MASTER

# Reviewing SWAP

Diederen, R.

*Award date:*
2007

# Reviewing SWAP

By: Robin Diederen
Student at the technical university of Eindhoven, The Netherlands
Student 558931
robin DOT diederen AT gmx DOT net

Version 2.00
09-05-2007
Maastricht, The Netherlands

# *Introduction*

Neubergher & Hughes is a softwarehouse for Linux based solutions, aiming at middle sized companies. NLcom, the Dutch distributor and support center for Neubergher & Hughes products, is where I was located. Besides their official role, NLcom has always been active with the development of Neuberger & Hughes products. Both companies have been in this market for quite some years (NLcom 8, Neuberger & Hughes 15) and have a vast amount of (technical) knowledge on the production process of (Linux based) software.

During my graduation period I've worked together with both the CEOs and CTOs of both Neubergher & Hughes and NLcom. These people introduced me to the open source community. To this community I own quite some gratitude. Many opensource developers seem to be very helpful and eager to explain techniques, even in little details. Without this open community, doing what I did and writing this thesis would have been much harder.

Of course, the most important aspect of doing a graduation trajectory like this, is learning and extending knowledge. And I must admit, I learned quite a bit from this project. Not just technical (practical) knowledge, but I also extended my theoretical knowledge and I gained insight in how theory and practice can be combined.

**As this document contains information which may harm the Neubergher & Hughes company, this document should only be accessible for Neubergher & Hughes, TU/e and NLcom employees. If you happen to find this document elsewhere, please inform me (by email).**

# *Summary*

Like many modern networked collaborative systems, Neuberger & Hughes' Exchange4Linux, a groupware server for Linux, is implementing a SOAP interface. The goal is to open up the Exchange4Linux server to a wide variety of new and existing (extensible) clients and applications. By using SOAP, easy access should be possible.

For that reason, Neuberger & Hughes developed, for Exchange4Linux, an interface and communication protocol called SWAP. To a large extent, SWAP has been generated from existing software without any formal specification of its functionality. This report describes the result of a "reverse engineering" research to describe and validate SWAP.

To judge the design and implementation of SWAP, a research was conducted. This research was started from scratch and adapted to the needs of Neuberger & Hughes.

The reader of this document is supposed to have some knowledge about both the theoretical and practical sides of formal modeling methods (mainly Petri nets), process algebras, basic networking, webservices and databases. In order to make this report self-contained, some explications of the techniques used have been added. Further information (including proofs), can be acquired from the appendices and external sources, which are mentioned in the footnotes of this document.

Thanks go out to:

- Marc Voorhoeve, Jeroen-Martijn vd Werf, Yaroslav Usenko, Jeroen Wulp (TU/e)
- Ryan Hughes, Helmuth Neubergher (Neubergher & Hughes)
- Gino de Jeu, Mark van den Bogaard, Timo Henczyk, Rik Bisschoff (NLcom)
- Joshua Boverhof, Charlie Moad, Rich Salz (ZSI / PyWebCVS)
- Scott Nichol (NuSOAP)

Words and abbreviations, colored in blue, can be looked up in appendix B1.

# Index

# 1. Abstract

Since 2002, Neuberger & Hughes, a German company, situated in the town of Plochingen (near Stuttgart), has been developing Exchange4Linux. This piece of serversoftware constitutes a Linux-based alternative to the Microsoft Exchange server (which is a groupware platform). As the name suggests, the server is Linux based. As with many Linux products, Exchange4Linux (sometimes referred to as Exchange4Linux) is an open source product.

Recently, Exchange4Linux has been extended with a new interface. The protocol that defines and uses this interface is called SWAP, which stands for Simple Workgroup Access Protocol.

By extending the Exchange4Linux server with SWAP, client development can be done much faster, compared to when developing “native” clients for each specific platform.

SWAP was designed to (partially) overcome certain drawbacks. The idea is that by making use of SWAP, developing client software should become easy and fast. This would open up the world for the Exchange4Linux server.

SWAP has not reached a final state yet. As SWAP was not written from a formal specification, but directly coded (be it partially automated) from the Exchange4Linux servercode (and a whitepaper), there is no guarantee this products works correctly and meets its requirements.

This document is mainly about the verification of the SWAP protocol and the changes that need to be made in order to get SWAP to a final and stable version.

## 1.1 Exchange4Linux

Exchange4Linux is a workgroup server. Workgroup servers are mostly used in collaborative environments where centralized managed communication (through, amongst others, public agendas and tasks) is important. Users in the workgroup have access to both personal and public data. In workgroups typically supported by Exchange4Linux, this data can be email, calenders (agendas), contacts, notes and tasks.

From an architectural point of view, the Exchange4Linux server is quite different from the market leading product: the Microsoft Exchange server. The software was developed in a typical Unix / Linux tradition; when possible, existing components are tied together; only non-existing parts are developed (either from scratch, or by modifying existing software).

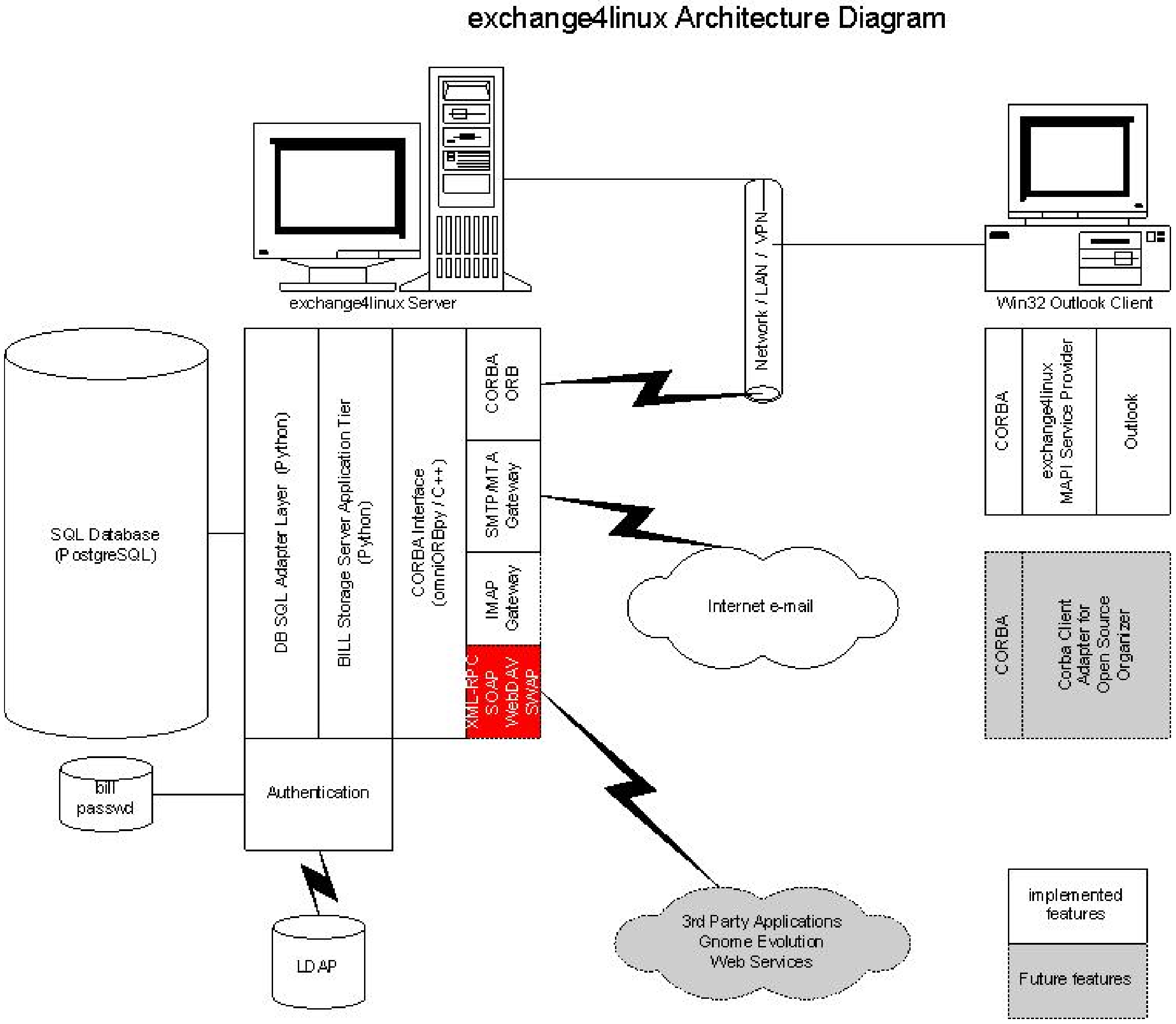

exchange4linux Architecture Diagram
exchange4linux Server
Win32 Outlook Client
Network / LAN / VPN
SQL Database (PostgreSQL)
DB SQL Adapter Layer (Python)
BILL Storage Server Application Tier (Python)
CORBA Interface (omniORBpy / C++)
CORBA ORB
SMTP/MTA Gateway
IMAP Gateway
XML-RPC SOAP WebDAV SWAP
CORBA
exchange4linux MAPI Service Provider
Outlook
CORBA
Corba Client Adapter for Open Source Organizer
Internet e-mail
bill passwd
Authentication
LDAP
3rd Party Applications Gnome Evolution Web Services
implemented features
Future features

*Illustration 1: ExchangExchange4Linuxinux architecture diagram*

Amongst others, Exchange4Linux is based on PostGreSQL (database), Apache HTTP (webserver) and OmniORB (CORBA implementation) and Python (programming language, which is used on the Exchange4Linux server). Exchange4Linux offers multiple interfaces, including a SOAP interface. SWAP rests on that SOAP interface. On the backend, the SOAP interface makes use of the CORBA interface. This CORBA interface has always been used as interface to the Exchange4Linux system. Before the SOAP interface was added to the Exchange4Linux server, the Exchange4Linux Outlook (by Microsoft) connector was the only possible client. This connector, which enables Microsoft Outlook to communicate with workgroup servers other than Microsoft Exchange, uses the CORBA interface on the Exchange4Linux server.

