Software Integration in Health Care

Emma Nilsson-Nyman *
Department of Computer Science
Lund University
June 2007

Abstract

We have studied software integration problems at the University Hospital in Lund through two scenarios which reflect their current workflow. Prototypes for these scenarios have been implemented using techniques developed within PalCom, an on-going research project in ubiquitous computing, aiming at developing techniques which make it easier to combine devices such as GPS devices and digital cameras. We have evaluated how well these techniques perform in a hospital environment, containing devices such as record-keeping systems and medical equipment. This kind of environment, which differs from environments where PalCom techniques have been applied so far, puts new demands on expressiveness and performance. We suggest language improvements and runtime system changes necessary to meet these demands.

1 Introduction

Health care today depend on a multitude of different electronic devices. Hospitals contain everything from pure software such as record-keeping systems to medical equipment like X-ray machines.

These devices need to work together in different combinations. Combining systems of this complexity is not easy. Most of these devices are highly specialized and rarely share protocols with other devices, i.e., speak any common language. These integration problems commonly lead to customized solutions between pairs of systems. A large number of different customized solutions create an overall system which is hard to overview.

Integration is a problem in many areas. We use a framework developed within PalCom [PP07], an on-going research project founded by the European Union, to combine devices. This framework was developed with the aim of supporting ubiquitous computing, i.e., computing which aims to handle a future where computers are everywhere and as natural a part of our everyday life as books or road signs [Wei91]. In particular small pervasive devices such as GPS devices and digital cameras, which easily can be moved between local networks, have been studied so far within PalCom.

We study the software integration problems at the hospital through the lens of two scenarios reflecting workflow examples at the hospital. We implement these scenarios using the PalCom framework. One purpose of using this framework in this environment is to evaluate how well it can handle it. The devices found at the hospital differ, somewhat, from the typical devices the framework originally was created to handle. Based on our observations we suggest changes.

The PalCom framework uses scripts called assemblies to express workflows between devices. We compare the assembly language to Petri nets [Pet62] and Statecharts [Har87] and discuss expressiveness using Workflow Patterns [vdAHKB03]. Based on these comparisons and discussions we suggest a graphical notation combining Petri nets with Statecharts to describe scenarios. We also suggest corresponding changes to the assembly language which connects the language to the suggested graphical notation.

This report starts with an introduction to the software integration problems at the hospital in Section 2 which is followed by an introduction to the PalCom system in Section 3. These sections are followed by a description of our suggested graphical notation in Section 4. We continue with an account of our scenario implementations in Section 5 using
the current PalCom framework and follow up with a description of our scenarios using the expressiveness of our suggested notation in Section 5. Before finishing we discuss possible extensions to the framework in Section 7, and then finally we cover related work in Section 8 and give a conclusion in Section 9.

2 Health Care Today

Health care today depends on a multitude of different devices and systems. Everything from pure software systems such as record-keeping systems to hardware such as X-ray machines can be found within the walls of a hospital.

The University Hospital in Lund actively uses over ten different record-keeping systems. Systems which have been chosen over time to serve different needs. For example, the needs of the emergency ward are different from the needs of one of the long-term medical care wards.

Along with these record-keeping systems the hospital contains a number of different types of more advanced medical equipment which provide their own software system.

All these systems need to work together in different combinations. Information from the X-ray system or from the chemical lab needs to be accessed together with information from the record-keeping systems. For example, a simple operation, such as moving a patient from one ward to another, commonly involves more than one record-keeping system. Information from the old ward needs to be accessible to the staff at the new ward.

We have studied the workflow at the hospital through two scenarios. The first scenario reflects the workflow when accessing information from more than one record-keeping system and the second scenario reflects the workflow when handling lab referrals, i.e., requests for results from lab tests taken on physical samples.

2.1 The Merge Scenario

In this scenario we focus on the need to access information originating in more than one record-keeping system. We use the Melior [Med07b] system and the CareVue [Med07a] system, record-keeping systems actively in use at the University Hospital, as examples.

For example, a nurse wanting to access all information about a patient residing in both record-keeping systems, would today need to access both systems separately.

It would be preferable to have one user interface where all information about a patient can be included in the same view. With this in mind the merge scenario looks like this:

A user will login to two different record-keeping systems and then access information from both systems through one user interface.

Both record-keeping systems require users to login before accessing information. However, it can be presumed that the user name and password for a user is the same in both record-keeping systems.

The record-keeping systems store information about the same patient. Due to this there is some overlap in information between the two systems.

2.2 The Referral Scenario

The referral scenario is based on the current workflow for handling of lab referrals at the hospital. We have studied the workflow, between one of the laboratories at the hospital and the Melior record-keeping system.

The laboratory uses a locally developed system called DECLab, linked to the hardware used for analysis, to store information about referrals.

The Melior system consists of one Melior server and a group of Melior clients, which connect to the server to get information which is presented to users.

The current workflow for sending a lab referral has two perspectives. The first perspective is that of a person, for example a nurse, sending a lab referral:

1. A nurse fills in a referral form on paper. An example of a standard referral form is shown in Figure 1.
2. After filling required information on the front of the referral form, the nurse attaches stickers found on the back to test tubes containing physical samples. These stickers connect the test tubes to the referral.
3. When finished, the nurse sends the referral form and the test tubes to the laboratory.
4. The nurse waits for test results from the laboratory. Test results will be available via a Melior client when ready.

The second perspective is that of the laboratory:

1. The referral form and test tubes arrive at the laboratory.
Figure 1: A standard referral actively in use at the University hospital in Lund. The top picture shows the front of the referral and the bottom picture shows the back of the referral. The front of the referral includes a referral ID, fields to fill in orderer ID, patient ID, test time etc and a list of tests grouped after test tube types. The back of the referral contains a couple of test tube stickers along with instructions on how stickers are applied to test tubes.
2. The form is scanned and OCR processed in order to locate identification of samples, on which, tests are to be made.

3. The test tubes are put into machines for analysis.

4. When test results are available they are sent to a local AMTrix server which passes them on to the appropriate recipient, in this case the Melior server.

The biggest problem in this workflow is that it is tedious and ineffective for the laboratory to use referral forms on paper. The laboratory receives approximately four million referrals a year which becomes more than ten thousand referrals a day on average. A future system allowing for electronic handling of referral forms would relieve the staff at the laboratory from a lot of tedious scanning work.

The referral forms going to this laboratory will always include physical samples, that needs to be marked with some identifier. Today this is done with stickers. However, the number of stickers on the back of a referral form is static. A future solution must provide a more flexible way of connecting test tubes to referral forms. Solutions involving sticker printers exist and could be integrated into a future solution.

With these things in mind, the referral scenario, from the perspective of a person sending a referral, looks like this:

A user fills in an electronic referral form in a Melior client. After sending the referral form the user attach stickers received from a nearby sticker printer to test tubes and send them to the laboratory. The user waits for test results which will be available via a Melior client when ready.

The same scenario from the perspective of the laboratory:

The staff at the laboratory receives test tubes and puts them into different machines for analysis. When results are available, the DE-CLab system will send them to the Melior system.

This suggested future workflow eliminates the need for papers and, hence, the need for scanning. This allows for the staff at the laboratory to focus more on other things than administration.

The people sending referral forms are not as affected by these changes since they still need to fill in the same amount of information and send physical samples.

3 Palpable, Ubiquitous, Pervasive computing

During the last fifty years computing has moved from big mainframe computers to small personal computers and beyond. The number of computers grows constantly as new areas become computerized and new gadgets are invented. Some say we are experiencing the dawn of a new era with ever-evolving gadgets such as mobile phones, ipods, PDAs etc. More and more different gadgets need to function in the same environment. It is a grand challenge to find new techniques addressing issues such as integration, integrity, modeling etc. The words frequently used to describe this new era of computing are ubiquitous, pervasive and ambient.

