This requires, for the entity that is affected by the exception, that it is possible to extend an interaction graph as part of the latter exception handling strategy. For entity "Sue" this is illustrated in Figure 18.

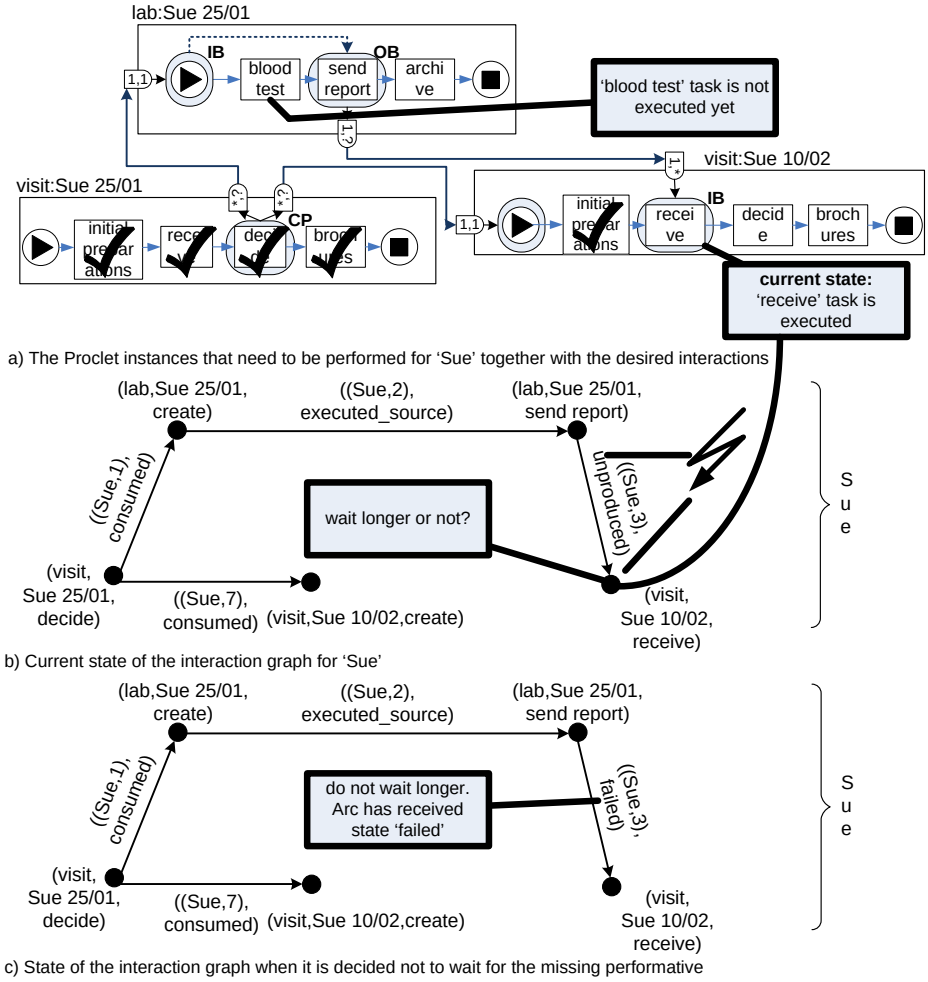


Fig. 17. Illustration of an exception that may occur in the context of a task that is executed. Here, no performative has been sent yet from the "lab" instance to the Proclet instance for the second visit. As a result, a problem occurs when executing the "receive" task for the second visit. One solution is to wait longer for the missing performative. Another solution is to force complete the task and to mark the corresponding interaction arc as having state "failed".

In order to be able to extend an interaction graph in case an exception occurs for a certain Proclet instance, a so-called *exception interaction point* may be defined for a Proclet class. An exception interaction point is similar to a configuration interaction point. However, only for the entities that are effected by the exception, the interaction graph may be extended.

In Figure 18a, an exception interaction point has been defined for the "visit" Proclet class. By following the outgoing arcs, it can be seen that an instance of the "lab" Proclet class and an instance of the "visit" Proclet class may be created in case an exception occurs for an instance of the "visit" Proclet class. Subsequently, in Figure 18b, it is illustrated for entity "Sue" how the interaction graph is extended using the exception interaction point of the "visit" Proclet class. That is, starting from the "visit:Sue 10/02" Proclet instance for which the exception occurred, it is decided to start an instance of the "visit" Proclet class. Next, the result of the "send report" task of the "lab" Proclet instance is used as input for the "receive" task of the new "visit" Proclet instance.

The resulting interaction graph for "Sue" can be seen in Figure 18c. As can be seen, the "(visit,Sue 10/02,exception)" node has been added representing the exception that occurred. Starting from that node, a next instance of the visit Proclet class is created (node "(visit,T3,create)"). Finally, the result of the lab test is used as input for the third patient visit (node "(visit,T3,receive)").

General

The approach for executing a task instance for which not all performatives have been received can be generalized as follows. This is schematically visualized in Figure 19 where two interaction graphs are shown. Remember that a task instance for which only performatives can be received is always linked to an inbox interaction point.

For a task instance "B", a corresponding interaction point "Bi" may be found in multiple interaction graphs. In case for task instance "B" not all performatives have been received, i.e. not all incoming arcs for the interaction nodes named "Bi" have state "sent", then an exception occurs for the Proclet instance in which the task instance occurs (in the figure, for entity "A" there is one incoming arc having state "unproduced" and for entity "B" there are two incoming arcs having state "unproduced"). Now, two options are possible.

According to the time-out value defined for the interaction point that belongs to task instance "B", a human actor may decide to reserve more time in which to receive missing performatives.

Another option is to force complete the task instance thereby ignoring the performatives that still need to be received. In that case, for all incoming interactions arcs for the interaction nodes named "Bi" which have state "unproduced", the state is changed into "failed" (in Figure 19, for entity "A" there is now one incoming arc having state "failed" and for entity "B" there are two incoming arcs having state "failed"). Additionally, for all affected entities, the exception graph may be extended. An entity is affected by the exception if for one or more interaction arcs in the corresponding interaction graph the state had to be changed into "failed". Finally, the extension of the interaction graphs may be done via the exception interaction point of the Proclet class for which the exception occurred.

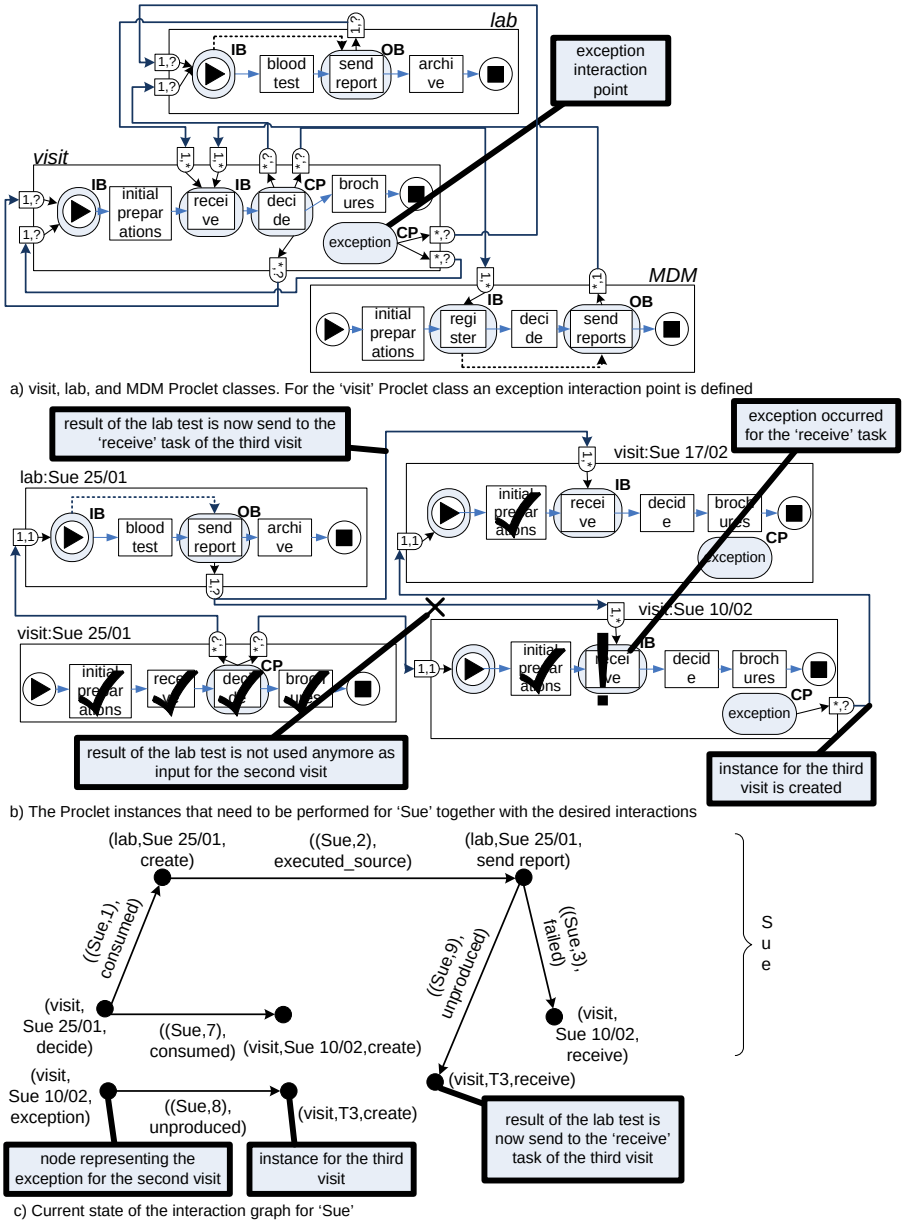


Fig. 18. An exception interaction point may be defined for a Proclet class (Figure a). As for "Sue" an exception occurred for the "visit:Sue 10/02" Proclet instance, the exception interaction point has been used for creating a next instance of the "visit" Proclet class which represent the third visit (Figures b and c). Also, the result of the "send report" task is used as input for the third visit.

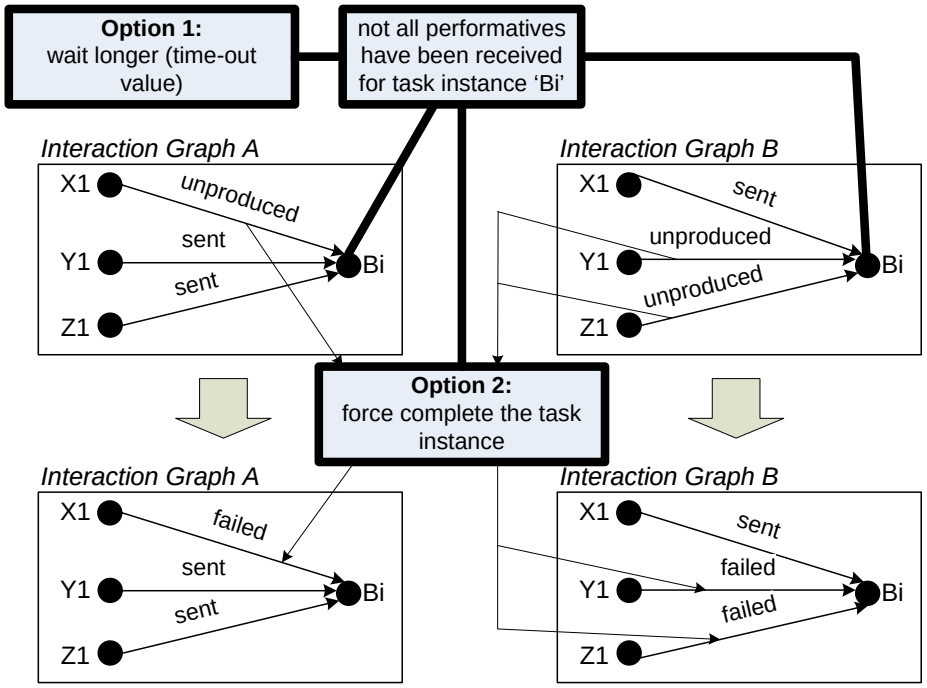


Fig. 19. For the general case it is illustrated how an exception is handled in the interaction graphs if for a task instance not all required performatives have been received yet.