Because the SOAP interface interacts with the server backend (through CORBA) and one or more front ends (using the SWAP standard by exchanging SOAP messages), this interface can be seen as both a server and a client (Illustration 2).

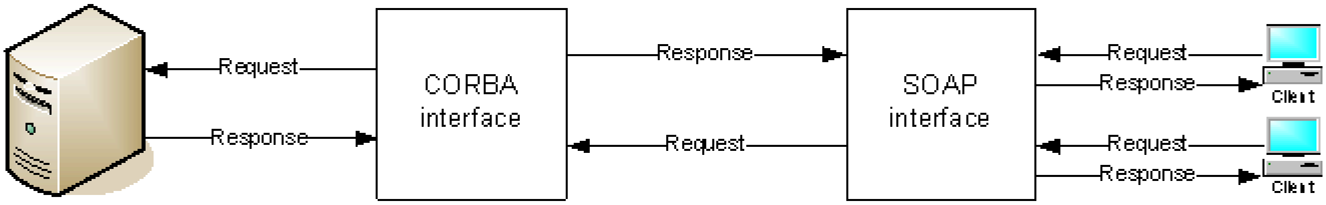


*Illustration 2: SOAP / CORBA communication*

Our main interest is the architecture and the client side of the SOAP interface.

## 1.2 SOAP

The SWAP protocol uses the SOAP protocol for messaging. SOAP, which stands for Simple Object Access Protocol, is the successor of XML RPC and was originally designed and backed by Microsoft.

SOAP was designed for transport over HTTP. The reasons may be obvious: HTTP is a broadly accepted and implemented protocol, making the SOAP protocol widely available. This allows SOAP messages to be sent over all HTTP enabled networks, including the internet; firewalls will not present any problems.

SOAP messages are formatted using XML, which is, in its standard form, a text-based markup language. Multiple messaging patterns are supported. The most common ones are RPC and DC document-style. RPC is a messaging style in which a client sends a message (request) in a predefined manner, to a server and the server answers (response); again in a predefined manner. Every RPC request and response comes with a method name and a set of attributes.

The document-style message pattern is not as strict as the RPC pattern is; in document-style, messages may be formed any way an author likes them to (as long as they are SOAP/XML compliant). Document-style messages offer more flexibility, but they are less advanced.

More information on the SOAP protocol can be found on [1].

## 1.3 SWAP Implementation

The SWAP protocol is implemented as a Python application, using the Zolera Soap Infrastructure (ZSI) Python component for the SOAP messaging part. With ZSI, SOAP messages can be composed, encoded, send, received and decoded with an acceptable effort.

The SOAP messaging style SWAP uses, is the “remote procedure call” (RPC) style. The messaging pattern of SWAP conforms to the RPC pattern : a user makes a call, which is handled and computed on the remote system, after which a response is returned. Despite what the names suggest, the messaging (RPC) style and messaging (RPC) pattern are totally unrelated. For SOAP, RPC messaging style is just a term for a certain SOAP authoring technique.

A more detailed explanation of SWAPs architecture and its position in Exchange4Linux is explained in the picture on the next page. All SWAP client requests and responses are sent as SOAP messages, formatted after the SWAP specification. With ZSI, messages are converted between Python objects and SWAP formatted SOAP messages. These Python objects are converted into (and from) CORBA objects, using the omniORB component.

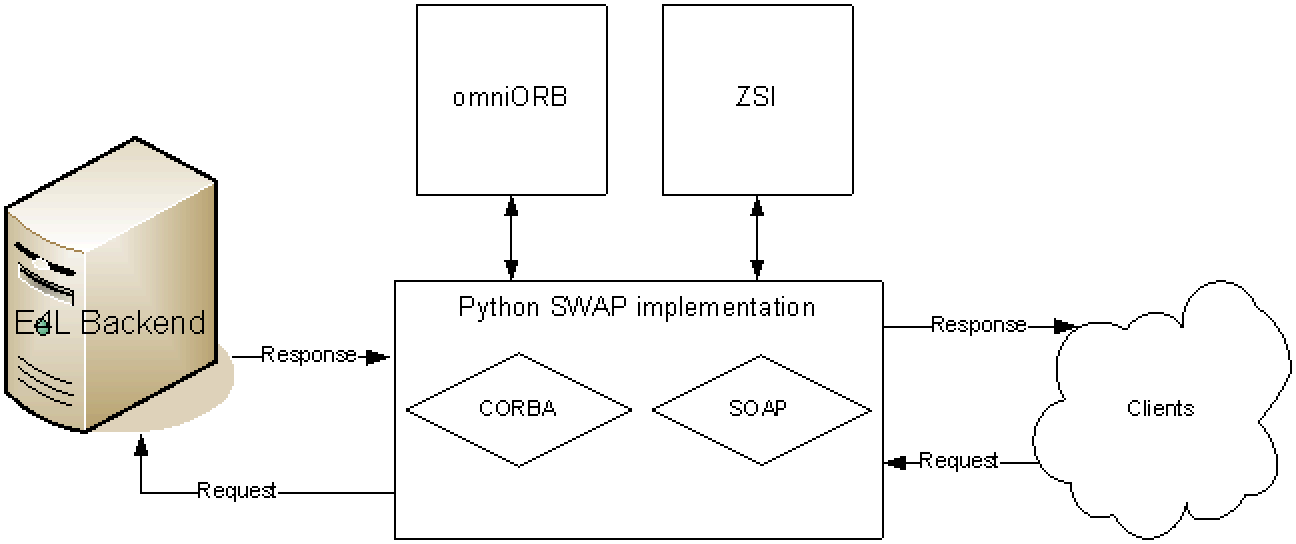


*Illustration 3: ExchangExchange4Linuxinux SWAP-related components*

The CORBA and SOAP implementations for Python, omniORB and ZSI are external components, supplied by third parties (open source projects).

# *2. Verifying SWAP*

SWAP abstracts from low level calls to perform operations on the Exchange4Linux workgroup server. By making a single SWAP call, multiple actions can be performed on the server backend. This way, SWAP acts as a simple, lightweight interface to a complex system. To make sure SWAP functions within the specified parameters, there are several subjects that are in need of research. Therefore, we need to validate SWAP against its requirements.

## 2.1 Requirements

For SWAP, there are no documented, formal specifications. There was no official set of requirements, so, in order to verify SWAP, its requirements have to be formulated, so a requirements study on SWAP has been conducted.

### *Requirement 2.1.1: stability*

We assume that the Exchange4Linux component is stable and fully functional. We also assume that for all components SWAP relies upon communication errors are properly dealt with. We therefor only concentrated on the proper termination from reachable states of the SWAP protocol itself.

We consider the following (typical) causes for system instabilities:

- communication errors (due to hardware errors or failing connections)
- deadlock situations (architectural problems or coding mistakes)
- other coding mistakes

### Subrequirement 2.1.1.1: Communication error handling

As SWAP is SOAP-based, we know that communication through SWAP will always be done using the HTTP protocol. This implies that the TCP/IP protocol will be used for data transport. The TCP/IP protocol has a built-in mechanism against basic (low level) communication errors. This mechanism, implemented in TCP and ICM, guarantees that data is transmitted and received properly. By calculating checksums, transmitted packets can be checked for errors. More on this subject can be read on [2].

Communication errors due to broken connections, cannot be handled by the TCP/IP built-in protection. The SOAP protocol does feature a fault detection mechanism, called the SOAPfault system. From errors generated by the TCP/IP stack,  SOAPfault can generate human readable error messages. More information on the SOAP faults can be acquired from appendix A12 and [3].

All programs using a SOAP framework fully supporting SOAPfault will react adequately at both low- and high level communication and deal with those errors as necessary.

We conclude that errors in the communication are fully caught by TCP/IP and SOAP; we do not need to do any checks on this.

### Subrequirement 2.1.1.2: Checking for deadlocks

In a deadlock situation, a running program (or session) cannot be executed any further. A deadlock is a state wherein no action can occur. Often, deadlocks in concurrent systems are caused by conflicts.

From a theoretical point of view, deadlocks are considered to be architectural flaws, as software architects / designers should, upon designing an architecture, make sure these errors cannot occur. In practice, deadlocks can also occur because of coding (programming) mistakes.

For deadlock checking, a model is needed. Models can be either language based (process algebra) or graphical.

### Requirement 2.1.2: platform independency

The SOAP protocol itself is platform independent. For any SOAP enabled platform, accessing any SOAP service should be possible. Typically both sides, client and server, translate between objects / variables in their local languages and messages encoded according to the SOAP standard. So, SOAP acts as both a communication layer and an in-between layer, often referred to as “middleware”. Data in SOAP messages complies to sharetypes as defined in the SOAP standard. During each translation, data provided by a sending program are translated to its equivalent in the SOAP standard and data in the SOAP standard is translated to equivalent data which fits the receiving program.

SOAP is mainly used for remote access of procedures and that is exactly what SWAP was developed for: accessing a remote, SWAP enabled, groupware server.

We know for a fact that most SOAP implementations do not implement *all* features of the SOAP specifications (SOAP 1.1 and SOAP 1.2), as specified by the W3C. Researching all possible platforms for SWAP clients is simply infeasible and from a business point of view, not really interesting.

What's really interesting to Neuberger & Hughes is whether certain platforms (and thus SOAP implementations on these platforms) can be used to access SWAP.

### Requirement 2.1.3: integration with existing products

During the last decade, a number of workgroup systems (servers and clients) has been developed and deployed. Although Microsoft's client, Outlook (not Outlook Express!), has been the market leader for years, there are some good alternatives. Most clients can be modified or extended to handle additional server types. Open source workgroup systems, such as the Zimbra collaboration suite or Scalix, can be modified to access Exchange4Linux through SWAP. Even open source webmail clients, such as Squirrelmail and HordeMail, should be able to make use of SWAP too (be it partially, as such clients often do not feature typical workgroup elements).