A system consisting of a multitude of heterogeneous devices easily becomes large and complex. We want a system which is easy to comprehend, expand and administrate. This is where palpable computing comes in, in the form of the PalCom project.

The PalCom project is an ongoing research project, founded by the European Union, which started in 2004 and which will end in 2007. One aim of this project is to focus on the mechanisms required for basic communication between ubiquitous devices, rather than to focus on what is communicated. Another goal is to make this kind of computing understandable.

Development within the PalCom project is scenario-driven. In scenario-driven development a couple of realistic scenarios are constructed which
put emphasis on desired functionality. Here are three examples of scenarios:

- **The Major Incident Scenario**
  At the site of a major incident a number of wireless biomonitors are placed on an injured person. These monitors measure blood pressure, pulse etc. Each biomonitor provide measures which contribute in giving an overview of the state of a patient. The challenge is to find a flexible solution which allows for these biomonitors to be used in any combination required.

- **The Tiles Scenario**
  Parts of the physical and cognitive training of children with Down’s syndrome are performed in swimming pools where floating tiles are used to build puzzles. These tiles, if put together correctly, form a picture or a word. Tiles put in a correct combination will light up. Doctors working with these children need to easily reconfigure tiles for different puzzles.

- **The GeoTagger Scenario**
  Landscape architects use gadgets such as digital cameras, global positioning system (GPS) devices, compasses, laptops etc. in their daily work. Each gadget provides information which needs to be combined with information provided by another gadget. To combine, for example, pictures with GPS coordinates is tedious and time-consuming work. An easy way of combining a camera with a GPS device would simplify tasks like these.

Within PalCom, a network connecting devices is used. Devices in this network offer services, similar to the kind of services you would find in a service-oriented architecture [HS05]. The services provided correspond to functions of, for example, a network camera, a database wrapper or a wireless GPS device. Communication between devices is performed using events, this communication is asynchronous and non-blocking.

Besides services, which connect to hardware or function as wrappers, a PalCom network contains software components. Software components provide services of a more generic kind which can be offered by an arbitrary device. A service combining pictures with GPS coordinates is an example of a service suitable for a software component. This is not a service known to reside within either a digital camera or a GPS device, but which is still needed in the GeoTagger scenario.

Devices, active in a PalCom network, announce themselves, and discover other devices, through a discovery protocol. On request a service will send its service description to all interested services on the network. A service description describes in commands and out commands offered by a service. A command has zero or more parameters, which contain values defined by *Multipurpose Internet Mail Extensions* (MIME) types [Arc07] such as text/plain or image/jpg.

3.1 Assemblies

Assemblies describe *paths of communication* between devices in a PalCom network. The assembly language is designed to be as simple as possible. Functionality which an assembly cannot provide should be put in a software component. An assembly can do the following:

- **Logic**
  The simplest form of assembly will wait for events and on the arrival of an event send one or more events.

- **State**
  Assemblies have internal states, in the form of variables, which can receive values on the arrival of events. The values of these variables do not affect which events an assembly will wait for or send.

- **Services**
  Assemblies can offer services called *synthesized services*.

- **Bindings**
  An assembly can put extra constraints on connections with, so called, binding. A binding can, for example, define a connection as mandatory which means that an assembly will not work without the connected service.

This is a way of handling situations in pervasive computing where mobile devices can disappear and reappear in a local area network.

Assemblies are described with the eXtensible Markup Language (XML) in the PalCom network. We use a similar more readable notation. An assembly can contain five main blocks:
assembly <assembly_name> {
  this = <global_service_name>;
  devices [ ... ]
  services [ ... ]
  connections [ ... ]
  script [ ... ]
  service_description <service_name> [ ... ]
}

The device block contains a list of devices that are of concern to the assembly. Each device is given a local name used within the scope of the assembly.

devices {
  <local_device_name> = <global_device_name>;
  ...
}

The service block contains a selection of services, available via devices in the device list, that are of relevance to the assembly. Each service is given a local name used within the scope of the assembly.

services {
  <local_service_name> on <local_device_name> = <global_service_name>;
  ...
}

The connection block contains a list of logical connections between services and the assembly itself.

connections {
  <local_service_name> on <local_device_name> -> this;
  ...
}

The script block contains a variable block and an eventhandler block. The variable block contains a list of variables. Each variable is given a MIME type and a name, local to the assembly.

script {
  variable {
    <mime_type> <variable>;
    ...
  }
  eventhandler {
    when <event> from <service> on <device> {
      send <event> to <service> on <device>;
      send <event>({variable}) to <service> on <device>;
      set <variable> = thisevent.<param>;
      set <variable> = <constant>;
      ...
    }
    ...
  }
}

The service_description block defines a synthesized service. The name of the service is given at the head of the section.

service_description <service_name> {
  command <cmd> direction (<in>|<out>) {
    param <param>
      with type <mime_type>;
    ...
  }
  ...
}

See Appendix A for the grammar of the assembly language.

3.2 The PalCom Browser

A browser has been developed within the PalCom project to simplify the construction of assemblies. The browser is developed as an Eclipse plug in [EP07] and is run from within an Eclipse application. See Figure 2 for a snapshot.

The browser connects to a PalCom network in the form of a device and in that way it discovers all other devices and services active on the network. Discovered devices and services are displayed in a graphical tree structure, called The Universe. This graphical display makes it easy for a user to get an overview of available devices. Assemblies are displayed in a similar tree structure reflecting their inner block structure. Figure 2 shows an example of an assembly tree along side of the universe tree.

To add connections between an assembly and a service a user can drag and drop a service from the universe tree to the assembly tree. This action will add the appropriate nodes in the devices, services and connections blocks of the assembly. By right clicking the nodes in the assembly
tree additional functionality becomes available. A user can, e.g., add when blocks, variables, service descriptions (the same as adding a synthesized services) etc.

Besides simplifying the construction of assemblies the browser also provides a way of controlling/testing services active in the PalCom network. The browser will, on demand, automatically generate a user interface for a chosen service. The content of the generated user interface maps directly to the service description of the chosen service. See Figure 2 for an example of a generated user interface.

3.3 GeoTagger Implementation

The GeoTagger scenario is an example of a scenario within the PalCom project. The following devices, services and commands are included in the scenario:

- A GPS device, providing a service gps with one outgoing command position(WGS84,NMEA-0183).

- A digital camera, providing three services which correspond to different parts of the camera hardware. One service photo which responds to user actions with an outgoing command photo_taken(). This service also has an additional incoming command store_photo(photo). The display hardware of the camera offers a service display with an incoming command show(coordinates). The third service storage, corresponding to some kind of temporary storage, has one incoming command sendme_photo and one outgoing command photo(photo).

- A PDA, running a software component providing a coordStuffer service which merges pictures and GPS coordinates. This service has one incoming command sendme_stuffed_image(coord,image) and one outgoing command stuffed_image(image).

- A laptop, working as back-end storage, providing a service photo_db with one incoming command store_photo(photo).

The communication flow between these devices expressed in an assembly looks like this:

```
assembly GeoTagger { 
  this = <global_service_name>;
  devices { ... } 
  services { ... }
  connections { ... }
  script { 
    variables {
      text/plain latestReadableCoord;
      text/plain latestStandardCoord;
    }
    eventhandler {
      when position from gps on gps {
        latestReadableCoord = thisevent.WGS84;
        latestStandardCoord = thisevent.NMEA-0183;
      }
      when photo_taken from photo on camera {
        send show(latestReadableCoord) to display on camera;
        send sendme_photo() to storage on camera;
      }
      when photo from storage on camera {
        send sendme_stuffed_image(latestStandardCoord, thisevent.photo) to coordStuffer on PDA;
      }
      when stuffed_image from coordStuffer on PDA {
        send store_photo(thisevent.Image) to photo on back-end;
        send store_photo() to storage on camera;
      }
    }
  }
}
```

4 Communication Model

In this section we will present a model, combining Petri nets and Statecharts, which we will use in the scenario implementations to express expected work-flows.