Instance Cancelation or Completion

A Proclet instance may be canceled or completed while still not all desired interactions occurred for it. First, we illustrate this kind of exception and the handling of it in the context of the first scenario. Afterwards, the exception caused by a case cancelation / completion and its handling is explained for the general case.

Outbox Interaction Point

One kind of exception that may occur in the context of canceling a Proclet instance is related to an outbox interaction point. This is illustrated in Figure 20. Here, for the first scenario, we assume that the "lab" Proclet instance is canceled and that no tasks for it have been executed yet (Figure 20a). Also, for the second visit, no tasks have been performed yet.

If in the interaction graph of entity "Sue" we look to the arcs that relate to the canceled "lab" Proclet instance (Figure 20b) then we see that the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has state "unproduced". This means that no performative has been sent yet from the "send report" task to the "receive" task of the second visit. Note that the "send report" task is linked with an outbox interaction point. As a consequence,

we have an exceptional situation as the respective performative will never be sent in the future, i.e. the defined interaction will never occur.

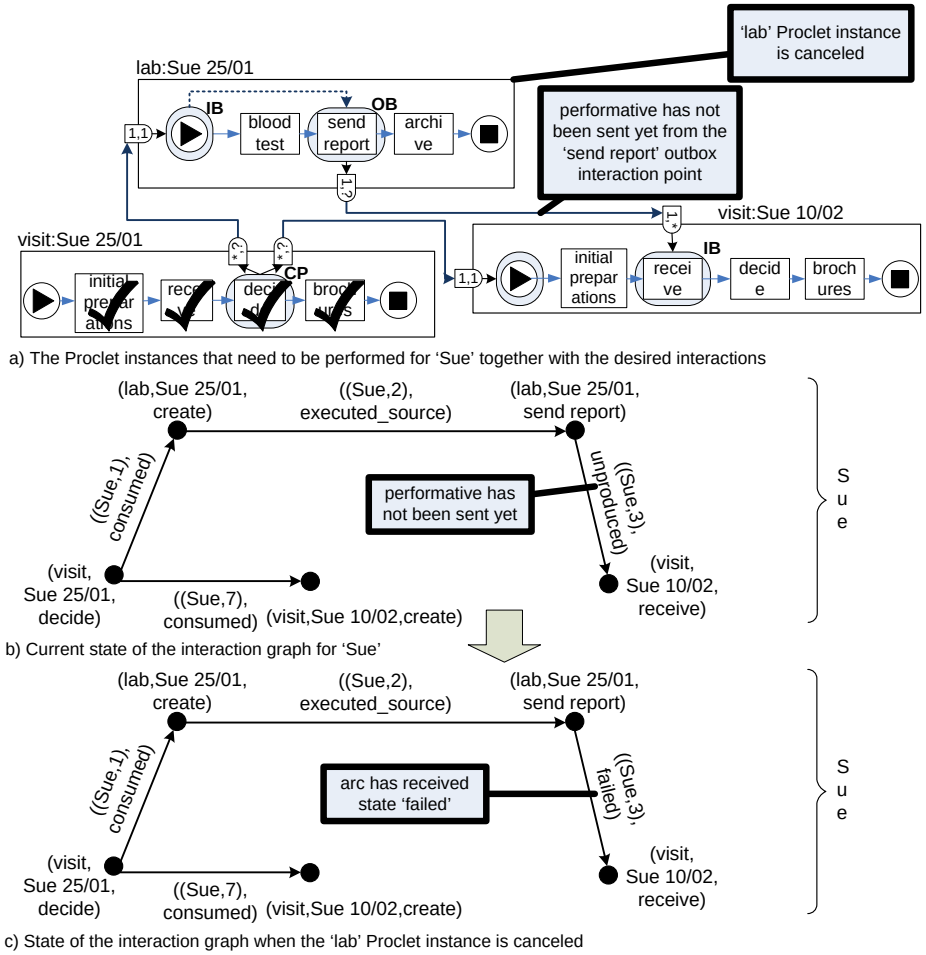


Fig. 20. Illustration of an exception that may occur if a certain Proclet instance is canceled. For a task instance which relates to an outbox interaction point, the required performative has not been sent.

Subsequently, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node receives state "failed" (Figure 20c). Moreover, similar to the previous exception, it is offered to a human actor to extend the interaction graph for "Sue". As the exception occurred for the "lab" Proclet instance due to its cancelation, the graph may be extended by using the exception interaction point of the "lab" Proclet class (not shown in Figure 20a). Note that

if the "lab" Proclet instance would have been completed and also no performative had been sent from the "send report" task (e.g. due to a choice in the process), then the same procedure would be followed as described above.

Inbox Interaction Point

Another exception that may occur in the context of canceling a Proclet instance is related to an inbox interaction point. This is illustrated in Figure 21. Here, for the first scenario, we assume that the "visit" Proclet instance for the second visit is canceled and that no tasks for it have been executed yet (Figure 21a). Also, for the "lab" Proclet instance we assume that no tasks have been executed.

Looking to the interaction graph of "Sue" (Figure 21b), for the arcs that relate to the canceled "visit" instance we see that the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node has state "unproduced". So, no performative has been sent yet from the "send report" task to the "receive" task of the canceled "visit" Proclet instance. Note that the "receive" task is linked with an inbox interaction point. So, we have now an exceptional situation as the performative which still needs to be sent from the "send report" task can never be consumed by the "received" task anymore, i.e. the defined interaction will never occur.

Subsequently, the arc from the "(lab,Sue 25/01,send report)" node to the "(visit,Sue 10/02,receive)" node receives state "failed" (Figure 21d). Also here, it is offered to a human actor to extend the interaction graph for "Sue". However, as the exception occurred for the "visit" Proclet instance due to its cancelation, the graph may be extended by using the exception interaction point of the "visit" Proclet class.

In Figure 21c, a comparable situation is shown. Here, the corresponding interaction graph is presented in case a performative is sent from the "send report" task but that it has not been consumed yet by the "receive" task of the canceled "visit" Proclet instance. Also here, by the cancelation of the Proclet instance for the second visit, the defined interaction will never occur. So, the respective interaction arc will get state "failed" (Figure 21d) and for entity "Sue" the interaction graph may be extended.

Note that a similar procedure is followed if a Proclet instance is completed and that for a task instance, corresponding to an inbox interaction point, not all performatives have been received yet.

General

The approach for canceling / completing a Proclet instance for which not all desired interactions have taken place can be generalized as follows. This is schematically visualized in Figure 22a and b where for both two interaction graphs are shown.

The first situation is depicted in Figure 22a where Proclet instance "Y" is canceled / completed. For Proclet instance "Y", multiple outbox interaction nodes may be found in multiple interaction graphs. In case for these interaction nodes (illustrated by nodes "Yi" and "Yj" in Figure 22a) there is at least one

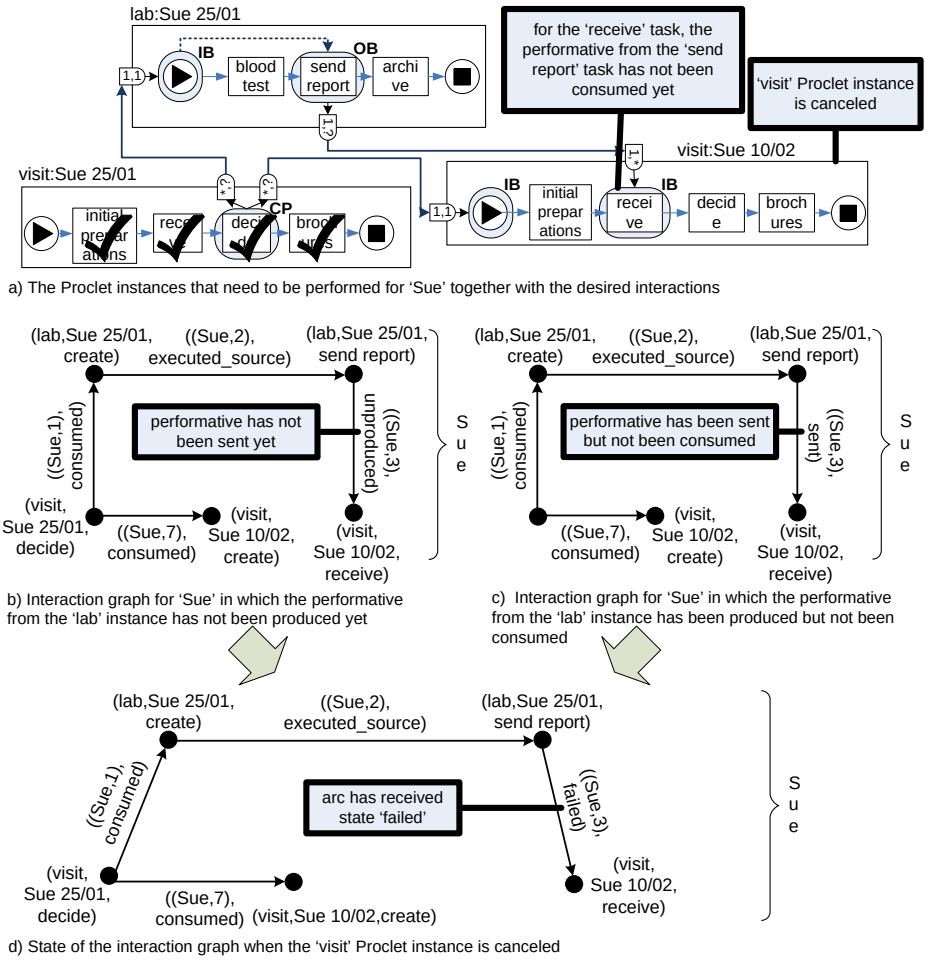
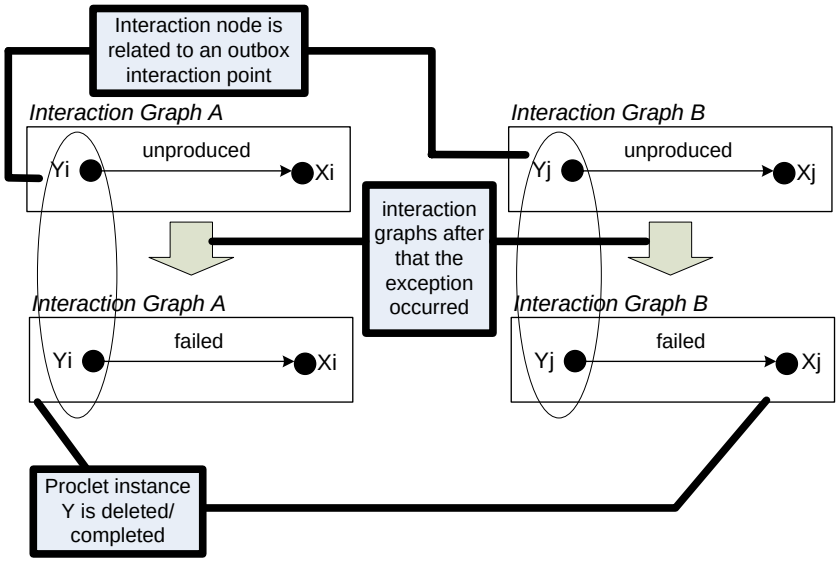


Fig. 21. Illustration of an exception that may occur if a certain Proclet instance is canceled. For a task instance which relates to an inbox interaction point, the required performative has not been received.

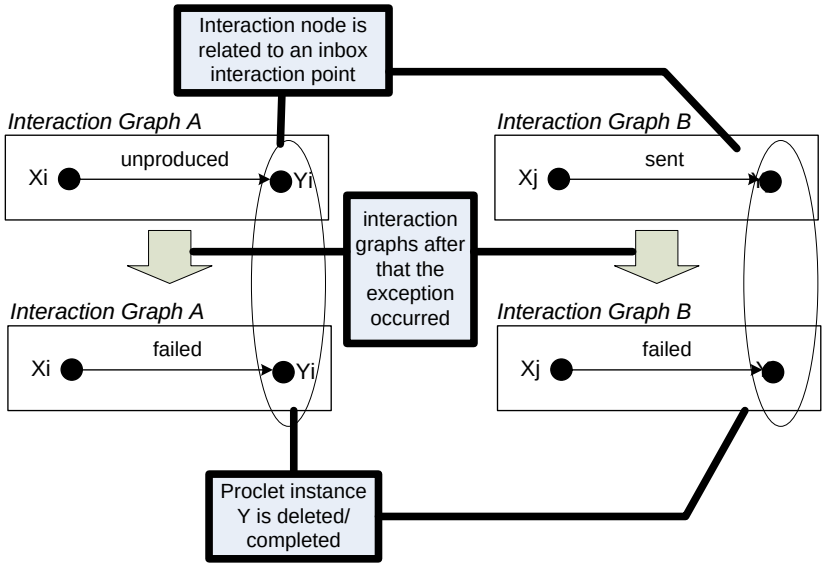
outgoing arc having state "unproduced", then an exception occurs for Proclet instance "Y', i.e. for such an arc the defined interaction can never occur.