Although it would be preferable to test *all* potential clients for their compatibility with SWAP, this is simply infeasible. Therefore we decide to ease up our requirements on this and decide to make a tactical choice.

Also, with new technologies on the rise, current Exchange4Linux users may want to upgrade to more modern software. Open source developers may want to create programs that access the server through SWAP.

As there is a WSDL file for SWAP, which can ease up development seriously, we try to use it where possible. More info on the WSDL standard can be found in the appendix A11.

### Subrequirement 2.1.3.1: PHP

One of the most popular languages on the web is PHP. NH is interested in the question whether SWAP and PHP can work together. This is an interesting question, as there is an important difference between Python (in which SWAP was programmed) and PHP: Python is a strongly typed language, PHP is loosely typed.

### Subrequirement 2.1.3.2: Zimbra collaboration suite

NH considers The Zimbra collaboration suite as a possible upgrade to existing Exchange4Linux users. Zimbra features a SOAP interface which allows access to all of the servers functionality. The idea is to use this SOAP interface to put data from the Exchange4Linux database in Zimbras database. Here lies a task for SWAP, as SWAP can be used to extract data from the Exchange4Linux database using SOAP.

### Requirement 2.1.4: high performance

SWAP is meant to be used on a wide number of platforms. Besides regular desktop PCs, SWAP is meant to be used on servers and multiple mobile devices too. Mobile devices (often referred to as ultra-thin clients) require optimized software, as they typically have limited processing power, a rather small amount of system memory and connect through slow (often paid-per-kilobyte) wireless connections. Of course, non-mobile devices can benefit from optimized software too.

### Subrequirement 2.1.4.1: minimal overhead

SOAP is known for introducing some overhead; the protocol is text based which is not as compact as binary data. Binary data is encoded as BASE64, which can lead up to a 33% increase in size. ASCII data however, can be shrunk by about 500%.

Besides the overhead that comes with SOAP (there is little to be done about that, when using SOAP), the structure of the messages can introduce overhead to. This overhead is something we can control and which we want to keep to a minimum.

As the SWAP framework was partially generated from the CORBA backend, which never was designed for mobile use (or optimized for that purpose), we expect some significant overhead.

### Subrequirement 2.1.4.2: smart architecture

Besides optimizing the body of messages, the number of messages should be optimized too.

SWAP communicates in a typical (classic) RPC manner: a client sends a request to the server and the server answers. This communication pattern is very simple, but it is known to have a few drawbacks. SWAP, being based on this pattern, may (and most likely, will) have inherited those drawbacks.

The main problem lies in the simplicity of this communication pattern. All calculations are done server-side and each operation requires at least two (rather lengthy) XML messages (request, response) to be transmitted.

So, how can we optimize the architecture of the SWAP protocol to enhance its performance? We consider some well known techniques and reason about their integration in SWAP. These optimizations are described in appendix A7.

Note that the SOAP messages transceived by SWAP are RPC-style SOAP messages. Despite what the name suggests, this has not anything to do with the RPC communication pattern used in the SWAP implementation.

## 2.2 Modeling decisions

To make formal verifications possible, we decided to model the SWAP component using a formal notation. Most formal notations have some mathematical background. This implies that performing operations on the model, in its actual format or a converted format, can aid in certain analysis.

We choose to model SWAP as a collection of Petri nets, mainly because of the ease of modeling and the wide number of available testing tools. In chapter 3.1, we describe the approach we decided upon. Basically, we verify certain properties Petri nets that have been modeled after the SWAP source code.

The approach we took in converting from Python source code to Petri nets, is described in appendix A1. From now on, we will refer to this process as “mknet”.

For the actual modeling, we found that Yasper, a graphical Petri net modeling program, developed by the Eindhoven technical university, would suit our needs quite well. Yasper uses the PNML format, which is basically a XML document. This is good, as XML files can easily be translated to other file formats, by making use of XML style sheets (XSLT).

### *2.2.1 Modeling complications*

The “mknet” conversion has the property that tests in the code (influencing the flow of control) are replaced by a nondeterministic choice in the Petri net. We call this simplification data abstraction. It dramatically reduces the state space, allowing to check properties. However, by the same feat. caution is required when interpreting the results of these checks.

Let us explain this by giving a little example:

```
read x
while (x * x) >= 0 do
      x := x - 1
return x
```

This code is converted to a sound Petri net. Despite this, the code is nonterminating, since the square of any numerical value for x will be at least zero.

This also works the other way around:

```
read x
if (x * x) >= 0 then
      return x
```

The code does terminate correctly; however it is converted to an unsound net.

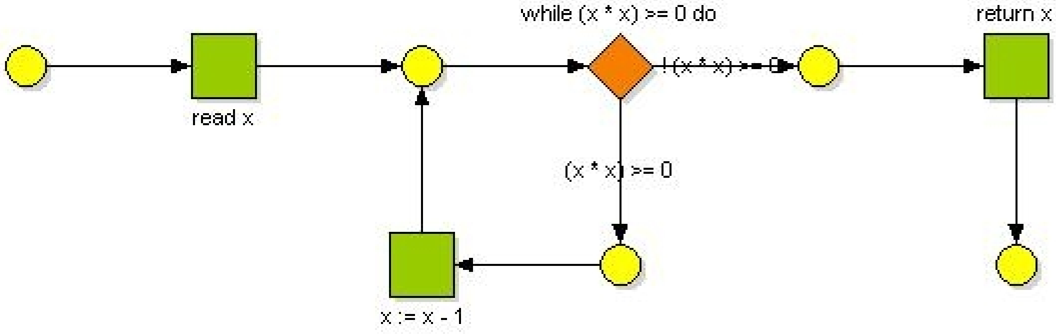


*Illustration 4: Sound Petri net for a nonterminating program*

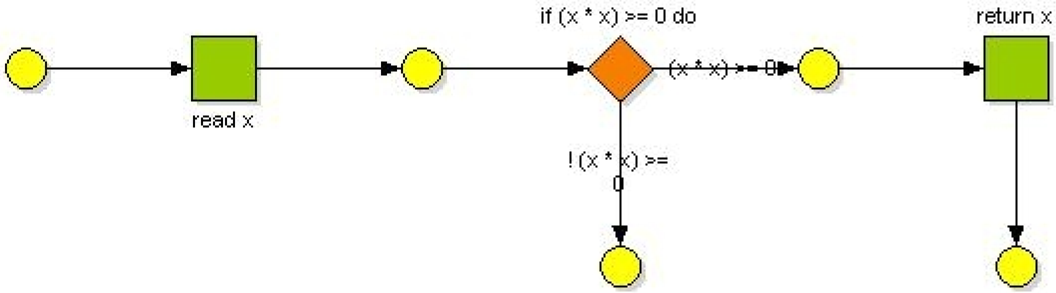


*Illustration 5: Non-sound Petri net for a terminating program*

The reason that we nevertheless abstract from data is that any approach taking the actual outcome of tests into consideration will lead to a spate space explosion that precludes any check. The sketched approach allows us at least to increase our confidence. We accept the fact that the proper termination in the net does not guarantee proper termination of the code.

We are also prepared to perform additional checks on the code when errors are found in its net.

Besides taking this approach, we also considered the use of colored Petri nets. Using colored Petri nets, we would not have to abstract from data. However, converting our (Python) source code to colored Petri nets would be a very complicated manner. Based on the number of plausible scenarios, when taking data into account, a state space explosion was expected to occur, even for very small colored Petri nets. Also, the number of analysis tools for colored Petri nets is rather limited and certain properties are for colored Petri nets undecidable.

### *2.2.2 Modeling and verification software*

To automate the testing of Petri nets, a toolset is required. Many toolsets for verifying various aspects of Petri nets exist. Different toolsets aim at different targets. We were in need of a soundness checker. After trying some, we decided to go with tools provided by the Eindhoven technical university (for practical reasons too). Two toolsets have been considered: mCRL2 and WofLan.

At first, a combination of Yasper and the mCRL2 toolkit seemed to be the most interesting deadlock testing. The mCRL2 toolkit provides a converter which can perform conversions from Yasper Petri nets to the mCRL2 process algebra. In the mCRL2 process algebra, we would not have to perform the data abstraction. As SWAP operates on a database and the (protocol) messaging itself depends on data too, we suspected that this data dependency might a possible cause of deadlocks.

Another tool for testing we considered is WofLan. WofLan is a pure soundness checker. The power of WofLan is the ability to check Petri nets, fully non-deterministic for non-soundness issues. One thing we have to keep in mind: by taking this approach, we totally abstract from data. By using nondeterminism to replace information lost by the data abstraction, this issue could be circumvented in an acceptable manner.

After some testing, we decided to go along with WofLan. The mCRL2 toolset was still under development and the stability of the product is not ready for production environments. Also, mCRL2 missed some important features (like support for hierarchy) and it did not scale well. Even for small Petri nets with minimal data dependencies, the generated state space grew too big for feasible model checking.

It has been decided we only want to do soundness checks on the control flow of the SWAP component (in a nondeterministic manner) and abstract from data. Given this situation, WofLan suits our needs perfectly. As we know that WofLan is a reliable tool for doing this, we have no reason to go with mCRL2.

Design documentation on SWAP is nonexistent. Therefore we performed our verification on the source code, with the added advantage of catching coding errors as well. For different parts of SWAP, different requirements apply. Because of this, we need to define exactly what requirement we want to test against.

# *3. Putting the tests in practice*

This chapter describes the tests we have done to verify whether SWAP meets the requirements specified in the previous chapter. We also explain how we did tests, why we made certain decisions and what our findings here are.

The tests described here are done with the most actual version of SWAP, version 0.7 that is.

Technical details, intermediate results and reasonings can be found in the appendices; when necessary, we refer to the section of the appendices containing the corresponding information.