This section begins with a presentation of what needs to be expressed in the two scenarios before we continue with a presentation of Petri nets and Statecharts. We sum up with a discussion on how to combine the two notations.

4.1 Requirements on expressiveness

Before we pick a model we need to clarify what needs to be expressed in the two scenarios presented...
The following elements are included in the scenarios:

1. **States**
   The implementation of the merge scenario includes a user interface which allows a user to login and fetch information from several data sources. This user interface has at least two different states — *not-logged-in* and *logged-in*.

2. **Parallel split**
   In both scenario implementations the execution path, at some point, needs to be split into two or more parallel execution paths. For example, in the merge scenario, where a user should log in to more than one system, it is much more efficient to send events to both systems at the same time, instead of sending events to one system and then to another system after the first has finished. The same is true for the referral scenarios where small parts of the execution path can be split into parallel paths.

3. **Synchronization**
   Sometimes when execution paths are split they need to be put together again. This is the case in the merge scenario where information originating in more than one source needs to be merged. The execution path is split into as many parallel paths as sources. To gather information from all these sources the parallel execution path needs to be synchronized into one execution path when all information is available.

4. **Sequence**
   The referral scenario includes operations which needs to be performed in a certain order — for example a referral id must be acquired before a referral can be sent.

5. **Instances**
   Both scenarios can potentially include several users. For example, a laboratory which receives thousands of referrals a day probably needs a system which can process several referrals simultaneously.

To aid in evaluating the expressive power of our model we use workflow patterns [vdAHKB03]. Workflow patterns are similar to design patterns [GHJV95] in the sense that they describe things that can be expressed within a language. Three of the basic control flow patterns fits our needs:

- **The Sequence Pattern**
At some point actions should be performed in a predefined order, i.e., one event should be sent before another event and so on. (Corresponds to requirement 4)

- **The Parallel Split Pattern**
  At some point the execution path is split into parallel paths, i.e., one event arrives and one or more events are sent. (Corresponds to requirement 2)

- **The Synchronization Pattern**
  At some point two or more parallel execution paths are joined into one execution path, i.e., waiting for one or more events to arrive, in any order, before sending one or more events. (Corresponds to requirement 3)

Expressed with workflow patterns we need a model which supports the sequence pattern, parallel split pattern and the synchronization pattern. To express these patterns we use elements from Petri nets. To support the remaining elements, states and simultaneous users, we use elements from Statecharts.

### 4.2 Petri nets

Petri nets, with a well-established theory [Pet77], [Mur89] dating back to the 60’s [Pet62], are commonly used to model and analyze systems [Pet81]. We are interested in the modeling aspects of Petri nets.

Petri nets supports the sequence pattern, the parallel split pattern and the synchronization pattern needed in our model.

#### Formal definition

A Petri net is a directed graph containing two kinds of nodes – places and transitions. Places connect to transitions, and vice versa. A place can contain zero or more tokens.

A transition will fire when each of its input places contain at least one token. When a transition fires it will consume one token from each input place and produce one token in each output place.

The formal definition of a Petri net graph looks like this:

**DEFINITION 1:** A Petri net structure, \( C \), is a five-tuple, \( C = (P, T, I, O, M_0) \). \( P = \{p_1, p_2, \ldots, p_n\} \) is a finite set of places, \( n \geq 0 \). \( T = \{t_1, t_2, \ldots, t_m\} \) is a finite set of transitions, \( m \geq 0 \). The set of places and the set of transitions are disjoint, \( P \cap T = \emptyset \).

\[ I : T \rightarrow P^\infty \] is the input function, a mapping from transitions to bags of places. \[ O : T \rightarrow P^\infty \] is the output function, a mapping from transitions to bags of places. \( M_0 : P \rightarrow \mathbb{N} \) is the initial marking, a mapping between places and a number of start tokens.

#### Example

Figure 3 illustrates how the sequence pattern, the split pattern and the synchronization pattern are implemented in a Petri net.

Figure 3: Illustration of the parallel split, the synchronization and the sequence pattern in a Petri net. Transition \( t_1 \) in the split net on the left will trigger when a token has arrived in \( p_1 \). Transition \( t_2 \) in the synchronization net on the right will trigger when a token have arrived in \( p_5 \) and in \( p_6 \).

An example of a Petri net using synchronization and a split is illustrated in Figure 4 on the following page. In the scenarios, places correspond to events. For example, in Figure 4 places could correspond to \( p_1 = \text{fetchInfo} \), \( p_2 = \text{fetchInfoA} \), \( p_3 = \text{infoAFetched} \), \( p_4 = \text{fetchInfoB} \), \( p_5 = \text{infoBFetched} \), and \( p_6 = \text{infoFetched} \).

The same Petri net expressed as a five-tuple:

\[ P = \{p_1, p_2, p_3, p_4, p_5, p_6\} \]

\[ T = \{t_1, t_2, t_3, t_4\} \]


---

7 A bag, sometimes called a multi-set, is a set but with the difference that each element can occur more than once [Pet62]. We use, for example, the notation \( P^\infty \) to denote a bag of places.
Hierarchical Petri nets

The main disadvantage with Petri nets is their poor ability to scale up when larger systems are modeled. The nets become large and complex and hard to grip. A Petri net variant called hierarchical Petri nets [Val79] address this problem by adding abstraction levels in the form of subnets to ordinary Petri nets.

A part of a Petri net promoting to become a subnet must start and end with a transition. In the example, in Figure 4 transition $t_1$ and transition $t_4$ could be the start and the end of a subnet. Subnets are expressed as dashed transitions which is illustrated in Figure 5.

To model cases where a device can return one of a set of out commands, as a response to an in command, we will allow subnets to end with more than one transition. An example of such a case is in the merge scenario where a system can return either a userLoggedIn or a userNotLoggedIn command as a response to a loginUser command. We add a delimiter to distinguish these subnets from ordinary subnets, illustrated in Figure 6.

User Interaction

The implementation of the merge scenario should provide a user interface. As a result some of the events in the communication model for that scenario will be generated by a user.

The same situation occurs in the GeoTagger scenario, mentioned as an example in Section 3. The digital camera in that scenario has a photo_taken event.
event which is triggered by a user. The GeoTagger scenario also contains a GPS device which periodically sends GPS coordinates to the GeoTagger assembly. The sending of coordinate events are triggered by a timer.

To express this in the model we create two new place nodes, illustrated in Figure 7— one node to express user interaction and one node to express timer interaction. User or timer interaction will trigger the creation of a token in these nodes.

Figure 7: Places from which tokens can appear. (a) This is a place which can produce a token via a user event such as the press of a button. (b) This is a place which will produce tokens based on a timer.

4.3 Statecharts

Statecharts, dating back to the mid 80's [Har87], are used to model reactive systems. The Statechart notation is similar to that used for state-transition graphs such as finite state machines [HU79]. Finite state machines have two drawbacks — scalability and orthogonality. Statecharts try to address these issues by providing notation to express hierarchical states and parallel states.

The merge scenario has at least two states — not-logged-in and logged-in. A logged in user can log out or fetch information indicating that the logged-in state might have substates. These substates are orthogonal, i.e., they are both active at the same time. Statecharts can express both substates and concurrent states.

We use the same Statechart notation as the one found in the STATEMATE language [HP98].

Notation

Statecharts can express hierarchical states, illustrated in Figure 8. The outermost state in a Statechart is called the root state. Internal states are called substates and states without internal states are called basic states. All states have a parent state except the root state.

Substates can be organized in two ways — as AND-states or as OR-states. A system which is in the parent state of a group of OR-states must also be in exact one of the OR-states. The Statechart, on top in Figure 9 illustrates OR-states.

A system which is in the parent state of a group of AND-states must also be in each AND-state. The bottom Statechart in Figure 9 illustrates AND-states. Statecharts use AND-states to model systems with orthogonal states.