Consequently, the latter mentioned arcs receive state "failed" (illustrated at the bottom of Figure 22a). Furthermore, for the affected entities, the opportunity is offered to extend the interaction graph.

The second situation is depicted in Figure 22b where Proclet instance "Y" is canceled / completed. For Proclet instance "Y", multiple inbox interaction nodes may be found in multiple interaction graphs. In case for these interaction nodes (illustrated by nodes "Yi" and "Yj" in Figure 22b) there is at least one incoming arc having state "unproduced" or "sent", then an exception occurs for



a) Proclet instance Y is deleted. For the deleted instance, all outgoing interaction arcs related to an outbox interaction point which have state 'unproduced' receive state 'failed'.



b) Proclet instance Y is deleted. For that instance, all incoming interaction arcs related to an inbox interaction point which have state 'unproduced' or 'sent' receive state 'failed'.

Fig. 22. For the general case it is illustrated how an exception is handled in the interaction graphs if a Proclet instance is canceled or completed.

Proclet instance "Y", i.e. for such an arc the defined interaction can never occur.

Consequently, the latter mentioned arcs receive state "failed" (illustrated at the bottom of Figure 22b).

2.3 Extending an Interaction Graph

In the previous sections, we have discussed about different aspects of the Proclets framework. When elaborating on the two scenarios always interaction graphs were given that were already defined. In this section, we elaborate upon how an interaction graph is extended for an entity.

The extension of an interaction graph is based on the current interaction graph of an entity. Also, it is based on the interaction points that exist for Proclet classes and how they are connected, i.e. internal and external interactions that exist between these interaction points. First, we illustrate the extension of an interaction graph in the context of the second scenario. Afterwards, it is explained for the general case.

Scenario

As shown in Figure 23a, for the second scenario we assume that for the first visit the "decide" task is currently executed. Moreover, we assume that an instance of the "MDM" Proclet class is running. The "decide" task is linked with a configuration interaction point which means that the interaction graphs of multiple entities may be extended. For patient "Sue" we want to achieve that a second visit is created for her and that she is discussed during the multidisciplinary meeting. Also, the result of the discussion for her during the multidisciplinary meeting needs to be used as input for the second visit. In order to achieve this, an interaction graph needs to be created.

Step 1:

However, at the moment the "decide" task is executed, no interaction graph for "Sue" is existing yet. Therefore, as part of the "decide" task, we indicate that for entity "Sue" an interaction graph needs to be created. The result of this action can be seen in Figure 23c. Here, we see that there is an interaction node with name "(visit,Sue 25/01,decide)" which refers to the "decide" task that is currently executed. Moreover, it is indicated that the node is *active*. This means that for the node currently a Proclet instance exists which has the same instance identifier. In that way, for the node interactions may be nominated which will potentially occur in the future. The interactions that may be nominated can be seen by looking to the Proclet classes and how they are connected in Figure 23b. In particular, if we look to the "decide" task in the "visit" Proclet class then we see that it has three outgoing ports. For them the following can be observed which is indicated by the dotted arcs between the Proclet classes and the interaction graph of "Sue".

– An outgoing port is connected with an inbox interaction point which is linked with the input condition of the "lab" Proclet class. As a result, for the "lab" Proclet class multiple instances may be created. However, note that this may be constrained by the cardinalities and multiplicities of the associated ports.

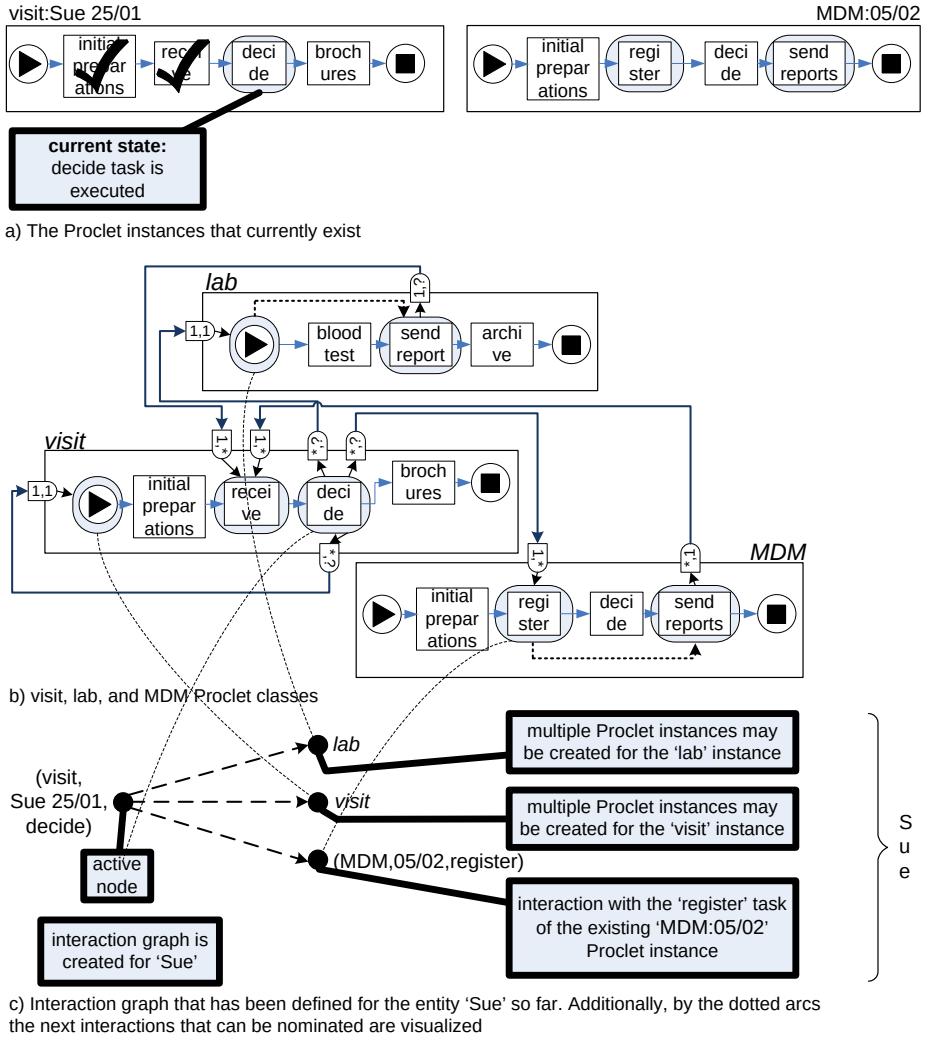


Fig. 23. Creating an interaction graph for "Sue". The possible interactions starting from the "(visit,Sue 25/01,decide)" node are indicated by dotted arcs.

– An outgoing port is connected with an interaction point which is linked with the input condition of the "visit" Proclet class. As a result, multiple instances of the "visit" Proclet class may be created.

– An outgoing port is connected with an inbox interaction point which is linked with the "register" task of the "MDM" Proclet class. As currently an instance of the "MDM" Proclet class is existing which has instance identifier "05/02", it is possible to have an interaction with the "register" task of that Proclet

instance. Note that we abstract from the current state of the "MDM:05/02" Proclet instance.

In Figure 23c, the interactions that may be nominated, starting from the "(visit,Sue 25/01,decide)" node, are indicated by dotted arcs. For them, a human actor decides to create one instance of the "visit" Proclet class which represents the second visit of "Sue". Additionally, it is decided to have an interaction with the "register" task of the existing "MDM" Proclet instance in order to register "Sue" for the multidisciplinary meeting.

Step 2:

The new interaction graph can be seen in Figure 24. As can be seen, there is an arc leading from the "(visit,Sue 25/01,decide)" node to the "(MDM,05/02,register)" node which represents the interaction with the "register" task of the "MDM" instance. The arc leading to the "(visit,T1,create)" node represents the instance of the "visit" that will be created in order to have the second visit of the patient. Note that a temporary instance identifier is used ("T1") because the instance still needs to be created. For the new interaction arcs, it can be seen that their state is "unproduced" as no performatives have been sent yet. Also, each arc obtains a unique instance identifier.

Additionally, in Figure 24a, we can see two nodes which are active. For the "(visit,Sue 25/01,decide)" node, the possible interactions are not shown in order to not clutter the graph. However, for example, it is still possible to create an additional instance of the "visit" Proclet class. The other active node is the "(MDM,05/02,register)" node for which the new interactions that can be nominated are indicated via dotted arcs. That is, the "(MDM,05/02,register)" node matches with the "register" task interaction point of the "MDM" Proclet class. For that interaction point, an internal interaction is defined that has the "send report" interaction point as its destination. So, an internal interaction with the "send report" task of the "MDM" Proclet instance is possible.

As can be seen in the figure, the "(visit,T1,create)" node is not active. The node matches with the interaction point that corresponds to the input condition of the "visit" Proclet class (Figure 24a). However, for that interaction point no outgoing external and internal interactions have been defined. So, no interactions are possible starting from the "(visit,T1,create)" node and subsequently, the node is not active.

Step 3:

The new interaction graph can be seen in Figure 25b. The internal interaction for the "MDM" Proclet instance has been added in order to use the outcome of the multidisciplinary meeting as input for the second visit. Note that the associated arc has a unique identifier and that the state is "executed none" as nothing has happened yet.

There are three active nodes in the interaction graph of which only for the "(MDM,05/02,send reports)" node the new possible interactions are visualized. That is, the "(MDM,05/02,send report)" node matches with the "send reports" interaction point of the "MDM" Proclet class (see Figure 25a). For that outbox interaction

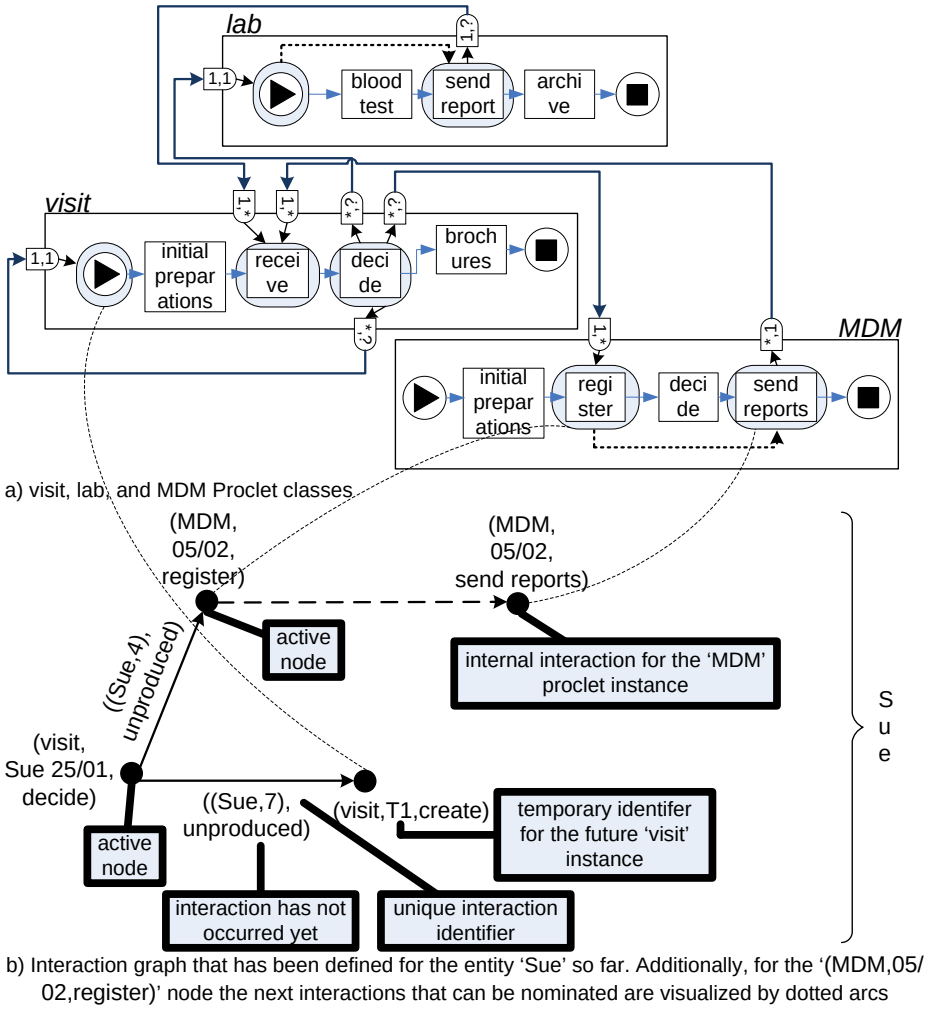


Fig. 24. Extending the interaction graph for "Sue". The possible interactions starting from the "(MDM,05/02,register)" node and the "(visit,T1,create)" node are indicated by dotted arcs.

point there is one outgoing port which is linked with the "receive" interaction point of the "visit" Proclet class. As this is an inbox interaction point, an interaction is possible with the existing "visit" Proclet class which has instance identifier "25/01" (node "(visit,Sue 25/01,receive)"). However, in the graph of entity "Sue" we see an interaction node for a "visit" Proclet instance with temporary instance identifier "T1" which represents the second visit. As it will exist in the future, also interactions may be defined for it. As a consequence,

an interaction with the "receive" task of this future instance is possible (node "(visit,T1,receive)").