## 3.1 Stability testing

### *3.1.1 Checking for deadlocks*

Before we can start with the actual deadlock-freeness testing on SWAP,  we do some explaining and some optimizations.

**Explanation of the architecture, related to modeling**

The SWAP protocol instances a component based architecture.

After connecting to a SWAP server, a client can perform one of the eight (main) operations SWAP features. From a functional point of view, these operations are independent of each other, but they all are a part of the SWAP protocol. Conclusion: we can model SWAP as a hierarchical Petri net; each SWAP operation will be modeled as a subcomponent of the SWAP supercomponent.

We distinguish these levels:

1. Package level (overview)
2. Session level (use-case like)
3. User level (expressed in SWAP calls)
4. SWAP level (expressed in SOAP calls)
5. Server level (these are CORBA OmniORB calls, which are treated as black boxes)

The first level defines the SWAP package. This level is used for initiating and ending SWAP. As this is a single transition with one input and output place (a minimal SMWF, which is always sound), this does not influence the results of any automated test.

The second level, the session level, gives an overview of the actions a user can perform in a session. These actions become available once a session has been started (after a successful authentication to the workgroup server).

As SWAP calls are pretty close to actual use cases and it is the correctness of the SWAP calls that we are interested in (not the architecture of a client), we choose to model the SWAP calls as if they were use cases. Such a use case can be viewed as an SMWF too: a call is initiated in its (only) input place, a single transition performs the call and the call ends in the single output place of that call. Of course, the “single transitions” are actually subnets which implies that a call is only sound if and only if its subcomponents are sound.

The third level, the SOAP level, defines how input data is picked up by the SOAP parser. After parsing, the data is submitted to the SWAP server, some calculations are done and the server transmits a response. If everything goes well, the data the server transmits is formatted as a SOAP message. In some special cases, in case of failures, the server generates non-SOAP (or even non-XML!) compliant messages.

These models of SWAPs functions are directly derived from the source code of the SWAP framework. This level always contains one SWAP server process.

The fourth level, the SWAP level, defines how the specified input data is rendered and submitted to the server. This level basically defines how a SOAP input is translated into calls for the CORBA backend. Our testing will be done at this level, as this is rather complex. After a SWAP call has been walked through, the response formatted or received by SWAP (depending on the correct or wrong termination of a call), will be transmitted to the caller.

Some SWAP calls make use of external functions. These functions are not part of the SWAP framework, but require SWAP to work correct. As these functions are black boxes, we model them as subcomponents containing one single transition (another sound, minimal SMWF net).

The fifth level, the server level, does computations at the backend, for the calls made by the SWAP operations. At this level, the high level SWAP calls are “converted” into multiple lower level calls. This conversion is done in multiple steps. First, a call is converted to its corresponding CORBA call. Then, such a CORBA call is submitted to underlying layers (lower level CORBA calls, calls to the BILL storage engine and to the SQL database). If all goes well, the server executes the requested action(s) and replies accordingly. The result is returned, though multiple layers, to the SWAP framework, which tries to parse this received data into a SOAP message, which is to be returned to the client.

In our tests, we tested single SWAP call against the backend. Our tests prove that the communication between such a SWAP call and the backend are free of errors. We did *not* test multi-user SWAP, as multi-user concurrency is something that is dealt with by the server backend, not by SWAP.

At this server level, many a thing can possibly go wrong. SWAP heavily depends on this server level; SWAP can only work correctly if the server backend works correctly too(!). For the verification of SWAP, no testing nor verification of the server backend was done. Testing this backend would require different verification techniques and was not asked for by NH. We reckon that this would be a hard and complicated matter, if possible (from a formal point of view) at all. As the server's correct functioning is an important aspect for testing SWAP, we *assume* the server to work correct. We assume that for every request made, the server generates a valid response. Also, we assume that no action performed by any other client, being executed at the same moment in time, influences the correct working of the server. These assumptions allow us to model the whole server level as a single transition. This also allows us to assume multi-user SWAP to work as required.

The assumptions we make are partially based on how the underlying database engine, PostgreSQL, works. Upon accessing the database (through SWAP or some other Exchange4Linux client), the records in the database which can be affected (by either that call or some other call on the same records) are snapshotted by the database engine. All actions in a transaction are performed on the snapshot. When all actions are performed, a commit follows. This commit is responsible for submitting mutations to the actual database. In case of problems, a rollback might occur.

Because of the snapshotting, we can argue that, upon accessing the database, each transaction operates on an abstraction (of the state) of the database. After all actions in the transaction have completed, the commit occurs. In case of problems, while performing the commit, a rollback can occur. This proves that all actions, up to the moment the commit successfully terminates, are inert. From this, we conclude that a commit can be considered to be an atomic action. So, this proves, that concurrent actions cannot lead to (additional) problems. This also proves that any commit is safe.

### *3.1.2 Mapping source code to Petri net constructs*

We chose to model our Petri nets to the control flow of SWAP (appendices A1 and A2).  SWAP only comes as source code; there are no (in-depth) documents on its architecture. Before we can start modeling this source code into Petri nets, we need to define how we will convert typical language constructs into Petri net equivalents.

Appendix A1 describes for each typical Python language construct how we chose to model a Petri net equivalent. The reasoning behind these Python-to-Petri-net mappings can be found in appendix A2.

We will give a short example on our conversion technique here. As start, we use the following Python code. We want to model the definition of this function (httphandler) as a classical Petri net.

```
def httphandler(req):
    func_path = ""
    if req.path_info:
        func_path = req.path_info[1:] # skip first /
        func_path = func_path.replace("/", ".")
    obj = getattr(server, func_path)

    # object cannot be a module
    if type(obj) == ModuleType:
        raise apache.SERVER_RETURN, apache.HTTP_NOT_FOUND
    if type(obj) != FunctionType:
        raise apache.SERVER_RETURN, apache.HTTP_NOT_FOUND
    return obj(req)
```

We first scan trough the source code. We notice that there is one starting point, there are three possible ways out, a few assignments and three if-statements. These constructs determine the control flow of our program. As we model our Petri nets to the control flow of the program, we can use our observations as a skeleton for the Petri net we want to build.

Our Petri net will have one initial place, connected to a single transition. There will be one output place too, but three transitions will connect to that output place. The if-statements are branching points in the Petri net which will be translated to a set of conflicting places/transitions (or XOR transitions, when using Yasper). Assignments will be pasted between the different constructs. As, for nondeterministic testing, there is hardly any use for these assignments / transitions, we could very well leave them out. Our Petri net clearly shows what we observed before. The source code of the program has been transcoded in a straightforward manner, including the control flow. The translation of the control flow is the most important part of the translation, as that is what we need for our testing (nondeterministic soundness checking).

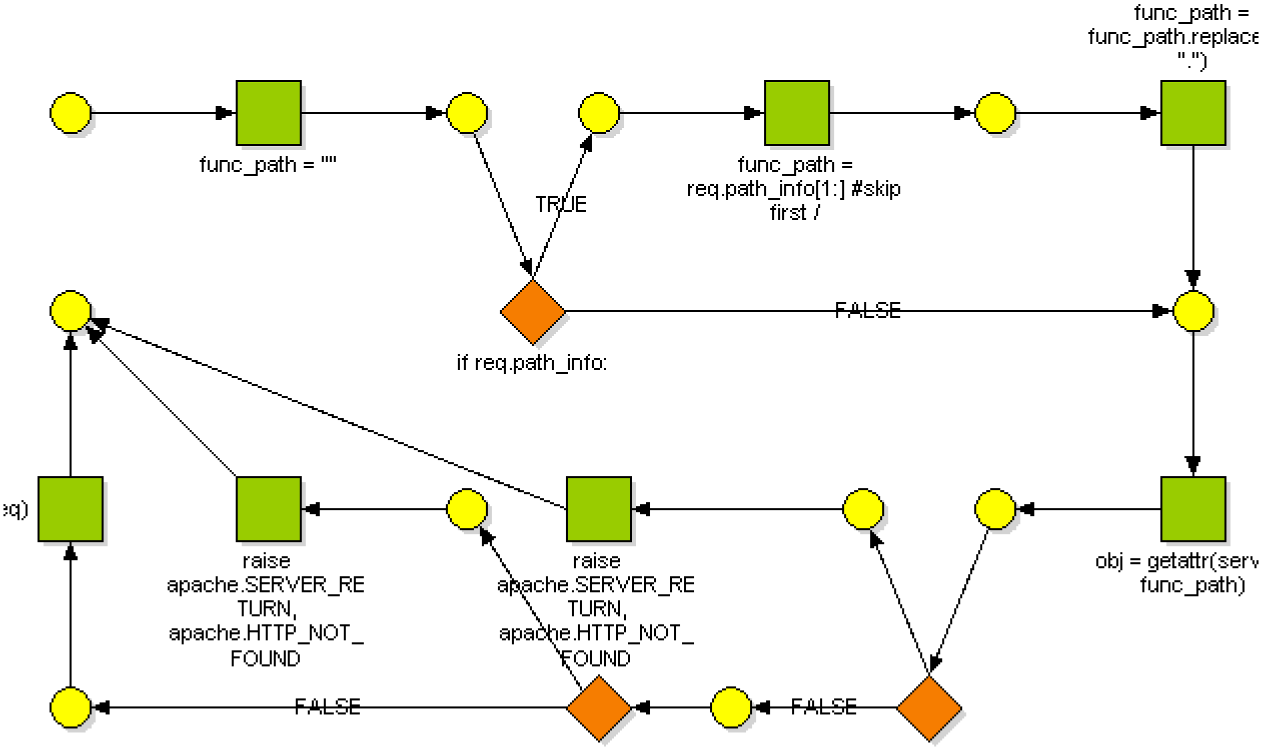

func_path = ""
TRUE
func_path = req.path_info[1:] #skip first /
func_path = func_path.replace ".")
FALSE
if req.path_info:
eq)
raise apache.SERVER_RETURN, apache.HTTP_NOT_FOUND
raise apache.SERVER_RETURN, apache.HTTP_NOT_FOUND
obj = getattr(serv func_path)
FALSE
FALSE

*Illustration 6: source code to petri net conversion*

During the translation from SWAP source code to Petri nets, we did compact the nets somewhat. In many cases, transitions can be subsumed without changing the behavior of the Petri net (modulo a branching bisimulation).