Figure 8: An example of hierarchical states in a Statechart.

Figure 9: Examples of OR-states and AND-states. A system which is in state A in the top Statechart must also be in either state B or C. A system which is in state D must be in both state E and F.

Transitions between states are triggered by events. Possible transitions are drawn as arcs between nodes in Statecharts, illustrated in Figure 10. Arcs are equipped with labels explaining which event they correspond to. An event can trigger the execution of one or more actions. These actions are listed on the arcs next to the event name.

A special kind of arc called a default transition is used to distinguish the default state in a set of OR-states. An example of such an arc is included in Figure 10.
To deal with the need to handle simultaneous users we add support for instances. We need to define where instances are created and where they are thrown away. To handle these two cases we create an instance transition and a sink node, shown in Figure [11]. Instance transitions will, when encountered, create an instance of its target state. Sink nodes will, when encountered, throw away the current instance. A user should end up in the default state of the parent state, of the instantiated state, when a sink node is encountered.

Figure 11: Example of a Statechart including an instance transition and a sink node. In this case state C will be instantiated on entry.

It is possible to express the split and the synchronization pattern in a Statechart. As a comparison to Petri nets, these patterns are expressed in a Statechart in Figure [12].

Figure 12: An example of how the split pattern and the synchronization pattern can be expressed in a Statechart. Transition State B -> State C, illustrates a split and transition State C -> State F, illustrates a synchronization. The second transition will only occur when all AND-states in State C have reached their final state (the states with a black dot in the middle).

To solve this problem we combine the two notations in the following way:

- We let basic states contain Petri nets which execute according to Petri net rules.
- We keep the state hierarchies with AND-states and OR states from Statecharts.
- We only let state changes be initiated by transitions in a basic state and hence do not use transitions with event/action labels.
- We allow default transitions in basic states. On entry to a basic state, with default transitions, the target places of those transitions will receive a token.
- Default transitions pointing on states work as in Statecharts.
- We use instance transitions to specify if a state should be instantiated. On entry to a state, which is entered via an instance transition, an instance of the state will be created.
- We allow sink nodes to occur in basic states to specify when an instance should be thrown away.

4.4 A Combined Model

Neither Statecharts nor Petri nets suits our needs completely on their own. We want to express parallel splits, sequences and synchronization in a Petri net manner and at the same time express states and instances in a Statechart manner.
Figure 13: An example of a Statechart expressing the states in the merge scenario. An instance of InstanceState will be created when waitLogin is left. The same instance will be thrown away if a fail event is received in processLogin or after processLogout.

In this way we meet all of the five requirements mentioned in Section 4.1. An example model, including all parts of the suggested graphical notation, is shown in Figure 14.

5 Scenario Implementations

In this section we illustrate how the two scenarios, mentioned in Section 2, can be implemented with the current PalCom framework. For each scenario we distinguish devices, services and commands and define a model, in the same way as we did for the GeoTagger scenario in the previous section. Before we give an account of the scenario implementations we will introduce a software component required in the merge scenario.

5.1 A Synchronization Component

The merge scenario requires synchronization of events. The assembly language, in its current form, does not support synchronization – it is not possible to wait for more than one event in a when block.
To solve this problem we create a software component offering a **Synch** service with two commands: one incoming command `operand(numberOfOperands)` and one outgoing command `result()`.

The component will start in a state where it waits for a number of `operand` commands before it sends a `result` command. After a `result` command has been sent it will automatically return to its start state.

There is no way of knowing which `operand` commands that will reach the component first, so all `operand` commands are equipped with a `numberOfOperands` parameter. The first `operand` command that arrives will set the number of operands the component will wait for.

Figure 16 illustrates how the communication flow changes when the synchronization component is added.

**5.2 The Merge Scenario**

In the merge scenario, information is combined from two record-keeping systems — the Melior system and the CareVue system. In this implementation we simulate these systems, each simulated system is wrapped in applications acting as devices on the PalCom framework.

The two simulated record-keeping systems should contain information about patients. The information should, to some extent, overlap between

---

Figure 15: The expected workflow of the GeoTagger scenario. Transitions in the Petri net marked with GeoTagger correspond to when blocks in the assembly shown in Section 3. The remaining transitions in the net correspond to transitions inside services.

Figure 16: In the Petri net on the left two events are synchronized. The Petri net on right illustrates how the net on the left looks after additional places and transitions, required when using the Synch service, have been added.
the two systems. To avoid a too big implementation we use a small information set which we divide between the two devices. The information set looks like this:

\[(ID, Name, Address, Phone, Email)\]

We divide the information between the devices in this way:

**Melior:** \((ID, Name, Address)\)

**CareVue:** \((ID, Name, Phone, Email)\)

The two record-keeping systems in the merge scenario require users to log in before they access information. To support this we add login and logout commands to the services offered by the simulated devices.

A user will acquire a session id at log in, which is passed as a parameter when asking for information. The session id will be erased when a user logs out.

The implementation of the merge scenario should include a user interface in which a user can log in, access information and log out. To satisfy this requirement we use the automatically generated user interface of a synthesized service provided by the PalCom browser, see Figure 2 for an example of an automatically generated user interface.

### Implementation

Each simulated device provides one service, **MeliorService** and **CareVueService**, containing two commands, on the form `fetchData(sessionID,patID)` and `dataFetched(data)`, for each piece of information contained within the device. To handle logs in and log outs each service is equipped with four commands; `loginUser(userName,passWord)`, `userLoggedIn(SessionID)`, `logoutUser(sessionID)` and `userLoggedOut()`.

The assembly in this scenario is controlled directly via a synthesized service which is added to the assembly in the form of a service description block. This service, which we simply call **Merge**, contains three commands: the same log in and log out commands as offered by the services on the devices and two additional commands for information fetching: `(fetchInfo(SessionID, patID) and patInfoFetched(name, address, phone, email))`. The expected workflow in this scenario is described in Figure 17. The Petri net in the figure includes three subnets which are described in Figures 18, 19 and 20. The corresponding assembly has been split up so that corresponding transitions and when blocks are shown in the same figure.

The user interface generated by the PalCom browser, for the synthesized Merge service, is shown in Figure 21.

### Evaluation

The PalCom framework can not, or only partly, express what is needed in the merge scenario. With the five requirements mentioned in Section 4.1 in mind, we conclude the following:

1. **States**
   
   This scenario includes three user actions; log in, fetch information and log out. These actions are not independent. A user must log in before the other two actions can be performed.

   States can not be expressed in an assembly. As a result the user interface, generated by the PalCom browser, includes all actions at the same time. This layout gives the impression that it is possible to fetch information or log out without having to log in first.

2. **Parallel split**

   This is supported in the assembly language and is included in the implementation of this scenario. For example, the `fetchPatInfo` command triggers a split where the `fetchName`, `fetchAddress`, `fetchPhone` and `fetchEmail` commands are sent.

3. **Synchronization**

   We need to use an external synchronization component to implement this scenario. Each synchronization requires an instance of the synchronization component which means three instances in this scenario.

   The use of the synchronization component results in additional when blocks and variables in the assembly. Information needs to be stored in variables during synchronization. The `operand` and `result` events going in and out of the component are not designed to carry data. A generic component capable of synchronizing information would be difficult to design.

4. **Sequence**

   This is supported in the assembly language and is included in the implementation of this scenario. For example, the `fetchName` command will be sent before the `operand` command in the `MergeInfo` subnet.
Figure 17: The expected workflow of the Merge scenario and the corresponding assembly. When blocks with no corresponding transition in the Petri net have been left out to give room to the rest of the assembly. The three subnets included in this figure are described further in Figure 18, 19, 20.
June 2007
Software Integration in Health Care

**Figure 18:** The Petri net, describing the Merge\_login subnet in Figure [17](#) on the left and the corresponding when blocks from the Merge assembly on the right.