The resultant interaction graph is shown in Figure 25c. As can be seen, an interaction has been added such that the result of the multidisciplinary meeting is used as input for the second visit. Note that a unique interaction identifier is used and that the state is "unproduced".

In case for the second visit the "'decide" task is executed, the interaction graph can again be extended. However, here it is not needed to provide "Sue" as entity identifier in order to extend the graph. That is, in the interaction graph of "Sue", we already find the "create" and "receive" node for the second visit. In that way, it is already indicated that the entity is relevant for the second visit and the graph for it may be extended once allowed.

General

In the scenario, we have seen how an interaction graph is extended taking into account existing and future Proclet instances. Also, external and internal interactions that are defined for Proclet classes are taken into account. Now it is explained for the general case how an interaction graph is extended.

Relevant entities:

When a task instance corresponding to a configuration interaction point is executed, it is allowed to extend the interaction graphs of entities. First, it is important to know for which entities the interaction graph may be extended. Here we distinguish two different cases:

- a human actor has provided the names of entities for which a corresponding interaction graph needs to be created and for which interactions can be defined.
- for the entity already an interaction has been defined for the Proclet instance of which the task instance is executed. So, in the graph of the entity, already a node exists which has the same instance identifier as the Proclet instance for which the task instance is executed. In this case we say that the entity is relevant for the Proclet instance.

Furthermore, in case an exception occurs for a certain Proclet instance, for the entities that are affected by the exception, the corresponding interaction graph may be extended. Note that an entity is affected by an exception if for one or more interaction arcs in the corresponding interaction graph the state had to be changed into "failed".

Extension:

In Figure 26, the procedure that is followed for extending an interaction graph is visualized. Before starting the procedure, it is first identified whether the instance of the configuration interaction point is itself present in the interaction graph. If not, an interaction node for it is added. Afterwards, the procedure is started by determining which nodes in the graph are *active*. A node is active if for its Proclet instance identifier still a Proclet instance exists with the same

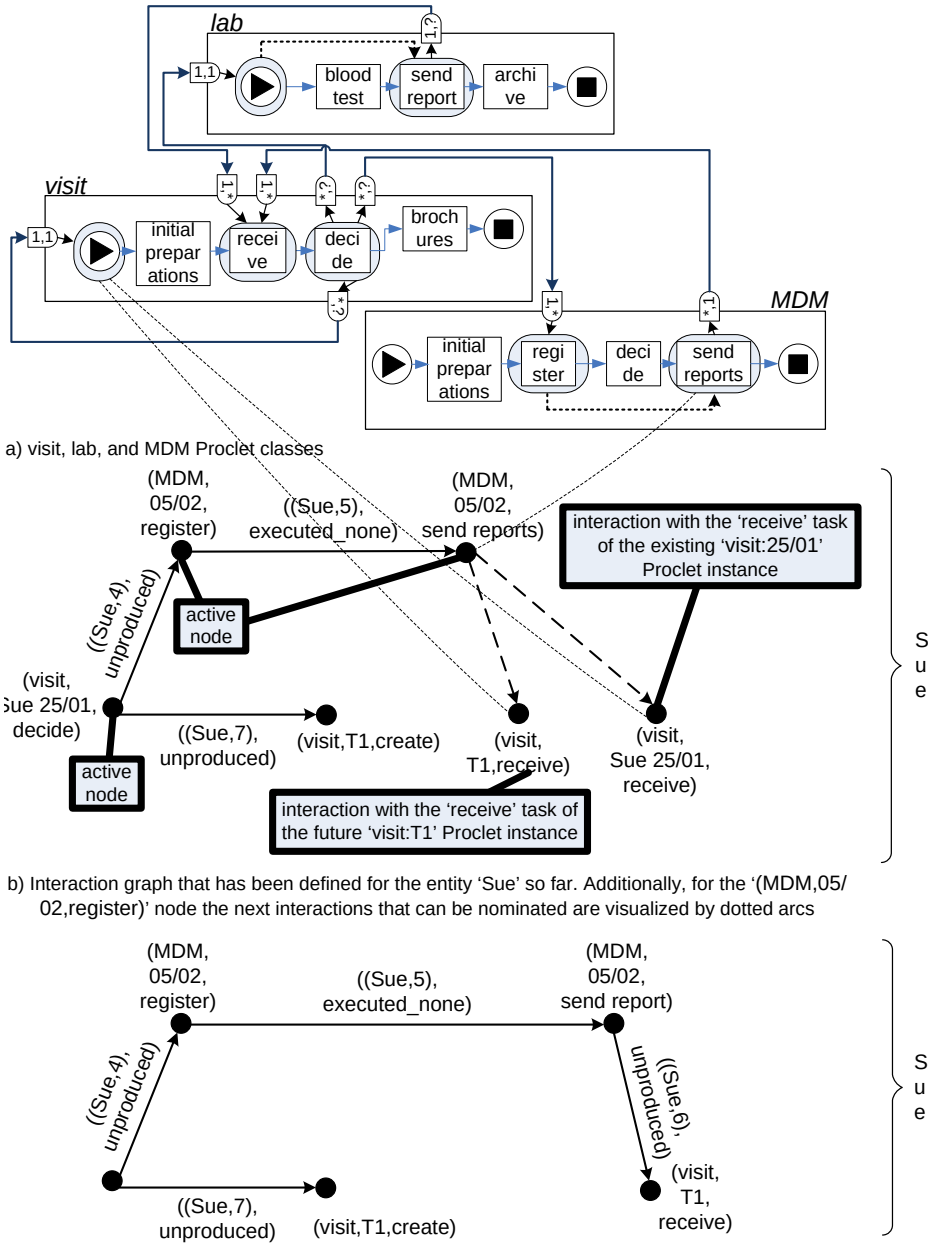


Fig. 25. Extending the interaction graph for "Sue". The possible interactions starting from the "(MDM,05/02,send report)" node are indicated by dotted arcs. The resultant graph is shown at the bottom.

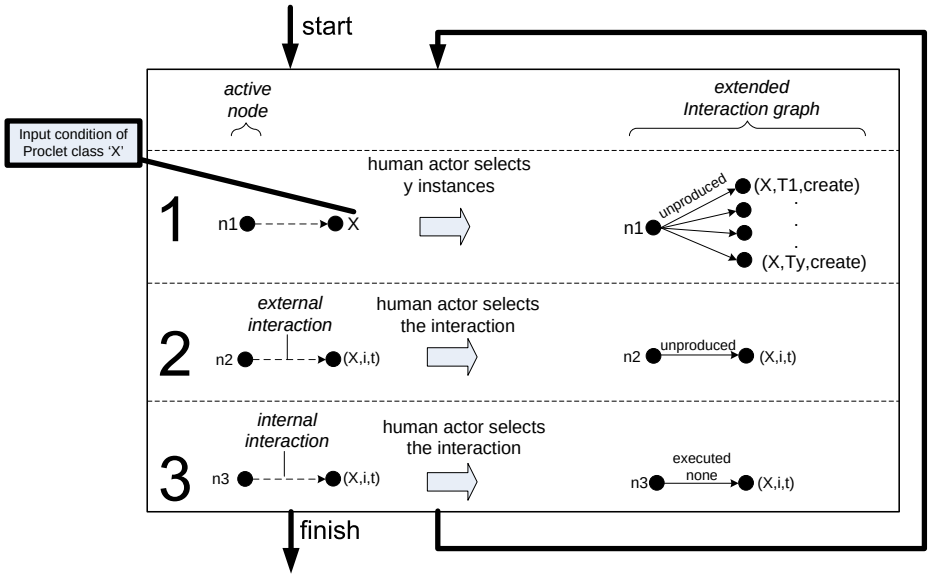


Fig. 26. Schematic representation of the procedure for extending an interaction graph for an entity. Each dot represents an interaction point.

instance identifier. Also, a node is active if it has a temporary instance identifier, i.e. the Proclet instance still needs to be created. Obviously, for active interaction nodes, interactions that are defined for it may potentially occur in the future.

After having determined all the active interaction nodes in the interaction graph, for each of them it is calculated in which next interactions the node can be involved, i.e. the candidate interactions. For calculating these candidate interactions for active nodes, three different situations can be distinguished which are indicated by the three numbers in Figure 26. For each different situation, we elaborate on the kind of interaction that is possible and how a nominated interaction leads to the extension of the interaction graph.

– The first situation relates to an external interaction. Via this external interaction it is possible to create instances of a given Proclet class. That is, for active node "n1" that is under consideration, the following is observed. Looking to the corresponding interaction point of node "n1" in its Proclet class, it can be seen that via an external interaction, it is connected with the input condition of Proclet class "X". As a result, multiple instances of Proclet class "X" may be created.
  For each instance that needs to created, an interaction arc is added leading from node "n1" to the node that is added for the new instance of Proclet class "X". Note that the newly added node has a temporary instance identifier which has "T" as prefix. Also, the added arc has interaction state "unproduced" and it has a unique interaction identifier.

– The second situation relates to an external interaction. Via this external interaction it is possible to have an interaction with a task instance of an existing or future Proclet instance. That is, for active node “n2” that is under consideration, the following is observed. Note that “n2” may relate to an existing or future Proclet instance. Looking to the corresponding interaction point of node “n2” in its Proclet class, it can be seen that via an external interaction, it is connected with task “T” of Proclet class “X”. Now, for Proclet class “X” an instance is existing which has instance identifier “i”. So, an interaction with task “T” of Proclet class “X” with instance identifier “i” is possible. Consequently, for node “n2” an interaction with node “(X,i,T)” may be nominated for extension in the graph.
  If selected, an interaction arc is added leading from node “n2” to the new node “(X,i,T)”. Also, the added arc has interaction state “unproduced” and it has a unique interaction identifier.
  Note that for Proclet class “X” also a future Proclet instance may be found in the interaction graph. That is, there is an interaction node referring to Proclet class “X” which has temporary instance identifier “Ti”. In that case, the nomination of the interaction and the subsequent extension of the graph is done in a similar way as for an existing Proclet instance.
– The third situation relates to an internal interaction. That is, for active node “n3” that is under consideration, the following is observed. Note that “n3” may relate to an existing or future Proclet instance. Looking to the corresponding interaction point of node “n3” in its Proclet class “X”, it can be seen that via an internal interaction, it is connected with task “T” of the same Proclet class. As we are dealing with an internal interaction, task “T” occurs in the same Proclet instance as where node “n3” refers to (say (temporary) instance identifier “i”). Consequently, for node “n3” an internal interaction with node “(X,i,T)” may be nominated for extension in the graph.
  If selected, an interaction arc is added leading from node “n3” to the new node “(X,i,T)”. Also, the added arc has interaction state “executed none” and it has a unique interaction identifier.