### 3.1.3 Building the source code model

As a first step in the translation and error-checking process, we use the self defined mappings (and technique) to create Petri nets from the source code. This is mostly a straightforward operation (the main difference is in the initial/final places/transitions of the mapping, as those often are combined with other places/mappings)

Applying these mappings results into the Petri nets which can be found in the appendix A3.

### 3.1.4 Filtering the source code model

The source code model is a direct copy of the source code itself (be it another language). Parts of that source code are unimportant to automated deadlock testing, so we want to leave those parts out, as testing on less complex models is less error prone and less intensive (thus safer and faster). By shrinking the Petri nets' size (count of nodes), we try to prevent a state space explosion from happening. When doing automated checking, which is usually done by the automated generation of a coverability graph, a state space explosion can easily occur.

After performing reductions, we want to be left with a less complex Petri net. If we perform this reduction well, the resulting Petri net should be bisimilar to the original net. To be precise, we want to create a branching bisimulation between the original Petri net and the reduced version. As a branching bisimulation is soundness preserving, we know that the reduced Petri nets are as sound as the original and more complex Petri nets are. The definition for bisimulation equivalence is borrowed from [4].

Petri net reduction is a subject which has been under research for over 20 years. There is a wide variety of reduction techniques; many of them show similarities. We decided to go with these 4 techniques:

1. Replacing subnets by single transitions
2. Applying predefined reduction rules
3. Parallel transitions reduction
4. Forced communication reduction

These techniques are explained in appendix A8.

#### Reductions in practice

We choose to combine the techniques mentioned above. By a combination of these techniques, we did success in achieving a sizable reduction.

The order in which the reductions are applied is important. If, for example, we use a reduction rule (option 2) on a net, we may break the ability to replace a bigger part of the original net by a single transition, resulting in a less optimal reduction. This issue necessitates a reduction strategy, which is described in appendix A8.

### 3.1.5 Flattening the model

After reducing the original Petri nets, we need to perform two more steps before our nets can be converted to an algebraic specification. The Woflan platform we are interested in to use for testing cannot handle the Petri nets as generated by Yasper. Woflan cannot handle Petri nets with hierarchy and XOR transitions, as they are no standard Petri net components.

These issues can be overcome by remodeling any hierarchical Petri net to an equivalent net, but without hierarchy and XOR transitions. This means that all (single) transitions defining a subcomponent will be replaced by the “full” definition of that subcomponent and all XOR transitions will be replaced by their classical Petri net equivalent. This will be done for each subnet on each level, so that we will end up with a non-hierarchical Petri net.

As a result of applying the flattening rules, we gain a bisimulation equivalence between the original net and the flattened model. This bisimulation guarantees that all the properties of the original net (including its drawbacks) are featured by the flattened net as well.

How these flattening conversions are done is explained in appendix A9[1].

### 3.1.6 Soundness check - WofLan specification checking

WofLan is a powerful soundness checker for (classical) Petri nets. For any Petri net, WofLan checks whether the net is sound and indicates possible causes in case of non-soundness. Just like Yasper, WofLan has been developed by the Eindhoven university.

Since the WofLan file format is not compatible with the Yasper file format, we have to do some conversions first, in order to get the Yasper-made Petri nets into WofLan readable format. First, we (again) apply the flattening conversions on the reduced nets. The intermediate result can  be used for making the final conversion to the WofLan file format.

It turns out that all SWAP components that should be sound actually are sound modulo data abstraction.

One component, the “Exchange4LinuxWeb” component, not being sound is actually what we intended for. Although this non-sound construct is no part of the original source code, we choose this to simulate the way SWAP really functions, as SWAP continuously listens for incoming connections. Of course, this could have been modeled as a sound Petri net too, but we prefer this livelock over a more theoretical approach. We reason this choice is safe, as this will lead to a non-terminating service spawning child processes.

[1] An automated LaQuSo flattening and XOR conversion tool has recently been developed.

| Level | Component | Check | Data |
|---|---|---|---|
| 1 | Exchange4LinuxWeb | Livelock required | Not sound |
| 2 | Authorization & session starting | May not deadlock, must terminate | Sound |
| 3 | CreateFolder | May not deadlock, must terminate | Sound |
| 3 | GetWorkgroupStorage | May not deadlock, must terminate | Sound |
| 3 | GetFolderList | May not deadlock, must terminate | Sound |
| 3 | GetLargeProperty | May not deadlock, must terminate | Sound |
| 3 | CreateObject | May not deadlock, must terminate | Sound |
| 3 | GetObject | May not deadlock, must terminate | Sound |
| 3 | UpdateObject | May not deadlock, must terminate | Sound |
| 3 | DeleteObjects | May not deadlock, must terminate | Sound |
| 4 | soap_CreateFolder | May not deadlock, must terminate | Sound |
| 4 | soap_GetWorkgroupStorage | May not deadlock, must terminate | Sound |
| 4 | soap_GetFolderList | May not deadlock, must terminate | Sound |
| 4 | soap_GetLargeProperty | May not deadlock, must terminate | Sound |
| 4 | soap_CreateObject | May not deadlock, must terminate | Sound |
| 4 | soap_GetObject | May not deadlock, must terminate | Sound |
| 4 | soap_UpdateObject | May not deadlock, must terminate | Sound |
| 4 | soap_ DeleteObjects | May not deadlock, must terminate | Sound |
| 5 | list_subfolders | May not deadlock, must terminate | Soundness is trivial |
| 5 | set_waiting_message | May not deadlock, must terminate | Soundness is trivial |

*Table 1: WofLan SWAP testing results*

### 3.1.7 Manual checking for deadlocks

In theory, deadlocks can be related to data. Petri nets constructs such as XOR transitions and conflicting places, can lead to such data-related deadlocks. Let us explain this with a little example.

*Example*
Suppose, a program , waiting for certain data to be provided to this (Pseudo) code:

```
repeat
      receive (x)
until G(X)
```

, where G(x) is a condition on x. Now suppose this condition cannot be evaluated to true, no matter what value x holds.

Because we abstract from data, errors like these will be left uncatched, when using the method described thus far. As we want to catch such errors too, we decided on doing some research on problems to be expected. This was done by scanning the SWAP source code for vulnerabilities as described above.

While scanning the SWAP source code (as described in appendix A6) for sensitive points, we noticed that most of the points which we would expect to be unsafe have been implemented in a safe way. The bigger part of these potential vulnerabilities are if-statements, with guards excluding each other. From logic theory, we know that always one of these conditions must hold. This implies that these if-statements cannot lead to deadlocks. Mathematically:

$guard \vee \neg guard \Rightarrow TRUE$

i

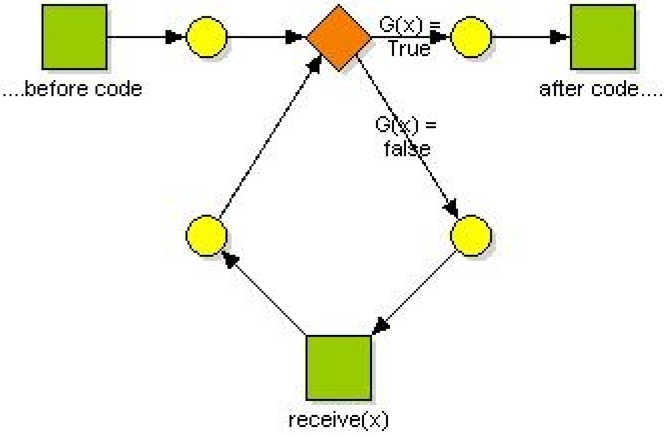


*Illustration 7: Petri net loop*

In our Python code, all if-statements have been implemented in a safe manner. Many if-statements only contain a single assignment, which never can lead to deadlocks. All other if statements have been augmented with a “raise” exception, which escapes the if-statement if an exception occurs. Of course, the same applies to the if-then-else and the if-then-else-elif statement.

We found that only one specific construct in the SWAP source code cannot fully be checked by using WofLan (or any other pure soundness checker on Petri nets). Since the construct is relatively simple, we decided to do this check manually. Appendix A6 describes this check.

### 3.2 Platform independency

From a theoretical point of view, SOAP based programs should run on any platform supporting SOAP. This should make SOAP-based programs practically platform independently. Neuberger & Hughes, with quite some experience in this field, is interested whether SWAP can cooperate with commonly used platforms.

As mentioned above, it is mainly the SOAP implementation that determines what can be done and what cannot. The server end, which is where SAP is situated, contains ZSI as its SOAP implementation. ZSI is Python-based and is regarded with Python as one of the best platforms for webservices, due to their mature SOAP and XML implementations.

It is well known that many SOAP implementations are far from complete. As SWAP does not use all the features of ZSI (and thus SOAP), SOAP implementations to be used for SWAP clients do not have to be fully compatible with neither ZSI or SOAP. A minimal requirement for possible SOAP clients is that an implementation to be used is at least compatible with all the SOAP features and types as used in SWAP. This also implies that SOAP 1.2 support is mandatory.

Researching SWAPs platform independency would actually be researching whether ZSIs features used by SWAP are compatible with the implementation of those features in likely (according to NH) client platforms. So, after identifying likely client platforms, we can check their specifications against SWAPs requirements.