**Figure 19:** The Petri net, describing the Merge\_logout subnet in Figure [17](#) on the left and the corresponding when blocks from the Merge assembly on the right.
eventhandler {
    ...
    when fetchPatInfo from Merge on this {
        send fetchName(sessionID, thisEvent.patID) to Melior on meliorDevice;
        send fetchAddress(sessionID, thisEvent.patID) to Melior on meliorDevice;
        send fetchEmail(sessionID, thisEvent.patID) to CareVue on carevueDevice;
        send fetchPhone(sessionID, thisEvent.patID) to CareVue on carevueDevice;
    }
    when nameFetched from Melior on meliorDevice {
        tmpName = thisEvent.name;
        send operand to synch3 on synchDevice3;
    }
    when addressFetched from Melior on meliorDevice {
        tmpAddress = thisEvent.address;
        send operand to synch3 on synchDevice3;
    }
    when emailFetched from CareVue on carevueDevice {
        tmpEmail = thisEvent.email;
        send operand to synch3 on synchDevice3;
    }
    when phoneFetched from CareVue on carevueDevice {
        tmpPhone = thisEvent.phone;
        send operand to synch3 on synchDevice3;
    }
    when result from synch3 on synchDevice3 {
        invoke patInfoFetched(tmpName, tmpAddress, tmpEmail, tmpPhone) on this;
    }
    ...
}

Figure 20: The Petri net, describing the Mergetinfo subnet in Figure 17 on top and the corresponding when blocks from the Merge assembly below.
5. **Instances**

The assembly in this scenario implementation can not handle more than one user. This is due to the fact that variables, related to a single user, are maintained within the **Merge** assembly. Instances, along with states, would solve this problem. Neither is, however, supported in the current PalCom framework.

Another thing to consider, which is unrelated to the five requirements, is how the overlap of information is handled. Both services in the implementation provide the name of a patient. In the assembly name is picked from the Melior service. The same information provided by the CareVue service is never used. This indicates that it might be useful to have a constructs which makes it possible to list services providing equal information. If one service is incapable of answering, another service from the list can be asked instead. Bindings might be a way of constructing such a list.

### 5.3 The Referral Scenario

The referral scenario includes two systems: the Melior system, the same as in the merge scenario, and the DECLab system. The DECLab system, at the test laboratory, is connected to the equipment used for analysis. In the current workflow referral forms on paper are used as described in Section 2.

In the implementation of this scenario we simulate the two systems and wrap them in applications acting as devices on the PalCom network. The simulated applications both include a user interface. The simulated DECLab system includes a user interface which allows a user to send test results. The simulated Melior system is implemented as a client-server solution. The client includes a user interface which allows a user to send referrals as well as search for test results.

#### Implementation

The Melior device provides one service **Melior-ReferralService** which contains three commands. One outgoing command *SendReferral(LocalID, OrdererID, DoctorID, PatientID, TestTime, MsgToLab, TestList)* and two incoming commands *ReceiveTestTubeList(LocalID, ReferralID, TestTubeList)* and *ReceiveTestResult(ReferralID, Test, Result, MsgFromLab)*.

The DECLab device provides three services. The first one **ReferralService** has one incoming command *ReceiveReferral(ReferralID, OrdererID,
DoctorID, PatientID, TestTime, MsgToLab, TestTubeList) and one outgoing command TestResult(ReferralID, Test, Result, MsgFromLab). The second service ReferralIDService, handling referral ids, has one incoming command RequestReferralID() and one outgoing command NewReferralID(ReferralID). The last service TestTubeListService, which knows which tests that can be taken on the same sample, has two commands. One incoming command ReceiveTestList(LocalID, ReferralID, TestList) and one outgoing command SendTestTubeList(LocalID, ReferralID, TestTubeList).

The expected workflow, in this scenario implementation, and the corresponding assembly is illustrated in Figure 22.

Evaluation

The PalCom framework can not express all parts of this scenario. With the five requirements, mentioned in Section 4.1 in mind we conclude the following:

1. States
The assembly in this implementation maintains variables, which values change as a referral is processed. In order to handle more than one user at a time support for instances is needed, see the evaluation of requirement five below. The need to support states is related to the need for instances.

Neither states nor instances can be expressed in the current PalCom framework.

2. Parallel split
This is supported in the assembly language and is included in the implementation of this scenario. The expected workflow in this scenario implementation only includes one parallel split, i.e., the SendTestTubeList command triggers a split where the ReceiveTestTube and SendReferral commands are sent.

3. Synchronization
There is no need to synchronize events in this scenario.

4. Sequence
This is supported in the assembly language and is included in the implementation of this scenario. For example, the RequestReferralID command will be sent before the ReceiveTestList command.

5. Instances
From the point where a simulated Melior client sends a referral until the point where a referral is received in the simulated DECLab device the assembly maintains values related to that referral. If another client sends a referral before the first client is finished, there is a risk of corrupt data.

The assembly can not cache data belonging to two different referrals at the same time because there is only one set of variables. Several sets of variables could be constructed if support for instances and states where added to the current PalCom framework.

The biggest issue in this scenario is the lack of support for instances and states. If these two requirements where excluded the PalCom framework, in its current form, would have meet all requirements, since there is no need for synchronization.

6 Scenarios Revisited

The current PalCom framework can not express all required parts of the studied scenarios. In this section we illustrate how the expected workflows in the scenario implementations might look like if the PalCom framework would support all requirements mentioned in Section 4.1.

An example of how the expected workflows in the merge scenario and the referral scenario might look like are illustrated in Figure 23 and in Figure 24.

7 Extending Assemblies

To support the graphical notation, illustrated in Section 6, the current PalCom framework needs to support the five requirements mentioned in Section 4.1. Requirement two and four, i.e., parallel split and sequence are already supported by the framework. In this section we suggest changes which would add support for the remaining three requirements, i.e., states, synchronization and instances.

We start with a section about synchronization which is followed by a section about states and a section covering instances. We end with a section on timers. As a consequence of suggesting support for synchronization we also suggest support for timers.

7.1 Synchronization

In the implementation of the merge scenario we need to synchronize commands. There is no support
Figure 22: The expected workflow in the referral scenario implementation and the corresponding assembly.
Figure 23: The expected workflow in the merge scenario described using the notation defined in Section 4.
Figure 24: The expected workflow in the referral scenario described using the notation defined in Section 4.
for synchronization in the current PalCom framework and, hence, we need to use a software component offering a synchronization service.

The issue at hand is whether such a basic functionality as synchronization belongs inside or outside of an assembly? We will start this section with a list of observations before we move on to a presentation of suggested language changes. We end the section with an evaluation.

Observations

The use of a synchronization component in the implementation of the merge scenario sheds light on a couple of issues:

- **Internal states**
  The synchronization component provides a functionality similar to a Petri net transition, with more than one incoming place.

  A Petri net transition that requires more than one token will wait until all tokens have arrived before triggering. As a result, a transition has several possible internal states – one state for each possible combination of tokens.

- **Initialization**
  With several possible internal states it is important that a synchronization component is in the right state at the right moment.

  To be sure of this an additional `reset()` command could be added to the synchronization component. This command when received could reset all internal states and move to a start state. This would correspond to removing all tokens going into a transition in a Petri net.

  In cases where a parallel split is followed by synchronization, as frequently is the case in the merge scenario, a `reset()` command can be sent to the synchronization component before a split. However, in cases where there are no obvious split-synchronize-combinations it might be hard to find an appropriate place to send the `reset()` command.

  Another issue concerns a possible race condition. If we send a `reset()` command how can we be sure that it reaches the synchronization component first? An `operand()` command might get there first.

  The common solution to this problem is to add a hand-shake protocol. A hand-shake protocol in this context means additional commands and a potentially slower initialization.

- **Temporary storage**
  To support synchronization, an assembly needs to translate incoming commands into `operand()` commands which are sent to the synchronization component. After sending all `operand()` commands the assembly needs to wait for a `result()` before it can move on.