After that the graph is extended in the above described way, the procedure is repeated. That is, a new set of active nodes is calculated after which for each of them it is calculated in which interactions the node can be involved. In that way, a human actor can select new interactions or can indicate that he is finished.

Note that the interactions that are defined between interaction nodes in the graph might be limited by the cardinality and multiplicity values of the involved input and output ports. After that a human actor is done with defining interactions that need to take place, it is checked whether the new interactions are in line with the cardinality and multiplicity values of the ports. For that, the interaction graph of the entity itself needs to be considered but also the interaction graphs of other entities. If this is not ok, then the extended graph for the entity is rejected and the human actor has the option to define interactions again.

2.4 Performatives

Performatives are sent in between Proclet instances. For such a performative we have already indicated that it contains three different attributes. However, more attributes are relevant for an entity. Therefore, below we present the attributes, and their meaning, that are most relevant for a performative. Note that these attributes are based on the ones that are presented in [4, 5].

- *Time*: the moment the performative was created / received.
- *Channel*: the medium used to exchange the performative.
- *Sender*: the identifier of the Proclet instance creating the performative.
- *Set of receivers:* the identifiers of the Proclet instances receiving the performative, i.e. a list of recipients.
- *Action*: the type of the performative.
- *Content*: the actual information that is being exchanged.
- *Set of interaction identifiers*: a list of interaction identifiers. In particular, for the interaction arcs for which the performative is sent, the associated interaction identifier is added to this set.

The role of the action attribute deserves some special attention. This attribute can be used to specify the illocutionary point of the performative. The five illocutionary points identified by Searle [23] (assertive, directive, commissive, declarative, expressive) can be used to specify the intent of the performative. Examples of typed performatives identified by Winograd and Flores are request, offer, acknowledge, promise, decline, counter-offer, and commit-to-commit [27] which each represents a change in the state of a conversation. In the model no restriction is made to any single classification of performatives (i.e. a fixed set of types). It is important to use the experience and results reported by researchers working on the language/action perspective [27] as these give an insight into the broader requirements in this area. Of course, it is possible to add more attributes to a performative.

For entities, the "content" field of a performative can be used for exchanging data between Proclet instances in a structured way. However, first we need to remember that a performative may be multicasted to multiple receivers. That is, for different Proclet instances of the same Proclet clas, the performative has the same task as destination. This is illustrated in Figure 27 in the context of the second scenario. Here, we see for both "Anne" and "Sue" that a performative is sent from the "send reports" task of the "MDM" Proclet instance. For both of them, the performative has the "receive" task of the "visit" Proclet class as destination. However, for "Sue" the performative needs to be received by the Proclet instance which has "Sue 10/02" as instance identifier and for "Anne" the performative needs to be received by the Proclet instance which has "Anne 12/02" as instance identifier.

Now, we explain how the "content" attribute is used for exchanging data for entities in a structured way. As part of this, we require that for this attribute a fixed data structure is used. For this data structure, we may have a list of entity identifiers. On its turn, for each entity identifier we may have a list of name-value

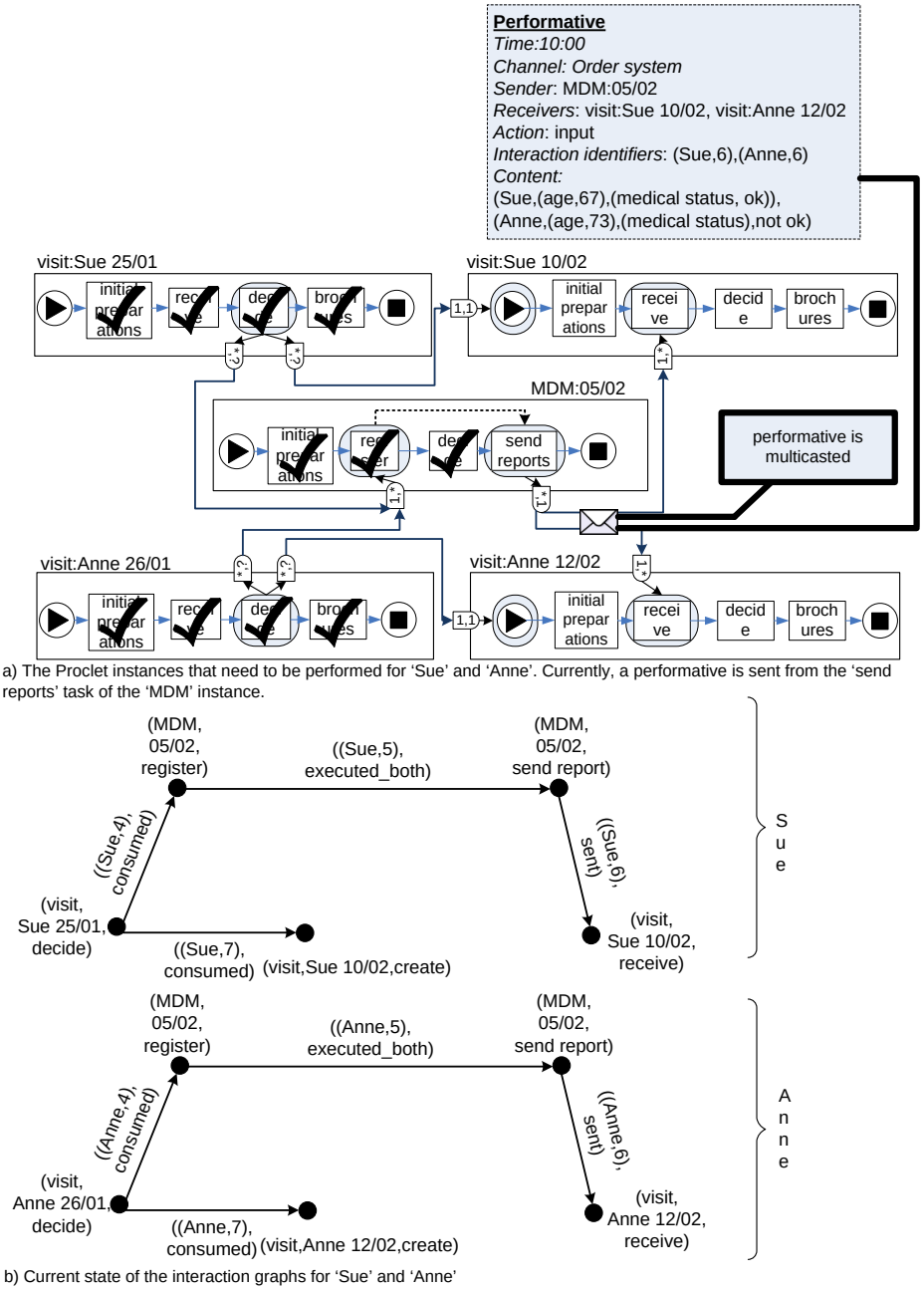


Fig. 27. Illustration of the attributes of a performative and its content.

pairs. This is illustrated in Figure 27 for the performative that is multicasted to the "visit" Proclet instances of "Sue" and "Anne". For the "content" attribute we see that there is an entity identifier element which has as identifier "Sue" and that has two name-value pairs. These name-value pairs indicate that Sue is 67 years old and that her medical status is ok. Similarly, for "Anne" we see that she is 73 years old and that her medical status is not ok.

Note that there is a close link between the information that is contained in the "content" attribute and the information contained in the "set of interaction identifiers" attribute. That is, for every interaction identifier in the "Interaction identifiers" field, a corresponding data element may be found in the "content" attribute which has the same entity identifier.

2.5 Formalization

In the previous sections, we have discussed all concepts and all aspects of the Proclets framework. An illustration of the resultant framework can be seen in Figure 28. In brief, there may be one or more Proclet classes. A Proclet class is a process definition for which instances may be created or destroyed. Via interaction points, ports, and channels, interactions between Proclet instances are possible. This occurs via performatives that are sent between these instances. Next, an interaction graph defines for an entity the interactions that need to take place between existing and future Proclet instances and their state. So, all interaction graphs together define the collaboration and communication between existing and future Proclet instances.

Having presented all concepts and all aspects of the Proclets framework, they now can be formalized. First, we formalize the syntax of both Proclet classes and the connections between them. Afterwards, we present a formal syntax for interaction graphs and a formal semantics for the enabling of task instances.

Connected Proclet Classes In this section, we formalize the syntax of both Proclet classes and how they are connected. Our formalizations are based on the YAWL workflow language [7]. Compared to other languages, YAWL is more expressive and has clear and unambiguous semantics. Therefore, this makes the YAWL language an excellent candidate for our extensions. Note that our extensions are in no way limited to the YAWL workflow language and can be applied to more complex notations (BPM, EPCs, BPEL, etc).

First, we provide some formalizations for YAWL itself. In particular, a YAWL net may be composed of a number of Extended Workflow Nets (EWF-nets). An EWF-net can be defined as follows:

Definition 1 (EWF-net). *An EWF-net is a tuple* $EWF = (C, \boldsymbol{i}, \boldsymbol{o}, T, F, split, join, rem, nofi)$ *such that:*

- C *is a set of* conditions*;*
- $\boldsymbol{i} \in C$ *is the input condition;*
- $\boldsymbol{o} \in C$ *is the output condition;*

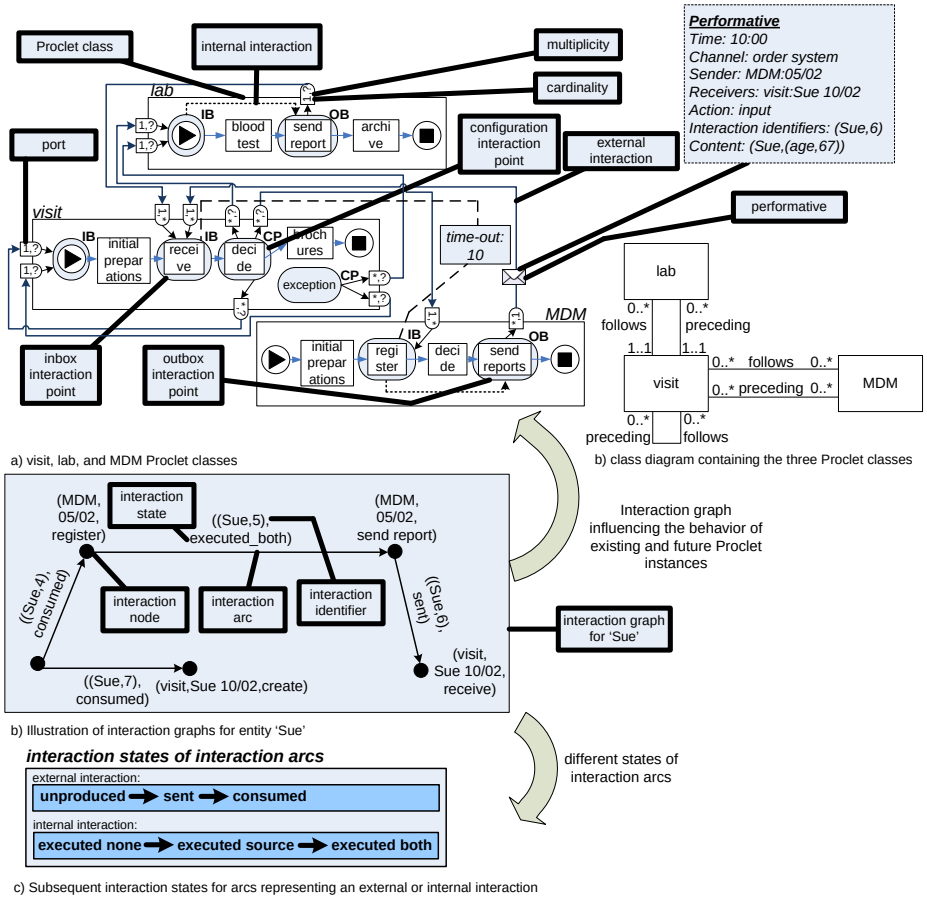


Fig. 28. Illustration of all concepts of the Proclets framework. Note that for interaction points, the abbreviations "CP", "IB", and "OB" represent a configuration, inbox, and outbox interaction point respectively.