SWAP comes with a WSDL (which stands for Web Service Description Language) file. These XML formatted files give the needed insight into the messaging structure of a webservice. WSDL files also describe how and where to connect to a webservice. Because WSDL files are actually XML documents, both authoring and reading them is fairly easy, for both humans and machines. Many modern development environments possess support for WSDL; they can generate WSDL files for webservices. WSDL files can also be used to generate SOAP client and servers implementations from. Also, many SOAP implementations possess some form of WSDL support; they either can convert a WSDL to client source code, or address a WSDL directly, so that SOAP messages are generated at runtime, from specified data and the WSDL.

The benefits of using a WSDL for us are clear; SWAP clients (often called stubs) can automatically be generated from this formal specification. This allows fast development and is less error prone than generating client implementations “by hand”.

More information on WSDL can be acquired from [5]. Appendix A11 gives a short description of the structure of WSDL documents.

### Typing problems

Communication with SWAP is done in RPC messaging style: the client sends a SOAP encoded message and the server responds with a SOAP encoded message which can be decoded by the client. Upon encoding and decoding messages, objects of the used language will go through some conversions. These conversions often include typecasting between SOAP/XML datatypes and the datatypes of the platforms on the server- and client endpoints. How these type conversions are done is specified in the SOAP implementation doing that conversion.

Although SOAP types should be identical for each SOAP implementation, the implementation of the language native types (on the server or client endpoint) can differ. Or worse: certain types might not exist at all.

Checking (efficiently) for typing issues requires a smart approach. From SWAPs WSDL, we know what types are used in SWAP. From ZSI, we know how these types correspond with Python types, so a logical approach would be to check the specification of Python types against their corresponding types in client implementations. This is not entirely safe though; as we know from our experience with a PHP client for SWAP, SOAP implementations might do type conversions, leading to unspecified problems. For more information on this subject, please read chapter 4.1.4, problem 3.

## 3.4 Performance testing

### 3.4.1 Overhead calculations

Calculating real life overhead is, in our case, impossible, as the amount of overhead depends on the type and amount of data that is transmitted in a message. Despite this, we can make some good calculations, showing where significant overhead can occur.

In order to get an idea how big the amount of (theoretical) overhead is, we evaluate the SOAP responses of SWAP. We only evaluate the responses, as we are in full control of the amount of overhead in the SOAP requests we make.

Some SWAP messages can be used for more than one single object type. Depending on the objecttype, the response from the Exchange4Linux server differs. So, to calculate the amount of overhead, we should, for each object type, make a call to the server and evaluate the specific response.

In the table below (table 2), we give the amount of theoretical overhead we calculated. On some of these calculations, we elaborated in appendix A10.

| *SWAP call* | *Overhead (theoretical)* |
|---|---|
| GetWorkgroupStorage | 0% |
| GetFolderList | 0% |
| GetContent (email) | 27% |
| GetContent (note) | 26% |
| GetContent (contact) | 24% |
| GetContent (journal) | 29% |
| GetContent (appointment) | 21% |
| CreateObject (email) | 27% |
| CreateObject (note) | 26% |
| CreateObject (contact) | 24% |
| CreateObject (journal) | 29% |
| CreateObject (appointment) | 21% |
| DeleteObject | 0% |
| GetLargeProperty | 0% |
| UpdateObject | Untested, call is broken |
| CreateFolder | 0% |

*Table 2: SWAP messages overhead (theoretical)*

Most of this calculated overhead is due to SWAP messages that contain redundant fields. As SWAP uses very generic messages, for multiple object types, it's common practice that some of the message fields remain unused. We also notice that the amount of overhead for messages not containing any data is small.

Those unused message fields are not much of an issue as these unused fields only add some plain text (tags), which is not much data, compared to the rest of the message, assuming the message contains besides the regular control data some object data too. This implies that calls submitting information (CreateObject and CreateFolder) will in practice have a low amount of overhead, despite the computed amount of theoretical overhead. This overhead we are willing to accept.

One might wonder why we are willing to submit redundant data anyway. The SWAP WSDL is the cause of this; upon sending out a CreateObject or CreateFolder request, an object for this type is instanced from the WSDL definition. Only the fields which are specified with data during the instantiation of this object will contain data.

There is another problem though: if, we have redundant message fields, these fields can contain redundant data too. Depending on the amount of data and redundant fields, the overhead can lead to too large messages. This is what happens in data retrieving calls; if a certain value is transferred in multiple fields (because of the compatibility issues), messages will easily grow big.

Suppose, for a worst case example, that an email with a(n) (binary) attachment is retrieved through SWAP. And, suppose, that the SWAP message contains this part twice. If the size of the whole SWAP message, without the attachment, is 10kb, and the attachment size is 10mb, than the overhead can come near to 99%, which is of course unwanted. In reality, in most cases, the overhead is not that severe.

### Working towards a solution

Optimizing the SWAP calls would lead to a lesser amount of overhead, but this would require the complete redesign of SWAP. SWAP was meant to be compatible with all versions of Exchange4Linux (≥3.00). Most of the superfluous data is there for compatibility reasons; different versions of Exchange4Linux require different key/value pairs, for the same data.

For now, Neuberger & Hughes does not consider performance the gain to outweigh the redesign costs. A future version of Exchange4Linux (version 4) will carry a filtered version of the entire interface.

From an architectural point of view, solving compatibility issues in the way which was done in the Exchange4Linux server backend, is not the best approach. As this solution, which is considered to be more of a work around, lowers the performance of the whole, we want to improve the situation, without modifying Exchange4Linux and the most part of SWAP.

By extending or modifying only SWAP in such a way so that every call with redundant data can be reduced to an optimized version (without redundant data) of that call, we can get rid of a fair part of the overhead, without making too much changes to the Exchange4Linux server and SWAP. This is not a pretty solution either, but given the situation, we consider it acceptable and it can be implemented in reasonable time.

### Current situation

When using ZSI to transmit SOAP messages, the SOAP messages are transmitted and received after some parsing steps (seen from a data-related point of view):

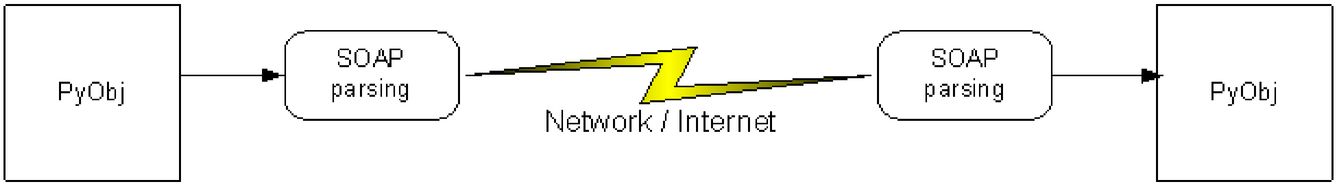


*Illustration 8: ZSI to ZSI communication*

Initially, in a ZSI-based program, a PyObj (Python object) is composed. This Python object contains both data and other descriptional information on the message to be created and send. After composing the object, a SOAP parsing module (ZSIs SoapWriter module in our case) takes over control and renders a SOAP request message, which is transmitted over a network (often, this network is the internet).

At the other side of the network, the same process happens, just the other way around. The decomposition of a SOAP message into regular application data (in our case, this is a Python object too) is done by another SOAP parsing module (ZSIs SOAPParsing module). After the SOAP message has been parsed, the data will be stored in a Python object again.

Because of the seriousness of the overhead in the data retrieving calls and the small impact of the overhead in the data submitting calls, we choose to aim at removing the overhead from the retrieving calls.

## Optimizing data retrieving calls

The server backend of the SWAP framework was created by partially exporting the Exchange4Linux CORBA backend by CapeClear's CapeConnect [7]. This export created SOAP calls for a (selected) set of CORBA calls.

Upon making a data retrieval call to SWAP, SWAP itself makes a request to the CORBA backend, reads the response by the CORBA backend and outputs this response in a SOAP message.

The conversion was done as follows:

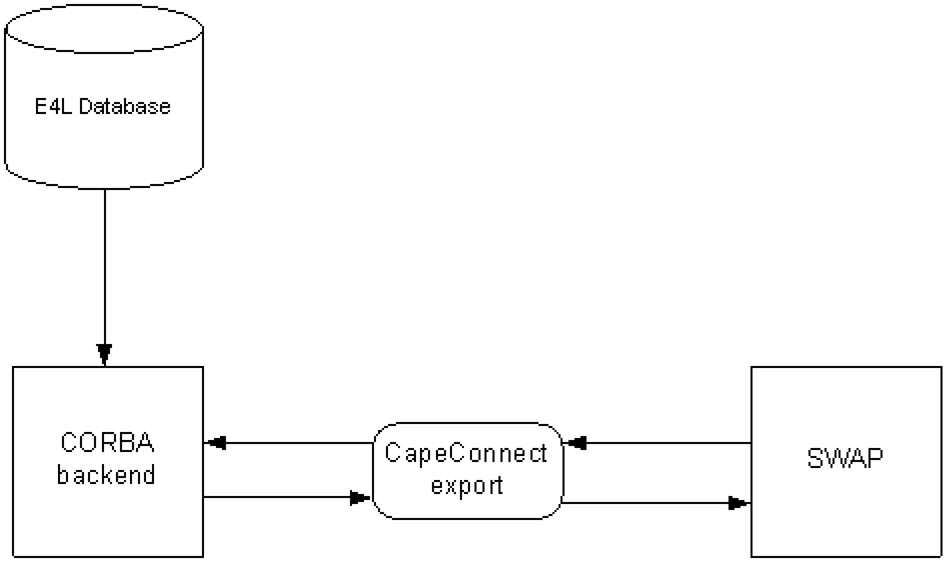


*Illustration 9: CORBA to SOAP export*

The CORBA messages are the ones that cause the redundant data. CORBA was used on Exchange4Linux before SWAP (and still is, as SWAP talks to the server through CORBA). When releasing new versions of Exchange4Linux, the architecture of the messages changes. To maintain the optimal compatibility with older clients, the CORBA calls have been kept compatible, by not deleting (or renaming) any keys in the messages, but only by adding new, often duplicate (seen from a data point of view), keys.