  The generic design of the synchronization component does not provide any way of sending information, except the number of commands, along with an `operand()` command or a `result` command.

  In cases where the incoming commands contain information needed after synchronization the information needs to be stored in the assembly. This applies to all information which appears in one `when` block and which should also be available in another `when` block.

  To avoid the administration of storing information in the synchronization case information would need to be passed through the synchronization component. This would require customized `operand()` and `result()` commands for each synchronization involving information.

- **Types**
  There are no types defining commands within the PalCom system. Commands are described by name and surrounding context. In this sense each command has its own unique type.

  When a `when` block is constructed, the event it will wait for is specified with its type, which means the name of the event and its source (service and device). If we were to create a software component with the same functionality, as a `when` block, it would need to have the same amount of information about an event available.

  Presumed that the software component should be reusable all type information would need to be passed over during an initialization phase. After receiving a list of events the component would need to connect to all devices producing those events.

  In the synchronization component no event data is used. The only information transmitted is the number of events, which is piggy-backed on the `operand` commands.

  A component constructed in this way is easy to put out of sync unintentionally. It only requires one extra `operand` event.
An extension of the when block would make type information more accessible. There would be no need to send away event information and the need for temporary storage would decrease.

**Language changes**

The current when block can be described like this:

```plaintext
when (<event>) do something
```

We suggest that synchronization is implemented as an and list, where the order in which events appear should not matter. This would make the when block look like this:

```plaintext
when (<event> and <event> and ...) do something
```

The order in which events arrive should not matter. In cases where the same event arrives twice, the last event should be picked.

To introduce synchronization into the assembly language we need to change the grammar of the language, see Appendix A for the complete grammar. The section of the grammar which concerns the when block looks like this:

```plaintext
EventHandlerScript : Decl ::= Variables:VariableList
EventHandlers:EventHandlerList;
VariableList:Decl ::= VariableDecl*;
EventHandlerList:Decl ::= EventHandlerClause*;
...
EventHandlerClause : Decl ::= <CommandName:String> ServiceExp [ CommandInfo ] Action*;
...
```

The `EventHandlerClause` production corresponds to a when block. If we rewrite the grammar, to allow and lists, we get:

```plaintext
...
EventHandlerClause : Decl ::= Event* Action*;
Event : Decl ::= <CommandName:String> ServiceExp [ CommandInfo ];
...
```

Another thing to consider is how events are referred to within a when block. When there is only one event we can reserve a special word, e.g., `thisevent` which always refers to that event. However, if we allow more than one event we need to distinguish events from each other.

There are two ways we can accomplish this. The first alternative is to use absolute names which means we would need to specify device, service and command name for an event each time we want to refer to it. This works fine provided that we do not allow duplicate events. It might, however, be tedious.

The second alternative is to declare variables local to the when block, i.e., equip each event with a variable valid within the scope of the corresponding when block.

We prefer the second alternative. To support it we need to change the grammar further:

```plaintext
...
Event : Decl ::= <CommandName:String>
    ServiceExp LocalName [ CommandInfo ];
...
```

Independent of solution the name analysis needs to be modified. During name analysis we make sure that events used actually exist, the same applies to parameters. The difference when shifting from one to many events is the number of times this check needs to be performed.

**Evaluation**

The and lists add synchronization support to the assembly language. This change removes the need to use a software component which means that fewer events needs to be sent back and forth.

The need for temporary variables when synchronizing events containing information is removed.

Initialization of an and lists can be done on entry to a state, which means there is no need to send initialization events.

**7.2 States**

States are needed in both scenario implementations. The need for states is strongly related to the need for instances, which is covered in the next section.

**Observations**

In the merge scenario, mentioned in Section 5, the assembly uses variables to store session ids. The values of these variables define the internal state. If the variables have values, that means a user has logged in, and vice versa.

It is not possible to include information about these internal states in the current assembly language. As a consequence, the synthesized service of the assembly provides log out and fetch information actions independent of whether a user is logged in or not.
We want to express states in a way which corresponds to how states can be expressed in the graphical notation, described in Section 4 and applied in Section 6. To sum up we want to express the following:

- State hierarchies.
- Transitions between states.
- Default transitions.

Services offered by an assembly should reflect the states of the assembly.

Language changes

We suggest that script blocks, which we prefer to call state blocks, include an init block where default transitions can be defined.

To express state hierarchies we suggest that state blocks are allowed to be nested. Transitions between states should be done with a goto statement.

Finally, it should be possible to exclude empty blocks within a state block.

An example of an assembly, corresponding to the communication model illustrated in Figure 23:

```plaintext
state Merge {
  init { goto waitLogin; }
  variables {
    text/plain sessionId1;
    text/plain sessionId2;
  }
}

state waitLogin {
  eventhandler {
    when (this.loginUser a) do {
      goto instanceState;
      send Melior.loginUser(a.userName, a.passWord);
      send CareVue.loginUser(a.userName, a.passWord);
    }
  }
  service_description service1 {
    command loginUser direction in {
      param userName type text/plain;
      param passWord type text/plain;
    }
  }
}

state InstanceState {
  init { goto processLogin; }
}

state processLogin {
  eventhandler {
    when (Melior.userLoggedIn a) and
    (CareVue.userLoggedIn b) do {
      sessionId1 = a.sessionId;
      sessionId2 = b.sessionId;
      invoke userLoggedIn;
      goto LoggedIn;
    }
    when Melior.userNotLoggedIn do {
      invoke userNotLoggedIn;
      goto waitLogin;
    }
    when CareVue.userNotLoggedIn do {
      invoke userNotLoggedIn;
      goto waitLogin;
    }
  }
  service_description service1 {
    command userLoggedIn direction out {
    }
  }
}

state processLogOut {
  eventhandler {
    when Melior.userLoggedOut and
    CareVue.userLoggedOut do {
      invoke userLoggedOut;
      goto waitLogin;
    }
  }
  service_description service1 {
    command userLoggedOut direction out {
    }
  }
}

state LoggedIn {
  init {
    goto waitLogOut, waitInfo;
  }
}

state waitLogOut {
  eventhandler {
    when Melior.userLoggedOut do {
      send Melior.logoutUser(sessionId1);
      send CareVue.logoutUser(sessionId2);
      goto processLogOut;
    }
  }
  service_description service1 {
    command loginUser direction in {
    }
  }
}

state waitInfo {
  eventhandler {
    when this.fetchInfo a do {
      send Melior.fetchName(sessionId1, a.patID);
      send Melior.fetchAddress(sessionId1, a.patID);
      send CareVue.fetchPhone(sessionId2, a.patID);
      send CareVue.fetchEmail(sessionId2, a.patID);
      goto processInfo;
    }
  }
  service_description service1 {
    
  }
}
```
The parts of the grammar concerning the script block and the service_description block look like this:

```plaintext
command fetchInfo direction in { 
  param patID type text/plain;
}
}

state processInfo { 
  eventhandler { 
    when (Melior.nameFetched a) and 
      (CareVue.phoneFetched c) and 
      (CareVue.emailFetched d) do { 
      invoke infoFetched(a.name, 
        b.address, c.phone, 
        d.email);
      goto waitInfo;
    }
  }
}
```

```plaintext
service_description service1 { 
  command infoFetched direction out { 
    param name type text/plain;
    param address type text/plain;
    param phone type text/plain;
    param email type text/plain;
  }
}
```

The support for states makes it possible for the PalCom browser to generate user interfaces which change appearance depending on state. For example, the user interface generated for the Merge assembly would require a user to log in before fetching information.

### 7.3 Instances

The assemblies, in both scenario implementations, should support more than one simultaneous user.

#### Observations

Things to consider, when considering support for instances:

- **Shared variables**

  Each instance of a state will have its own set of variable. Variables in a state are always accessible to a substate, even if there are several instances of a substate. To support this all variables need to be protected with semaphores.