- T *is a set of tasks;*
- $F \subseteq (C \backslash \{o\} \times T) \cup (T \times C \backslash \{i\}) \cup (T \times T)$
- *every node in the graph* $C \cup T, F)$ *is on a directed path from* i *to* o*.*
- $split \colon T \to \{AND, XOR, OR\}$ *specifies the split behavior of each task.*
- $join \colon T \to \{AND, XOR, OR\}$ *specifies the join behavior of each task.*
- $rem \colon PT \nrightarrow \mathbb{P}(T \cup C \backslash \{\boldsymbol{i}, \boldsymbol{o}\})$ *specifies the additional tokens to be removed by emptying a part of the workflow, and*
- $nofi \colon T \nrightarrow \mathbb{N} \times \mathbb{N}^{inf} \times \mathbb{N}^{inf} \times \{dynamic, static\}$ *specifies the multiplicity of each task.*

Note that the tuple (C, T, F) corresponds to a Petri net where C corresponds to places, T corresponds to transitions, and F is the flow relation. However, there are two important differences. First of all, an EWF-net has two special

places: **i** and **o**. Secondly, the flow relation also allows that tasks are directly connected. Semantically, the latter can be seen as an implicit place connecting two transitions.

Based on an EWF-net, a Proclet class can now be defined in the following way:

Definition 2 (Proclet class). *A Proclet class is a tuple* $N = (C, \boldsymbol{i}, \boldsymbol{o}, T, F, split, join, rem, nofi, \boldsymbol{PCid}, \boldsymbol{eip}, IP, IP_{type}, time\_out, P, card, mult, dir, PC, II)$*, where:*

- $(C, \boldsymbol{i}, \boldsymbol{o}, T, F, split, join, rem, nofi)$ *is an EWF-net;*
- ***PCid*** *is the unique identifier of the Proclet class;*
- ***eip*** *= is the identifier of the exception interaction point (no name clashes are assumed). If no exception interaction point is defined for the Proclet class, the value of* ***eip*** *is "null";*
- $IP \subseteq T \cup \{\boldsymbol{i}\} \cup \{\boldsymbol{eip}\}$ *is the set of interaction points. An interaction point may only be a task, input condition or the exception interaction point;*
- $IP_{type}\colon IP \rightarrow \{CONFIGURATION, INBOX, OUTBOX\}$ *is a function which maps interaction points on to a type;*
- $time\_out\colon IP \nrightarrow \mathbb{N}$ *is a partial function which maps a time out value (numeric value) to an interaction point. A time out value only may be defined for interaction points which are of the* $INBOX$ *type and are not linked to an input condition.*
- P *is a non-empty finite set of ports;*
- $card\colon P \rightarrow \{1, +, *, ?\}$ *is a function which maps a cardinality on to a port;*
- $mult\colon P \rightarrow \{1, +, *, ?\}$ *is a function which maps a multiplicity on to a port;*
- $dir\colon P \rightarrow \{IN, OUT\}$ *is a function which maps a direction on to a port. A port is either an input port or an output port;*
- $PC \subseteq \{(p,t) | p \in P \wedge t \in \{\boldsymbol{i}\} \wedge IP_{type}(t) = INBOX \wedge dir(p) = IN\} \cup$
  $\{(p,t) | p \in P \wedge t \in T \wedge IP_{type}(t) = INBOX \wedge dir(p) = IN\} \cup$
  $\{(t,p) | t \in T \wedge p \in P \wedge IP_{type}(t) = OUTBOX \wedge dir(p) = OUT)\} \cup$
  $\{(t,p) | t \in T \wedge p \in P \wedge IP_{type}(t) = CONFIGURATION \wedge dir(p) = OUT)\} \cup$
  $\{(t,p) | t \in \{\boldsymbol{eip}\} \wedge p \in P \wedge IP_{type}(t) = CONFIGURATION \wedge dir(p) = OUT\}\}$*. PC is the set of connections between ports and interaction points. There may only be connections leading from input ports to interaction points of type INBOX and there may only be connections leading from interaction points of type OUTBOX and CONFIGURATION to output ports. Moreover, the input condition is an interaction point of type INBOX and the exception interaction point is an interaction point of type CONFIGURATION.*
- $(\forall_{(x_1,y_1) \in PC \wedge x_1 \in P}\colon (\forall_{(x_2,y_2) \in PC \wedge x_2 \in P \wedge (x_1,y_1) \neq (x_2,y_2)}\colon x_1 \neq x_2))$ *and* $(\forall_{(x_1,y_1) \in PC \wedge y_1 \in P}\colon (\forall_{(x_2,y_2) \in PC \wedge y_2 \in P \wedge (x_1,y_1) \neq (x_2,y_2)}\colon y_1 \neq y_2))$*. Every port is connected to one and only one interaction point.*
- $II \subseteq \{(a,b) | a \in IP \wedge b \in IP \wedge IPtype(a) = INBOX \wedge IPtype(b) = OUTBOX\}$ *is the set of internal interaction arcs. An internal interaction arc may only be directed from an interaction point of type* $INBOX$ *to an interaction point of type* $OUTBOX$*.*

– $(\forall_{(x_1,y_1)\in II}\colon (\forall_{(x_2,y_2)\in II\wedge(x_1,y_1)\neq(x_2,y_2)}\colon x_1 \neq x_2))$ *and*
$(\forall_{(x_1,y_1)\in II}\colon (\forall_{(x_2,y_2)\in II\wedge(x_1,y_1)\neq(x_2,y_2)}\colon y_1 \neq y_2))$. *An interaction point is only the head or tail of one and only one internal interaction arc.*

Proclet classes are connected with each other via ports in order to allow for interactions. Therefore, we define a set of connected Proclet classes as follows:

Definition 3 (set of connected Proclet classes).

A set of connected Proclet classes is a tuple $S = (Q, P^{\diamond}, PCQ, CH, channel)$ where:

– Q is a set of Proclet classes;
– $P^{\diamond} = \bigcup_{N\in Q} P_N$ is the set of all ports;
– $PCQ \subseteq \{(a,b)|N_1 \in Q \wedge N_2 \in Q \wedge a \in P_{N_1} \wedge b \in P_{N_2} \wedge dir_{N_1}(a) = OUT \wedge dir_{N_2}(b) = IN\}$ is the set of all output ports that are connected with input ports;
– $(\forall_{(x_1,y_1)\in PCQ}\colon (\forall_{(x_2,y_2)\in PCQ\wedge(x_1,y_1)\neq(x_2,y_2)}\colon x_1 \neq x_2))$ and
$(\forall_{(x_1,y_1)\in PCQ}\colon (\forall_{(x_2,y_2)\in PCQ\wedge(x_1,y_1)\neq(x_2,y_2)}\colon y_1 \neq y_2))$. Every port is only connected to one and only one other port.
– CH is a non-empty finite set of channels;
– $channel\colon PCQ \to CH$ is a function which maps two connected ports on to a channel.

Note that Figure 28a fully defines a set of connected Proclet classes.

Interaction Graphs In previous sections, we have discussed in a informal way the semantics of the interactions that occur between and inside Proclet instances for entities. To this end, we have introduced the notion of an interaction graph for entities. This also allowed us to define the enabling of task instances. Below, we will provide a formal syntax for interaction graphs and a formal semantics for the enabling of task instances. We will do this in two steps. First, we provide some necessary preliminaries in order to be able to define a valid interaction graph. Second, we formalize an interaction graph itself. Finally, we formalize the enabling of task instances.

In order to be able to define interaction graphs, we need to be able to refer to Proclet class identifiers, all interaction points within Proclet classes, and (temporary) Proclet instance identifiers. Moreover, for the definition of interaction arcs, we need to be able to refer to all internal and external interactions that exist within and in between all Proclet classes. Therefore, we start with introducing the following formal notions.

Definition 4 (Preliminaries Interaction Graph).

Let $S = (Q, P^{\diamond}, PCQ, CH, channel)$ be a set of connected Proclet classes. The preliminary definitions for an interaction graph are the following.

– $PCid^{\diamond} = \{\mathbf{PCid}_N\}$ is the set of all Proclet class identifiers;
– I is a finite set of Proclet instance identifiers;

- I_{temp} is a finite set of temporary Proclet instance identifiers;
- $I \cap I_{temp} = \varnothing$, both I and I_{temp} contain unique identifiers;
- $II^{\diamond} = \{II_N | N \in Q\}$ is the set of all internal interaction arcs (no name clashes assumed);
- $PII\_ARCS = \{((\mathbf{PCid}_N, a), (\mathbf{PCid}_N, b)) | N \in Q \wedge (a, b) \in II_N\}$ is the set of all internal interaction arcs for all Proclet classes;
- an *external interaction* path p from an interaction point in net N_1 to an interaction point in net N_2 is a path $p = \langle n_1, n_2, n_3, n_4 \rangle$ such that $(n_1, n_2) \in PC_{N_1} \wedge (n_2, n_3) \in PCQ \wedge (n_3, n_4) \in PC_{N_2} \wedge N_1 \in Q \wedge N_2 \in Q$;
- $PEI\_ARCS = \{((\mathbf{PCid}_{N_1}, n_1), (\mathbf{PCid}_{N_2}, n_4)) | N_1, N_2 \in Q \wedge n_1 \in IP_{N_1} \wedge n_2 \in IP_{N_2} \wedge n_1$ is the first node and n_4 is the last node on an *external interaction path* $p\}$ is the set of all external interaction arcs between all Proclet classes;

An interaction node refers to a concrete interaction point of a Proclet instance. This means that we need to be able to make references to Proclet class identifiers, (temporary) identifiers of Proclet instances, and interaction points within Proclet classes. Therefore, $PCid^{\diamond}$ is the set of all Proclet class identifiers. For referring to identifiers of existing Proclet classes, we have I which provides a set of Proclet class identifiers. Directly related to this, I_{temp} provides a set of temporary identifiers for future Proclet instances. Together I and I_{temp} contain unique identifiers.

For interaction arcs, it is necessary to refer to all interaction points within Proclet classes and all internal and external interactions that exist in between these interaction points. Therefore, we have the set $II^{\diamond}$ which contains all the internal interactions that exist within all Proclet classes. For referring to internal interactions within Proclet classes, we need to link these interactions with the identifier of the Proclet class. Therefore, for both the source and destination interaction point of the internal interaction, we attach the identifier of the respective Proclet class. In that way, the set PII_ARCS contains all the possible internal interaction arcs that exist within all Proclet classes. For external interactions, we already remarked that in that case the direction of the arc between the two interaction nodes is the same as the direction of the corresponding external interaction of the associated Proclet classes. So, we need to have a path from the source interaction node a of the external interaction to the destination interaction point b of the external interaction. Such a path starts at source interaction point a, leads subsequently via an output port and an input port (which is contained in set PCQ), to the destination interaction point b. We call such a path an *external interaction path*. However, in the interaction graph there is only a direct connection from interaction point a to interaction point b ignoring the ports that are in between. This is defined by the set PEI_ARCS which contains all the possible external interactions that exist between two Proclet classes. Note, that in a similar fashion as for PII_ARCS, also the source and destination interaction points of external interactions are linked with the identifier of the respective Proclet classes.

By having defined above presented notions, we are now able to define an interaction graph in the following way.

Definition 5 (Interaction Graph).