These duplicate keys are the main source of the redundant data (and thus the overhead).

The best solution to this would be to clean up the CORBA backend and re-generate / re-author SWAP after the new backend. This would break the compatibility with many clients though. As Neubergher & Hughes is planning a backend clean up for the next major version of Exchange4Linux, adding another interface is not interesting right now (as that might mean that in the near future, two different interfaces have to be remodified).

We consider multiple solutions to this issue, based on the idea of filtering unwanted data out. These solutions are described in appendix A13.

### Integration

The optimized solution we choose will be positioned before the parsing component of ZSI. The method we call "object filtering", described in appendix A13, filters unwanted attributes from objects, before transmitting them. This object filtering can be regarded as a form of data compression. The argument of the ZSI parsing module will thus become the filtered object instead of the one that was received from the (CORBA) backend.

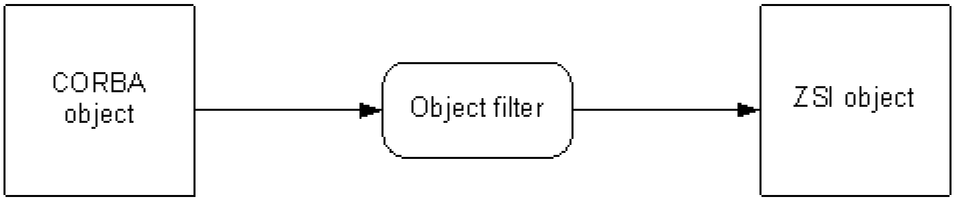


*Illustration 10: Object filter integration*

# *4. Putting SWAP in practice*

Here we describe the practical side of SWAP. We tested two scenarios in which SWAP is an important asset: a PHP application accessing SWAP and a Python application combining SWAP with another SOAP interface.

## 4.1 SWAP + PHP

PHP is one of the most prominent languages for web applications. Many webapplications are developed in PHP and PHP is supported on many (web)servers. This all makes the combination of PHP with SWAP quite interesting.

Before PHP version 5, there was no official support for SOAP in PHP. As SOAP has been used for several years, a few third-party PHP-SOAP implementations have been developed. Starting from the latest major release of PHP, PHP5, SOAP support is integrated into the language. We ran some tests with the native PHP-SOAP implementation, but this quickly showed some serious errors. Many SOAP operations are not or only partially supported. So we used the NuSOAP implementation. This is the oldest SOAP implementation for PHP, which is also considered to be the best matured and featured.

As PHP4 is still the most popular version of PHP, PHP4 compatibility is a must. NuSOAP can be used on PHP4, without any compromises.

### *4.1.1 Expected problems*

PHP is a loosely typed language. As both SOAP and the server, written in Python, are strongly typed, we will have to be on our guard for typing problems. PHP features less types than Python does.

PHP can “guess” what the type of a certain variable is;  every variable whose type cannot be guessed, results in a string typed variable. Also, complex types (structs, records) are unknown to PHP; PHP renders all complex types in multidimensional arrays.

So, we conclude, that it is important to check for typing errors, which may occur during the encoding and decoding of SOAP messages.

### *4.1.2 NuSOAP and WSDL*

NuSOAP supports WSDL files. Unlike many other SOAP implementations, NuSOAP can address a WSDL directly. When using a WSDL, NuSOAP can access a SOAP service by specifying that service from a WSDL and the data to send to it. Many developers regard this as a nice approach, as this allows fast and easy development of SOAP clients. The drawback of this, is that the SOAP encoding and decoding are done “on the fly”, resulting in less control over the messages.

### *4.1.3 SWAP functions*

Next in line is testing if all 8 of the SWAP functions can successfully be used from a NuSOAP/PHP implementation. Our findings are in the table below.

An example of the program code of the SWAP PHP client can be found in chapter 4.1.6.

| *SWAP function* | *Result* | *Comments* |
|---|---|---|
| GetWorkgroupStorage | OK | |
| GetObject | Almost OK | Incompatibility between server and client BASE64 implementation; see chapter 4.1.4, problem 1. |
| GetLargeProperty | Not OK | PHP cannot handle the chunks SWAP returns (chunk decoding error). The only solution would be to modify how the server returns requested data. |
| GetFolderList | OK | |
| CreateObject | Not OK | AnyType treated as a StringType; see chapter 4.1.4, problem 3 |
| CreateFolder | Not OK | AnyType treated as a StringType; see chapter 4.1.4, problem 3 |
| UpdateObject | Unknown, probably not OK. | The call is broken; UpdateObject extends CreateObject which does not work; see chapter 4.1.4, problem 3 |
| DeleteObjects | OK | |

*Table 3: SWAP functions from PHP*

As the table above shows, only 3 out of 8 SWAP functions work without problems. In the next section, we will investigate the existence and nature of possible solutions.

### *4.1.4 Encountered SWAP problems*

From our attempt to implement a NuSOAP/PHP SWAP client, we know there are a few problems. In this section we will investigate those errors and try to come up with solutions. NH prefers solution on the client side, since modifying SWAP would require quite some efforts.

### Problem 1: BASE64 decoding error

Although BASE64 is an official standard (RFC 2045 [8]), there are multiple ways to encode (and encode) special characters.

One of those special characters is the newline character (carriage return feed; often referred to as CR -carriage return-, LF -line feed-, or CR/LF).

The PHP implementation of BASE64, for what the newline character is concerned, differs from the Python BASE64 implementation. This results in a faulty decoding: newline characters are omitted. Python encodes a newline character as “0A” (hexadecimal), while PHP decodes this HEX code to “CR”, while it actually needs a “CR/LF” pair.

This error is not really a SWAP error, but rather a compatibility issue between PHP and Python. We could solve this issue by modifying the SWAP framework, but we will not, as this would result in a “dirty hack” (which is not interesting for this research). This would also result in some overhead, as we would have to send each BASE64 message twice of even three times (in case we want to support all three types of newline characters).

## Problem 2: typing errors

The following *simple* SOAP types are used in SWAP [9]:

| *SOAP Type* | *Python / ZSI supported* | *PHP / NuSOAP supported* |
|---|---|---|
| string | yes | yes |
| anyType | yes | limited |
| complexType | yes | yes |

*Table 4: Simple SOAP types*

### ComplexType types

The SOAP type complexType is used for various complex data structures. A structured data set, often referred to as a “struct”, which is used by many programming languages (including Python and thus SWAP), is not known in PHP. The same goes for other complex Python types, such as lists and dictionaries; all of those types are in PHP implemented with (multidimensional) arrays.

The multidimensional approach PHP takes is quite a bit more rudimentary than the approach of Python (and many other languages for that fact). The main drawback of this approach is that it is harder to create and parse these multidimensional arrays (resulting in lower performance too), but this does not prevent SWAP for working with PHP.

### AnyType types

SWAP also uses the SOAP anyType construct. As the name suggests, an anyType typed variable can hold values of any type. This variable is often used to send entire objects, like XML documents, in a SOAP body.

The SOAP implementations on both ends should know how to handle these variables, as the server and/or the client implementations should deal with them accordingly. When using an anyType typed variable, the type is changed upon assignment.

For instance: the server sends out an integer using an anyType variable. In that case, the SOAP implementation should convert the variable to an integer. On the client end, the anyType-as-integer variable is received and reconverted into an integer. This is a rather powerful feature of SOAP, as this allows more generic structures, keeping interfaces simple and easy.

Here lies a compatibility issue: SOAP implementations should be able to recognize types and convert them as such. This conversion should be a done to a type that is compatible with the type on the server end.

As PHP is a loosely typed language, we expected its abilities to (correctly) guess the supposed type of a anyType variable not as good as we need. NuSOAP being an open source product (written in PHP) allowed us easily to evaluate how NuSOAP treats anyType typed variables.

The answer: all anyType typed variables are automatically typecasted into stringtyped variables. Here lies a problem, as the SWAP server expects not just stringtypes in the anyTyped variable (but also integers and BASE64 encoded variables). This will give problems on the server end, as by the server, stringtyped variables will be treated as strings. This is a serious problem, rendering NuSOAP unusable for the SWAP framework as it stands.

A quick search on the web tells us that the anyType is a problem with many SOAP implementations. From a theoretical point of view, this is not an error in the SWAP framework, but in practice, it hurts the compatibility of SWAP.

Therefor it's advised to refrain from using the anyType, but rather specify a somewhat bulkier, though less problematic interface. This implies that changes to SWAP will need to be made, if we want better compatibility.

**Solution 1:** change the WSDL

By extending the WSDL with some additional messages, the anyType problem can be overcome. As we know for a fact, what the Exchange4Linux server is expecting in anyType fields, we can add messages to the WSDL which explicitly require such a typed variable to be entered. So, if we know that in a variable, we expect a integer (xsd:integer type), we make a new message, in which the anyType specification (xsd:anyType) has been replaced by an integer specification.

In big systems, this solution would be worthless, as many combinations may exist. Suppose a WSDL features messages which contain not just 1, but rather *n* anyType typed variables. Then suppose (as this is the case with SWAP), these variables can hold 3 types (string, integer, BASE64). The maximum number of messages would be $n^3$. The conclusion is clear: this is not feasible in big systems. A better solution is provided below.

**Solution 2:** change the server implementation

Modifying the server implementation in the same way we could modify the WSDL, by writing for each specific message a specific handler (server side SOAP services, for dealing with WSDL messages), would lead to the same kind of problem we described above: the number of handlers would easily explode. In practice, handlers are more complex than WSDL messages, so the result of this solution would even be less preferable!

There is a smarter way however. If we extend the handlers and the messages somewhat, we still can do with a limited number of messages. The idea is to add, for each anyType typed variable, another variable, which holds information on the type in the anyType. With this descriptional information added, the clients and server deduce the field's data type, making type guessing superfluous.