  This is currently not the case, which is mainly due to the fact that when blocks are executed in a sequential manner in the current implementation.

- **Communication**

  With several instances of the same state we might end up in a situation where several instances send the same command to the same service.

  For example, several instance of the InstanceState, in the merge scenario illustrated in Section 6, might send a `fetchName` command to the MeliorService at the same time. To know where to send `nameFetched` commands the MeliorService must have a way of distinguish the different instances from each other.

  This kind of communication is probably something which must be supported as far down as in the communication protocol.
Language changes

We suggest that support for state instances is added to the PalCom framework. It should be possible for an assembly constructor to define points where state instances should be created and where they should be thrown away. These operations correspond to an instance transition and a sink node in the graphical notation presented in Section 4.

We suggest that two new statements – fork and kill – are added which correspond to these two actions. With these new statements the assembly in the referral scenario might look like this:

```plaintext
assembly Referral {
  script {
    eventhandler {
      when SendReferral ... {
        fork;
        send RequestReferralID ... ;
      }
      when NewReferralID ... {
        send ReceiveTestList(...);
        send SendReferral(...);
      }
      when SendTestTubeList ... {
        kill;
        send ReceiveTestTubeList(...);
      }
      ...
    }
  }
}
```

When the assembly manager encounters a fork, a new instance can be created, and when a kill is encountered, the instance concerned can be thrown away.

We need to change the grammar of the assembly language to support a fork and a kill statement. The part of the grammar concerning when blocks looks like this:

```plaintext
... EventHandlerClause : Decl ::=<CommandName: String> ServiceExp
  [CommandInfo] Action*;
...
```

If we add the fork and kill actions, the grammar looks like this:

```plaintext
... EventHandlerClause : Decl ::=<CommandName: String> ServiceExp
  [CommandInfo] [SessionAction] Action*;
abstract SessionAction : Decl ::=;
ForkAction : SessionAction ::= ;
KillAction : SessionAction ::=;
...
```

Evaluation

Support for instances make it possible for an assembly to handle more than one user at a time.

7.4 Timers

Adding synchronization to the assembly language introduces new issues which need to be addressed. when blocks which wait for more than one event have several internal states. Events can arrive in any combination and there is no limit on how many events that can be included in an and list.

Observations

By supporting synchronization one issue is handled but other issues emerge:

- **Evaluation**
  The occurrence of an event is handled as a boolean value – an arrived event evaluates to true and the opposite evaluates to false.
  The value of an expression depends on the evaluation time. An assembly will wait for events from the time it is started until the time when it is stopped. There is no specific time limit on how long an assembly will wait for an event during its lifetime.

- **Freshness**
  Events in an and list might arrive with long delays in between. In cases where information becomes outdated this could be a problem. In such cases it might be better to reset when blocks if certain events have not arrive within a predefined time limit, rather than to wait a while longer.

Language changes

Time outs specifying evaluation time should be added to deal with these issues. Assembly constructors should be allowed to define time outs for when blocks with and lists.

If an and list is timed out a user might want to perform an action. The merge scenario has an example of such an and list. If the when block waiting for userLoggedIn commands in the processLogin state would time out the assembly should move back to the waitLogin state.

It should be possible to define time outs on different levels in the state hierarchy of an assembly and it should be possible to redefine time outs in substates
or when blocks. The default evaluation time for all and lists should be the life span of the assembly.

We suggest that a time-out statement is added which can occur in init blocks or when blocks. The time-out statement should include time in milli seconds and a block of actions which should be executed if the time out is triggered.

A time-out statement which occurs in an init block defines the time out for all substate or when blocks on finer levels, relative to the state which is initialized. It should be possible to redefine evaluation time or the list of actions, or both, on any level. If only the list of actions is redefined the evaluation time should not need to be redefined.

```plaintext
state Outside {
    ...
    state waitForLogin {
        ...
        when (Melior.loginUser a) and (CareVue.loginUser b) do {
            time-out 1000 {
                goto waitForLogIn;
            }
            sessionId1 = a.sessionid;
            sessionId2 = b.sessionid;
            invoke userLoggedIn;
            goto Inside;
        }
        ...
    }
    state processLogin { ... }
}
```

or we could define the evaluation time for the whole state:

```plaintext
state Outside {
    init {
        time-out 1000 {
            goto waitForLogIn;
        }
        goto waitForLogIn;
    }
    state waitForLogin { ... }
    state processLogin { ... }
}
```

The grammar concerning possible actions look like this:

```plaintext
... EventHandlerClause : Decl ::= Event* Action*;
Init : Decl ::= [TimeOutBlock] GotoAction*;
...
```

### Evaluation

Support for a time out construct allows an assembly constructor to define the evaluation time for one, or several, when blocks. Support for synchronization, without support for timers, would mean the same evaluation time for all and lists.

**8 Related Work**

Health care has struggled with system integration problems [SG90] for a while. In this thesis we investigate how well a service-oriented architecture, such as the PalCom framework, could simplify integration at the University hospital in Lund.

Ideas concerning service-oriented computing and architectures have been around in the web development sphere for a while [HS04]. This service-oriented way of thinking is changing the way new web applications are developed [Pas05].

As the number of service oriented applications available increases new standards emerge. The World Wide Web Consortium (W3C) [WWC07] provides a multitude of different standards, e.g., one markup language called the Web Service Definition Language (WSDL) [W3C07a] with the purpose of defining web services and another markup language called Simple Object Access Protocol (SOAP) [W3C07b] with the purpose of defining the protocol used between web services. WDSL, compared to service descriptions in PalCom, have a much more function-oriented way of defining operations. Each operation, or port, has an input and output message. There is no connections between the in and out commands in a PalCom service, besides what can be understood from the names of the commands.

There are two main ways to combine web services: orchestration and choreography [Pel03]. Web service orchestration an external web service defines the interaction between web services. In web service choreography web services work on there own, but follow an agreed upon behavioral contract. PalCom assemblies support both orchestration and choreography, since they can both provide synthesized services, and in that way take part in the workflow, or only describe the workflow. W3C maintains a
standard for a choreography language called Web Service Choreography Description Language (WS-CDL) [W3C07] which combines web services defined with WSDL.

Another organization providing standards is the Organization for the Advancement of Structured Information Standards (OASIS) [OAS07]. They maintain, e.g., the Web Services Business Process Execution Language (WSBPEL) [OAS06] standard and the Universal Description, Discovery and Integration (UDDI) [WO07] standard. WSBPEL is a workflow language, handling both orchestration and choreography, supported by major products such as IBM’s WebSphere [IBM07] and Microsoft’s BizTalk [Mic07]. UDDI defines an ontology register where web services can be registered and looked up. There is no similar technique in PalCom, since no ontology is used to describe services.

The Internet, as we know it today, contains billions of web pages. The contents of major parts of these web pages are only understood by humans. There are no semantics describing the content. Most pages are described using the Hyper Text Markup Language (HTML) [W3C07] which only provides layout information. There is a vision of a so called Semantic Web [SHBL06] where information will be semantically defined and understandable to a computer. Ontology registers such as the UDDI register is a step in this direction.

An urge to standardize how to offer services and how to communicate information, also exists outside of the web development sphere. Producers of wireless or network-connected gadgets such as, e.g., mobile phones, digital cameras or CD players need standards to make their gadgets more accessible. An example of such a collaboration is the uPnP Forum [For07] which defines device types such as, e.g., CD player types and printer types. Another example is Bonjour [App07] which also defines protocols for communication of device and service info over a network.

We have, in part, used Petri nets to visualize the communication paths in the scenarios included in this report. Workflow languages such as Yet Another Workflow Language (YAWL) [For07], which focus on expressing workflows, exist. YAWL uses a Petri net based notation and supports all currently known workflow patterns. Both Petri nets, and its derivative YAWL, use a graphical notation to express workflows. Workflows can also be described with calculus notation [PW05] such as the π-calculus [Mil99].