Let $S = (Q, P^{\diamond}, PCQ, CH, channel)$ be a set of connected Proclet classes. An interaction graph for an entity is a tuple $IG = (\textbf{entity\_id}, IN, IA, ID, arcState, iid\_arc)$ where:

- **entity_id** is an unique entity identifier (no name clashes are assumed with entity id's of other interaction graphs);
- $IN \subseteq \{(c, i, p) | c = \textbf{PCid}_N \wedge i \in (I \cup I_{temp}) \wedge p \in IP_N \wedge N \in Q\}$ is the set of interaction nodes in an interaction graph;
- $IA \subseteq \{((c_1, i_1, p_1), (c_2, i_2, p_2)) | (c_1, i_1, p_1) \in IN \wedge (c_2, i_2, p_2) \in IN \wedge ((c_1, p_1), (c_2, p_2)) \in (PEI\_ARCS \cup PII\_ARCS)\}$ is the set of interaction arcs in an interaction graph of an entity;
- $(\forall_{n \in IN} : (\exists_{(n_1,n_2) \in IA} : n = n_1) \vee (\exists_{(n_1,n_2) \in IA} : n = n_2))$, each interaction node has at least an incoming arc or an outgoing arc;
- $arcState : IA \rightarrow \{UNPRODUCED, CONSUMED, SENT, EXECUTED\_NONE, EXECUTED\_SOURCE, EXECUTED\_BOTH, FAILED\}$ is a function which maps interaction arcs on to an arc state.
  For $(c, i, p) \in IA$ and $(c, p) \in PII\_ARCS$, we have $arcState((c, i, p)) = \{EXECUTED\_NONE, EXECUTED\_SOURCE, EXECUTED\_BOTH, FAILED\}$, i.e. an interaction arc referring to an internal interaction may only have state "EXECUTED_NONE", "EXECUTED_SOURCE", and "EXECUTED_BOTH".
  For $(c, i, p) \in IA$ and $(c, p) \in PEI\_ARCS$, we have $arcState((c, i, p)) = \{UNPRODUCED, CONSUMED, SENT, FAILED\}$, i.e. an interaction arc referring to an external interaction may only have state "UNPRODUCED", "CONSUMED", "SENT", and "FAILED".;
- $iid\_arc : IA \rightarrow \{\textbf{entity\_id}\} \times ID$ is a function which maps each arc onto a unique interaction identifier. Such an interaction identifier consists of a pair of which the first value is the **entity_id** and of which the second value is an unique identifier. This implies $(\forall_{(a_1,b_1) \in rng(iid\_arc)} : (\forall_{(a_2,b_2) \in rng(iid\_arc) \wedge (a_1,b_1) \neq (a_2,b_2)} : b_1 \neq b_2))$;

An interaction graph corresponds to a specific entity. In the interaction graph we have interaction nodes which are contained in the set IN. Interaction nodes are connected via interaction arcs (IA). An interaction arc is either an internal or external interaction. In order for an interaction node to be included in the graph, it must have at least one incoming or at least one outgoing interaction arc. Furthermore, for interaction arcs, the specific state of the interaction is kept by the *arcState* function and a unique interaction identifier for the interaction is kept by the *iid_arc* function. Note that Figure 28b fully defines an interaction graph for the entity "Sue".

Furthermore, the enabling of task instances related to inbox interaction points can be formalized in the following way.

Definition 6 (Enabling of task instances).

Let $S = (Q, P^{\diamond}, PCQ, CH, channel)$ be a set of connected Proclet classes.

1. For Proclet class c ($c = \mathbf{PCid}_N, N \in Q$) having an instance with identifier i ($i \in I$), the task t ($t \in T_N$) is enabled if the following holds for the associated INBOX interaction point (c, i, t), $t \in IP_N, IP_{type\,N}(t) = INBOX$:
   - the task t in Proclet instance i is *enabled*;
   - for all interaction graphs in which interaction node (c, i, t) occurs, the following needs to hold: $(\forall_{((c_1,i_1,t_1),(c,i,t))\in IA}:$ $arcState((c_1, i_1, t_1), (c, i, t)) = SENT \vee$ $arcState((c_1, i_1, t_1), (c, i, t)) = FAILED)$, i.e. all incoming interaction arcs (representing an external interaction) for node (c, i, t) in all interaction graphs need to have either state $SENT$ or $FAILED$.
2. For Proclet class c ($c = \mathbf{PCid}_N, N \in Q$) having an instance with identifier i ($i \in I$), the task t ($t \in T_N$) is enabled if the following holds for the associated OUTBOX interaction point (c, i, t), $t \in IP_N, IP_{type\,N}(t) = OUTBOX$:
   - the task t in Proclet instance i is *enabled*;
   - for all interaction graphs in which interaction node (c, i, t) occurs, the following needs to hold: $(\forall_{((c_1,i_1,t_1),(c,i,t))\in IA}:$ $arcState((c_1, i_1, t_1), (c, i, t)) = EXECUTED\_SOURCE)$, i.e. all incoming interaction arcs (representing an internal interaction) for node (c, i, t) in all interaction graphs need to have state $EXECUTED\_SOURCE$.
3. For Proclet class c ($c = \mathbf{PCid}_N, N \in Q$) having an instance with identifier i ($i \in I$), the task t ($t \in T_N$) is enabled if the following holds for the associated CONFIGURATION interaction point (c, i, t), $t \in IP_N, IP_{type\,N}(t) = CONFIGURATION$:
   - the task t in Proclet instance i is *enabled*;

Furthermore, for an input condition of a Proclet class also an *INBOX* interaction point can be defined with multiple input ports. In that way, an instance of the respective Proclet class is created if the following holds for the interaction point (c, i, p), $c = \mathbf{PCid}_N, i \in I_{temp}, p \in IP_N, N \in Q$:

- for all interaction graphs, there is only one occurrence of interaction node (c, i, p) which has only one incoming arc a. For arc a, the following needs to hold: $arcState(a) = SENT$

3 Architecture

The Proclet framework which has been discussed in the previous section forms the conceptual foundations for augmenting existing WfMSs with inter-workflow support. The architecture of such an augmented WfMS is presented in Figure 29 and consists of three components. The Workflow Engine and the Workflow Client Application components provide the basic functionality that may be expected from any WfMS. Next, the Inter-Workflow Service component is responsible for adding inter-workflow support. The interface between the WfMS and the service has been indicated by a cloud with number "1" in it.

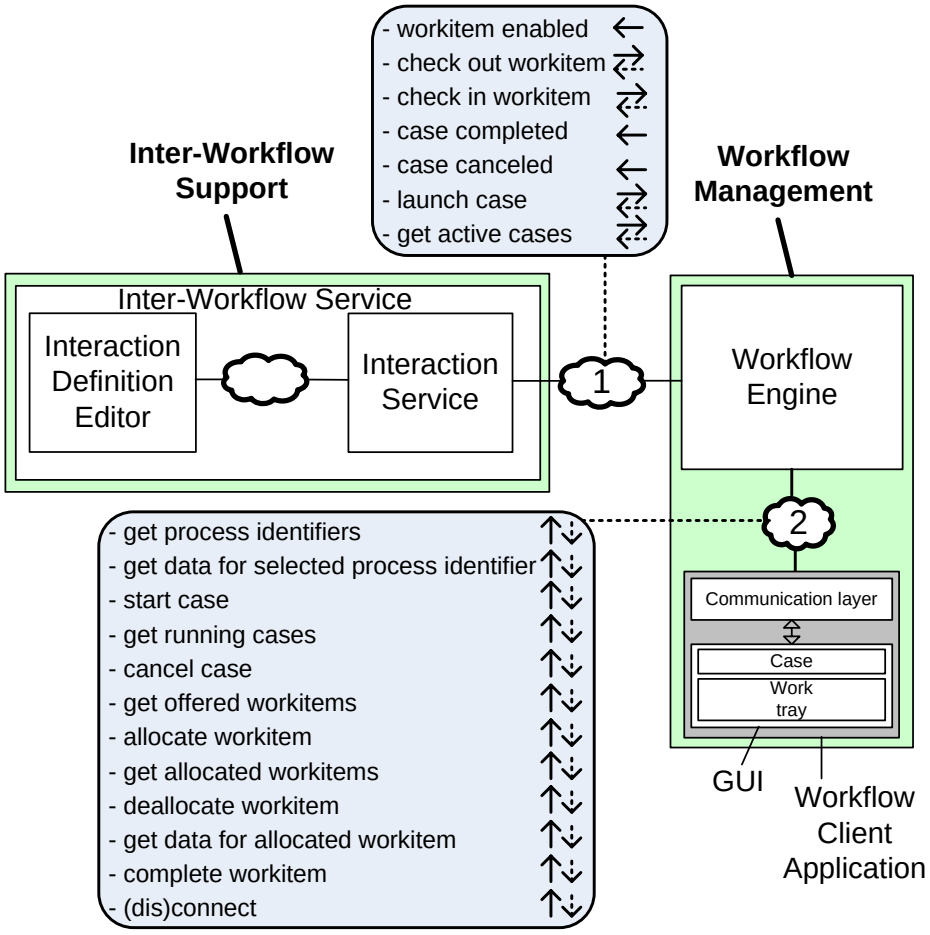


Fig. 29. Architecture of a WfMS augmented with inter-workflow support.

In sections 3.1 to 3.3 the individual components in the architecture are discussed in more detail. For each component a description of the main functionality is provided together with a discussion on its interactions with other components.

3.1 Workflow Engine

The workflow engine is the heart of the system and provides those facilities which are required for the logistical completion of cases. The basic facilities that are important to us regard the creation or deletion of cases and ensuring that workitems for tasks are carried out in the right order and by the right people. Moreover, data concerning cases and workitems is managed.

3.2 Workflow Client Application

Users working with the WfMS do so via the Workflow Client Application which delivers the basic user interaction facilities expected of this facility. The component consists of a GUI and a communication layer component. The GUI is responsible for the interactions with the users. The communication layer component serves as a connection layer between he engine and the GUI and takes care of the communication between them.

Subcomponents The GUI component consists of three different subcomponents. Here, the "worktray" provides the same facilities as a classical worktray. That is, workitems for tasks are advertised, allocated, and performed via the worktray. Furthermore, only one user can interact with the WfMS to indicate the completion of a workitem. This prevents concurrency issues should multiple users want to complete the same workitem. Finally, the "Case" component provides facilities regarding the creation and deletion of cases.

Communication As can be seen in Figure 29, one interface is defined for the communication between the Workflow Client Application and Workflow Engine. The interface with number "2" defines the communication that needs to take place between an engine and a Workflow Client Application in order to satisfy the basic facilities that are to be provided by an engine.

3.3 Inter-Workflow Service

The Inter-Workflow Service comprises of the Interaction Service component and the Interaction Definition Editor component. These and additional aspects are discussed in more detail below.

Interaction Service The *Interaction Service* component is responsible for storing and taking care of the interactions that take place between Proclet instances. More specifically, for tasks for which interactions are necessary, a corresponding interaction point is defined at the service side which means that the execution of these tasks is deployed to the Inter-Workflow Service. In that way, for such a task instance the service identifies which interactions are necessary, i.e. whether the sending and receiving of performatives is necessary. If yes, then these interactions are taken care of which also may involve the instantiation of Proclet classes. Next to that, the Interaction Service takes care of identifying whether exceptions occur (e.g. the cancelation or completion of a Proclet instance). Based on the decision of a human actor, an exception is handled. Finally, based on already defined interactions for an entity, subsequent interactions that are possible are determined in case the opportunity is offered to extend an interaction graph. As part of this, it is needed to have an overview of the cases that are existing in the WfMS.

Interaction Definition Editor A human actor will only have contact with the Inter-Workflow Service via the *Interaction Definition Editor*. In that way, the component offers the ability to define interactions for an entity, i.e. extending the corresponding interaction graph, both in normal and exceptional situations. Here it should be noted that possible interactions for an entity, which are determined by the Interaction Service, are offered to a human actor via the editor. From these possible interactions, a selection is made and send to the Interaction Service such that new possible interactions are calculated and offered again. Next to that, identified exceptions are presented such that a human actor can decide about how they need to be handled (e.g. take no action or extend the interaction graph for an entity).

Inter-Workflow Related Extensions Moreover, note that in Section 2.5, we extended an EWF-net with concepts for providing inter-workflow support (e.g. interaction points, ports, connections between ports). However, these extensions can be applied to any workflow language. For the architecture presented in Figure 29, this means that ordinary process definitions can be stored at the engine side (e.g. EWF-net definitions presented in Definition 1) and that the inter-workflow related extensions can be stored at the Inter-Workflow side. That is, the extensions which are presented by definitions 2 and 3 (e.g. interaction points, time out values, connections between ports) are stored at the Inter-Workflow support side. In this way, by making this separation we can truly add Inter-Workflow support to any WfMS. Logically, the interaction graphs are also stored at the Inter-Workflow Service side.

Communication with the Engine In order to enable the Inter-Workflow Service to serve a workflow engine, an interface between them needs to be defined. The interface consists of a number of events and methods and has been kept to an absolute minimum in order to ease the number of workflow engines that benefit from the capabilities offered by the Inter-Workflow Service. Additionally, being a service, this allows a single instance of the service to take care of the interactions between Proclet instances and the definition of them by a user. As can be seen in Figure 29, in the rectangle that is linked to the cloud with number 1, an overview of these methods and events is provided. The first three methods relate to the outsourcing of the performance of the workitem to the Inter-Workflow Service. The two next methods relate to the completion and cancelation of a case for which it needs to be determined if exception handling is needed. The last two methods relate to the launching of a case in the engine for a Proclet instance and obtaining an overview of the cases that are running in the engine.

Finally, note that the Interaction Service and the Interaction Definition Editor are tightly coupled as large volumes of information are exchanged.

4 Related Work

In the introduction we have indicated that the entire patient process of elective non-routine healthcare processes can be seen as a cloud of standardized process fragments for which the ultimate selection of the fragments and the interactions between them are patient specific. In order to support these loosely coupled workflow fragments and the interactions between them, inter-workflow support is required.

Contemporary workflow languages and systems provide limited support for the modeling and execution of loosely coupled interacting workflows. Instead, one is forced to squeeze real-life processes into a single "monolithic overarching workflow" which describes how an individual case is handled in isolation. When doing this, the modeler looses the overview, the natural structure of work is lost, and the required flexibility cannot be offered to the medical professionals involved. This issue has been recognized in the literature [4, 5, 25, 10, 18, 20] and is not limited to the healthcare domain. It also applies in other areas, e.g. the automotive domain [20], or when reviewing papers for a conference [5]. As a consequence of assuming that workflows execute completely in isolation, current WfMSs do not provide an adequate means for inter-workflow coordination [25, 17, 16].

One of the earliest formalisms which acknowledged the above mentioned problems is the *Proclets* framework [4, 5, 25] proposed by van der Aalst et al. In comparison to the Proclets framework there are a limited range of alternate approaches that deal with the same issues [18] as those dealt with by the Proclets framework. Müller et al. have worked on the Corepro framework [19, 20] which allows automatic generation and coordination of individual processes, operating at different levels of aggregation, based on their underlying data structure. For a specific component in the data structure, the corresponding structure of the process is described by the data used during the life cycle of the component. Relationships between the components in the data structure, which can capture one-to-many and many-to-many relationships, indicate process dependencies. Based on the data structure, the initial number of process instances created is decided at run-time. The creation of new instances at runtime is possible, but requires an ad-hoc change to the related data structure. In this context, it is also important to mention that several WfMSs, such as FLOWer (via a dynamic subplan) and YAWL (via a multiple instance task), offer support for creating multiple concurrent instances of a task. They support a one-to-many relationship between the task and its instances. However, many-to-many relationships can not be captured.

Browne et al. specifically focus on the healthcare domain and present a two-tier, goal-driven model for workflow processes in the healthcare domain [11–13]. A goal-ontology, presented as a directed acyclic graph, is utilized to represent the business model at the upper level and is decomposed into an extended Petri-net model for the lower level workflow schema. A mapping is defined from the goal-graph to (sub)processes and tasks such that each of the (sub)processes is designed in a way that achieves one of the upper level goals. This approach

leads to a hierarchy of process models with a number of the top-level goals being implemented through subprocesses. However, there is no interaction between subprocesses in contrast to the classical way in which hierarchical processes communicate with each other in a top-down fashion.

Bhattacharya et al. take the so-called business artifacts of a process as a starting point [9, 10, 21]. A business artifact is an identifiable, self-describing unit of information. First it needs to be identified which business artifacts need to be processed in order to achieve a certain business goal. Based on this an artifact-centric process model can be constructed which consists of the business artifacts itself, business tasks, and repositories. A repository describes a buffer for an artifact. Tasks can push an artifact into a repository or pull it out of a repository. In this way, the content of one or more artifacts can only be changed by the execution of a task. Consequently, aggregation issues are only handled at the task level instead of at the process level.

Batch-oriented tasks are tasks that are based on groupings of lower aggregation elements. The concept of a batch-oriented task was introduced in [8] in order to allow for a task that is executed for multiple instances at the same time. In [22], the problem is defined and deliberations are provided on the technology support required to deal with the issue.

Finally, Heinlein focuses on the synchronization of concurrent workflows [17, 16, 15]. An interaction graph is proposed which specifies how multiple workflows need to be synchronized. In such an interaction graph, tasks, (user-defined) operators, and parameters may be used for defining in a general way how tasks between multiple workflows need to be synchronized, i.e. the sets of permissible execution sequences are specified. At run-time, the graph is transformed, via interaction expressions, into an operational model consisting of states, state transitions, and state predicates. In this way, the defined task synchronizations can be enforced such that tasks that are allowed to be executed, but that are not permissable according to the interaction graph, are disabled. Finally, all involved workflows operate at the same level of aggregation and no data is exchanged between workflows.

5 Conclusions

In this chapter, we have presented the design of a WfMS augmented with inter-workflow support facilities. These facilities are based on the Proclets Framework. An important concept in this framework are interaction points. Via these interaction points, at design time, possible interactions between Proclet classes can be modeled without the need to define complex pre- and postconditions. Subsequently, at run-time they allow users to select interactions between Proclet instances.

In case a WfMS augmented with inter-workflow support would be used in a healthcare environment for the daily management of patient flows several benefits can be obtained. First of all, workflows exist which deal with a single patient but also workflows exist dealing with a group of patients. Using current WfMSs

these workflows need to be described in separate, disconnected, models. By using our system these workflows are still described in their own models but connections between them are possible (e.g. registering a patient for a multidisciplinary meeting). So, the mix of granularities that co-exist in workflows can be taken into account. Moreover, interactions between them can be captured.

Directly related to the above mentioned issue is that one-to-many or many-to-many relationships may exist between entities in a workflow. Using contemporary workflow languages these relationships are typically not expressed. In the Proclet framework, these relationships are explicitly captured.

Also, for healthcare processes in general, a doctor proceeds in a step-by-step way in which a series of subprocesses are started. However, it can not be guaranteed that these finish in time before the start of the next consultation with a doctor. By using our system, these processes are supported by modeling them as a series of fragments running at different speeds, but that are still connected in some way. That is, a process can be considered as a series of intertwined loosely-coupled workflow fragments.

References

1. W.M.P. van der Aalst. Verification of Workflow Nets. In P. Azéma and G. Balbo, editors, *Application and Theory of Petri Nets 1997*, volume 1248 of *Lecture Notes in Computer Science*, pages 407–426. Springer-Verlag, Berlin, 1997.
2. W.M.P. van der Aalst. Formalization and Verification of Event-driven Process Chains. *Information and Software Technology*, 41(10):639–650, 1999.
3. W.M.P. van der Aalst. Business Process Management Demystified: A Tutorial on Models, Systems and Standards for Workflow Management. In J. Desel, W. Reisig, and G. Rozenberg, editors, *Lectures on Concurrency and Petri Nets*, volume 3098 of *Lecture Notes in Computer Science*, pages 1–65. Springer-Verlag, Berlin, 2004.
4. W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Workflow Modeling using Proclets. In O. Etzion and P. Scheuermann, editors, *7th International Conference on Cooperative Information Systems (CoopIS 2000)*, volume 1901 of *Lecture Notes in Computer Science*, pages 198–209. Springer-Verlag, Berlin, 2000.
5. W.M.P. van der Aalst, P. Barthelmess, C.A. Ellis, and J. Wainer. Proclets: A Framework for Lightweight Interacting Workflow Processes. *International Journal of Cooperative Information Systems*, 10(4):443–482, 2001.
6. W.M.P. van der Aalst and K.M. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, Cambridge, MA, 2002.
7. W.M.P. van der Aalst and A.H.M. ter Hofstede. YAWL: Yet Another Workflow Language. *Information Systems*, 30(4):245–275, 2005.
8. P. Barthelmess and J. Wainer. Workflow Systems: a Few Definitions and a Few Suggestions. In N. Comstock and C.A. Ellis, editors, *Proceedings of the Conference on Organizational Computing Systems - COOCS'95*, pages 138–147, Milpitas, California, September 1995. ACM Press.
9. K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F.Y. Wu. Artifact-Centered Operational Modeling: Lessons from Customer EFngagements. *IBM Systems Journal*, 46(4):703–721, 2007.
10. K. Bhattacharya, C. Gerede, R. Hull, R. Liu, and J. Su. Towards Formal Analysis of Artifact-Centric Business Process Models. In G. Alonso, P. Dadam, and

M. Rosemann, editors, *International Conference on Business Process Management (BPM 2007)*, volume 4714 of *Lecture Notes in Computer Science*, pages 288–304. Springer-Verlag, Berlin, 2007.
11. E.D. Browne, M. Schrefl, and J.R. Warren. A Two Tier, Goal-Driven Workflow Model for the Healthcare Domain. In *Proceedings of the 5th International Conference on Enterprise Information Systems (ICEIS 2003)*, pages 32–39, 2003.
12. E.D. Browne, M. Schrefl, and J.R. Warren. Activity Crediting in Distributed Workflow Environments. In *Proceedings of the 6th International Conference on Enterprise Information Systems (ICEIS 2004)*, 2004.
13. E.D. Browne, M. Schrefl, and J.R. Warren. Goal-Focused Self-Modifying Workflow in the Healthcare Domain. In *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS-37 2004) - Track 6*. IEEE Computer Society Press, 2004.
14. M. Dumas, W.M.P. van der Aalst, and A.H.M. ter Hofstede. *Process-Aware Information Systems: Bridging People and Software through Process Technology*. Wiley & Sons, 2005.
15. C. Heinlein. *Workflow and Process Synchronization with Interaction Expressions and Graphs*. PhD thesis, Fakultät für Informatik, Universität Ulm, 2000. in German.
16. C. Heinlein. Workflow and Process Synchronization with Interaction Expressions and Graphs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 243–252. IEEE Computer Society, 2001.
17. C. Heinlein. Synchronization of Concurrent Workflows Using Interaction Expressions and Coordination Protocols. In R. Meersman and Z. Tari, editors, *On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, and ODBASE*, volume 2519 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, Berlin, 2002.
18. V. Künzle and M. Reichert. Towards Object-Aware Process Management Systems: Issues, Challenges, Benefits. In T. Halpin, J. Krogstie, S. Nurcan, E. Proper, R. Schmidt, P. Soffer, and R. Ukor, editors, *Proc. 10th Int'l Workshop on Business Process Modeling, Development, and Support (BPMDS'09)*, volume 29 of *Lecture Notes in Business Information Processing*, pages 197–210. Springer-Verlag, Berlin, 2009.
19. D. Müller, M. Reichert, and J. Herbst. Data-Driven Modeling and Coordination of Large Process Structures. In Z. Bellahsène and M. Léonard, editors, *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, volume 4803 of *Lecture Notes in Computer Science*, pages 131–149. Springer-Verlag, Berlin, 2007.
20. D. Müller, M. Reichert, and J. Herbst. A New Paradigm for the Enactment and Dynamic Adaptation of Data-Driven Process Structures. In R. Meersman and Z. Tari, editors, *Advanced Information Systems Engineering*, volume 5074 of *Lecture Notes in Computer Science*, pages 48–63. Springer-Verlag, Berlin, 2008.
21. A. Nigam and N.S. Caswell. Business Artifacts: An Approach to Operational Specification. *IBM Systems Journal*, 42(3):428–445, 2003.
22. S. Sadiq, M. Orlowska, W. Sadiq, and K. Schulz. When Workflows Will Not Deliver: The Case of Contradicting Work Practice. In W. Abramowicz, editor, *Proceedings BIS'05*, 2005.
23. J.R. Searle. *Speech Acts*. Cambridge University Press, Cambridge, 1969.
24. A.H.M. ter Hofstede, W.M.P. van der Aalst, M. Adams, and N. Russell, editors. *Modern Business Process Automation: YAWL and its Support Environment*. Springer-Verlag, 2010.

25. W.M.P. van der Aalst, R.S. Mans, and N.C. Russell. Workflow Support using Proclets: Divide, Interact and Conquer. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering*, 32(3):16–22, 2009.
26. M. Weske. *Business Process Management: Concepts, Languages, Architectures*. Springer-Verlag, Berlin, 2007.
27. T. Winograd and F. Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Ablex, Norwood, 1986.