A number of different notations similar to Statecharts and Petri nets exist. For example the JGrafchart language [EA04]. UML activity diagrams and UML state diagrams [SP00]. UML state diagrams use a notation similar to the one used in Statecharts. The notation used in UML activity diagrams looks like the Petri net notation with additional constructs. The JGraphchart language use a notation similar to Petri nets but with the difference that places are called steps. Steps can contain, e.g., substeps or actions which is different from, how we use places in the Petri net part of our model, but similar to how we use states.

Platforms, other than the PalCom framework, which address the problems of ubiquitous, pervasive and ambient computing exists, for example, a platform for Ambient Oriented Programming [AP07] which supports a language called AmbientTalk [DTVCDM06].

9 Conclusions

In this thesis we have investigated how well the PalCom platform solves the integration problems found at the University Hospital in Lund.

It becomes apparent when studying the two scenarios included in this report that there is a need to express workflows. The original question of whether the PalCom platform can solve integration problems boils down to the question of whether assemblies are suited to describe workflows.

Our conclusion is that assemblies with the help of software components can express workflows but that this approach is insufficient. The assembly language viewed as a workflow language would benefit from a modified when construct. The lack of support for the synchronization pattern is the main reason why there is a gap between the assembly language and other workflow languages. Additional support for synchronization would also tighten the gap between Petri net notation and the assembly language. This connection to Petri nets or calculus, such as the π-calculus, opens up for a more theoretical study of assemblies, e.g., studies of liveness.

Adding synchronization support into the assembly language raises questions concerning evaluation time. How long should the waiting time be between the point where, from a given set of events, the first event arrives until the point where the last event arrives? We suggest that a time-out construct is added which allows constructors of assemblies to define the evaluation time for a single when construct or a group of them.

The merge scenario uses an automatically generated user interface provided by the PalCom browser. There is no way of expressing states in an assembly.
so the provided user interface offers all user actions in one view. If automatically generated user interfaces are to be used as applications there needs to be a way of expressing states in an assembly. We suggest that a state notation, inspired by the Statechart notation, is added to the assembly language.

In both scenarios there is a need for an assembly which can handle more than one user. In the scenarios studied so far in the PalCom project this has not been the case. We suggest that constructs are added which make it possible to define where state instances should be created and where instances should be thrown away.

9.1 Future Work

The changes to the PalCom framework suggested in this thesis have so far not been implemented. To support the suggested changes modifications would need to be made in all parts of the framework related to assemblies – the assembly language, the assembly manager and the PalCom browser.

Further changes to the language allowing an assembly constructor to define actions related to exceptions might further improve the expressiveness of the language. For example, a construct, similar to the time-out construct, handling lost connections could be added.

The scenarios studied in this thesis have been implemented using simulated versions of the Melior system, CareVue system and DECLab system. Both scenarios could possibly be re-implemented using real applications.

The communication model presented in the thesis suggest that assemblies, more or less, can be constructed directly from a graphical notation. This could be investigated further.

The link to Petri nets and process calculus suggests that it should be possible to do, e.g., liveness analysis on assemblies.

Acknowledgements

Thanks to Boris Magnusson, supervisor of this thesis, to Görel Hedin, for guidance and proofreading, and to Patrik Andersson, head of the IT department at the University Hospital in Lund.

References


[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addision-Wesley, 1995.


A The Assembly Language

The assembly language grammar in JastAdd [Uni07] notation.

abstract Decl : AbstractXMLRepresentable ::=;
AssemblyDescriptor : InfoRoot ::= AssemblyInfo*;
AssemblyInfo : InfoRoot ::=<Released:boolean> Devices:DeviceDeclList
Services:ServiceDeclList
Connections:ConnectionDeclList
[ EventHandlerScript ]
| ServiceDescription |
DeviceDeclList : Decl ::= DeviceDecl*;
ServiceDeclList : Decl ::= ServiceDecl*;
ConnectionDeclList : Decl ::= ConnectionDecl*;
DeviceDecl : Decl ::= Name:Identifier URN;
ServiceDecl : Decl ::= LocalName:Identifier Decl:AbstractServiceDecl;
abstract AbstractServiceDecl : Decl ::=;
SingleServiceDecl : AbstractServiceDecl ::= ServiceName:Identifier DeviceUse<URNSuffix:String>;
AltServiceDeclList : AbstractServiceDecl ::= ServiceDecl:AltServiceDecl*;
AltServiceDecl : SingleServiceDecl ::=<Prio:String>;
ConnectionDecl : Decl ::= Provider:ServiceExp Customer:ServiceExp;
Identifier : Decl ::=<ID:String>;
DeviceUse : Decl ::= Identifier;
abstract ServiceExp : Decl;
ServiceUse : ServiceExp ::= Identifier;
ThisService : ServiceExp;
EventHandlerScript : Decl ::=<Variables:VariableList>
| EventHandlers:EventHandlerList |
VariableList : Decl ::= VariableDecl*;
EventHandlerList : Decl ::= EventHandlerList:Decl ::= EventHandlerList:Decl |
| EventHandlerList:Decl |
VariableDecl : Decl ::=<VariableType> Identifier;
abstract VariableType;
MimeType : VariableType ::=<TypeName:String>;
EventHandlerClause : Decl ::=<CommandName:String> ServiceExp |
| CommandInfo | Action |
abstract Action : Decl ::=<CommandInfo | Action |
AssignAction : Action ::= VariableUse ParamUse;
abstract ActionWithParams : Action ::=<Command:String> ParamValue:Use*;
SendMessageAction : ActionWithParams ::= ServiceExp;
InvokeAction : ActionWithParams ::= ServiceExp;
InvokeAction : ActionWithParams ::= ServiceExp;
SendRequestAction : ActionWithParams ::= ServiceExp;
InvokeAction : ActionWithParams ::= ServiceExp;
ConstantUse : Use ::=<Name:Identifier>;
MissingUse : Use ;
B Implementation details

This appendix contains further implementation details such as class diagrams and database content for the merge scenario and the referral scenario.

B.1 The Merge Scenario

An overview of classes involved in the merge scenario implementation can be found in Figure 25. The information divided between the two devices looks like this. The Melior device:

123456-7890, Calle Svensson, Storgatan 10, 23456, Lund
012345-6789, Bo Ek, Tunagatan 20, 34567, Lund

The CareVue device:

123456-7890, Calle Svensson, 046-786543, calle@spray.se
012345-6789, Bo Ek, 046-34567, bo@ek.org

The users, existing in both systems, have the following user names and passwords:

Admin:AdminPass
sigge:sigge

B.2 The Referral Scenario

An overview of classes involved in the referral scenario implementation is shown in Figure 26. Messages sent to a Melior server from a Melior client looks like this:

SEND_REFERRAL ordererID doctorID recipientID patientID testTime msgToLab testlist
FETCH_REFERRAL_IDS localID
FETCH_TEST_RESULTS referalID

The test list in the SEND_REFERRAL command has the following format:

test, test, ...

The communication flow between classes in the implementation of the server part in the referral scenario is illustrated in Figure 28.

The communication flow between classes in the implementation of the DECLab part of the referral scenario is shown in Figure 29.

The test tube list in the REFERRAL_IDS commands has the following format:

{referralTubeID, tubeType, comment, test, test, ...}+
Figure 25: Classes in the implementation of the merge scenario. Classes with double borders belong in either the PalCom library or the Java Standard Library. Classes with emphasized names are abstract.

Figure 28: Communication flow between classes in the server part of the implementation of the referral scenario. Active objects – threads – are marked with loops.

Figure 29: Communication flow between classes in the DECLab part of the implementation of the referral scenario. Active objects – threads – are marked with loops.
Figure 26: Classes in the implementation of the referral scenario. Classes with double borders belong in either the PalCom library or the Java Standard Library. Classes with emphasized names are abstract.