The drawback of this solution is the overhead this introduces, but since the number of anyType typed variables in SWAP (like in many SOAP based architectures) is quite small, we accept this drawback. As this additional message is short (for instance, an integer field, with a 1-digit code for each type), the introduced amount of overhead remains between bounds.

### *4.1.5 Conclusion on PHP/NuSOAP with SWAP*

In its current form, SWAP cannot be used with NuSOAP/PHP. We see that both PHP and NuSOAP have some drawbacks which could make a client implementation for SWAP as it stands, a hard or even impossible wish. By patching both sides for the errors described in chapter 4.1.4, these issues can be overcome.

NuSOAP, being an open source project, is under active development. Therefor it can be expected (but not depended on) that missing features will be added and incorrect ones improved. Bearing in mind these observations, NH will have to decide the future of SWAP with PHP/NuSOAP.

### *4.1.6 Some PHP/NuSOAP (SWAP) code example*

The code example hereunder gives some insight in the way PHP/NuSOAP works.

Direct connection to the WSDL:

```
$client = new soapclient_nu('http://localhost/Groupware.wsdl', 'wsdl');
$client->setCredentials('robin5', 'passme');
```

Here, we bind, with some credentials, to the WSDL file. Data from the WSDL file is used to establish the connection the actual SWAP server.

Arrays with input data, specified as in the WSDL:

```
$option = array ('key' => "OPKey", 'value' => "", 'status' => "");
$param_gws = array ('store'=> "robin5", 'options' => $option);
$gws = $client->call('GetWorkgroupStorage', $param_gws);
unset ($option);

$option = array ('key' => "2OPKey", 'value' => "ABLCDS", 'status' =>
      "2OPStat");
$param_gfl = array ('storageId' => "robin5", 'objectId' => "", 'options' =>
      $option);
$gfl = $client->call('GetFolderList', $param_gfl);
unset ($option);
```

This first two lines show how objects (represented in arrays) are composed. After composing the object, the object is transmitted. That is what the third line does. The second part of the program performs a similar action; an object is composed and transmitted.

Result parsing:

```
$message = $gfl['result']['message'];
print "<b><h1>Result of GetFolderList call: $message</h1></b>";
$count = count($gfl['folders']['item']);
print $count;
print_r ($gfl);
```

After the SWAP server has responded with a SOAP message, the response has to be parsed and decoded. Again, objects are represented as arrays, so we have to decompose those arrays into variables or sub arrays (which may contain arrays as well). For decomposing these arrays, we also use the WSDL, as this WSDL exactly describes the response that is expected.

Result printing:

```
print "<h2>List of all folders on the server:</h2>";
foreach ($gfl['folders']['item'] as $item)
  {
   foreach ($item['entry'] as $entry)
   {
      print "<font size=1><b>";
```

```
          print $entry['value'];
          print "<br>";
          print "</font></b>";
       }
      }
    print "<hr>";
```

This final code fragment shows how decomposed data is printed to the screen.

## 4.2 Zimbra collaboration suite and SWAP

Although details differ, most groupware systems have a common core. This core, which exists of typical objects (and operations on them), such as notes, contacts, emails etc, is available in about any groupware system. It should therefor be possible, in one way or the other, to make multiple groupware systems communicate with each other. In our case, we want to make use of SWAP on the Zimbra collaboration suite, in order to build a communication layer between Exchange4Linux and the Zimbra collaboration suite. SWAP can act as an inbetween layer for server-to-server use too.

As we want to connect the Exchange4Linux server and the Zimbra collaboration suite (short: Zimbra) through SWAP and SOAP, we make use of the ZSI (Python) implementation. ZSI is regarded as a mature SOAP implementation and SWAP was based upon ZSI, so, for the SWAP side of the implementation, we expect little to no problems. Also, by trusting in ZSI's maturity, we hope to avoid problems such as the ones we did face with PHP/NuSOAP.

Zimbra, including its SOAP interface, is developed in Java. The used Java SOAP implementation, JAX, is also considered to be mature, but not as feature-rich as ZSI is.

The open source edition of Zimbra possesses a documented SOAP interface. Using the document as a guide, we have developed a Python client for Zimbra, using ZSI.

As Zimbra is far more featured than Exchange4Linux is, we only need a part of Zimbras features in order to migrate data from the Exchange4Linux database to the Zimbra database. Because Exchange4Linux and Zimbra have a different data definition for their objects, there is an additional hurdle to take: how should the data be mapped? We will not worry about that for now; first we have to figure out if SWAP and Zimbras SOAP interface could possibly work together. If this can work, Neubergher & Hughes will determine how the data will be mapped.

Zimbras SOAP interface introduces another challenge, besides possible compatibility issues between the Java JAX and the Python ZSI implementation.

In SOAPs first few years, SOAP with WSDL was mainly used in RPC/encoded style. As a result of this, most SOAP implementations handle RPC/encoded style best. Zimbras SOAP interface however, requires document style soap (a so-called document/literal binding). ZSI can do this, but not as good and easy as it can do RPC/encoded style. This requires some investigation: can we get ZSI to meet Zimbras demands?

### *4.2.1 Expected problems*

We learned a bit from our previous experience with NuSOAP/PHP. Again, we will be using two different SOAP implementations and programming languages on the server and client sides and we expect compatibility issues. Although the gap between Python and Java is much smaller as the gap between PHP and Python (Java and Python are often considered as direct competitors), minor differences do exist.

From the documentation on Zimbras SOAP interface, we know that Zimbra requires document-style SOAP. It is well known that document-style SOAP is less supported than classic RPC style SOAP is. ZSI is known to be strong with RPC style SOAP and it can do *some* document-style SOAP.

### *4.2.2 ZSI and WSDL*

ZSI features an extensive support for WSDL. Like NuSOAP, ZSI allows direct access to the WSDL. The same drawback as with NuSOAP applies: for complex interfaces, the use of this direct interface may become infeasible. There is another way in which ZSI can use WSDL; through a converter called “WSDL2Py”. This converter can create the classes and client stubs required for accessing the SOAP service described by that WSDL. With this technique, it should be possible, that with little client code a Python client for the SOAP service can be achieved.

Upon calling WSDL2Py on a WSDL file, when the call is received, the converter should generate two files: a services and a types file. The services file creates the client stubs. These client stubs provide an interface to the underlying ZSI functions. When providing the correct data to the client stubs, using ZSIs underlying functions, these client stubs render the specified information and request into a SOAP message and request. The information to be provided is an object. This object is an instance of the corresponding message class in the messages file.

*Client stub example:*

```
def Authenthicate(self, request):
    if isinstance(request, AuthRequest) is False:
        raise TypeError, "%s incorrect request type" % (request.__class__)
    kw = {}
    # no input wsaction
    self.binding.Send(None, None, request, soapaction="Zimbra_Authenticate",
        **kw)
    # no output wsaction
    response = self.binding.Receive(AuthResponse.typecode)
    if isinstance(response, AuthResponse.typecode.pyclass) is False:
        raise TypeError, "%s incorrect response type" % (response.__class__)
    return response
```

Here we see how a client stub is built. The most important aspect is the “Send” operation. The action called “Zimbra_Authenthicate” is transmitted; the object to be transmitted is the “request” parameter, which was provided to the function upon initialization. After calling the “Send” operation, a response is to be expected. This response is caught by the “Receive” call; the object called “response” is instanced with the data received by the “Receive” call. Note that either the request or response data doesn't meet its expectation (that is the case if it does not comply to its predefined form), exceptions will be thrown!

*Type class example:*

```
class AuthRequest_Dec(ZSI.TCcompound.ComplexType, ElementDeclaration):
     schema = "http://localhost/zimbra/public/zimbra.wsdl"
```

```
        literal = "AuthRequest"
        def __init__(self, **kw):
            ns = ns0.AuthRequest_Dec.schema
            TClist = [ns0.accountreqoptions_Def(pname="account", aname="_account",
                  minOccurs=0, maxOccurs=1, nillable=False,
                  encoded=kw.get("encoded")), ZSI.TC.String(pnam
            kw["pname"] =
                  ("http://localhost/zimbra/public/zimbra.wsdl","AuthRequest")
            kw["aname"] = "_AuthRequest"
            self.attribute_typecode_dict = {}
            ZSI.TCcompound.ComplexType.__init__(self, None, TClist, inorder=0,
                  **kw)
            class Holder:
                __metaclass__ = pyclass_type
                typecode = self
                def __init__(self):
                    # pyclass
                    self._account = None
                    self._password = None
                    self._preauth = None
                    self._prefs = []
                    return
            Holder.__name__ = "AuthRequest_Holder"
            self.pyclass = Holder
```

This code fragment shows how a message object is composed. The first few lines of code are for ZSIs internal use and are mostly automatically generated. The body of the "__init__" function is the most important, as here all message parts are declared. Our example has four parts: account, password, preauth and prefs. Note that prefs is a list, so this message part may contain more than a single value!

*Client code:*

```
def main():
  msgb = AuthRequest()
  msgb.Account = msgb.new_account('robin')
  msgb.Account.set_attribute_by("name")
  msgb.Password = "passme"

  loc = ZimbraSoapLocator()
  port = loc.getZimbraPortType (tracefile = sys.stdout)

  port.Authenthicate(msgb,[])
```

ZSIs client code is rather compact. After instancing an object for the request "msgb", data is added to the object. Then, the object is transmitted and the response is printed to the system's screen ("sys.stdout").

### *4.2.3 Exploiting ZSIs abilities*

After writing a WSDL for Zimbras SOAP interface (or better, for the parts of the interface we want to use), we used the WSDL2Py tool to generate a types file and client stubs for our client to communicate with.

While developing the Zimbra client software, we found ZSI to have a few problems with the Zimbra SOAP interface. It turns out that ZSI can do what we need it to, but not without making some slight modifications and using functions in a non-standard way.

### Problem 1. SOAP header objects are not rendered

During the conversion from WSDL to Python, message header objects, specified in the used WSDL, are not taken into account. The WSDL2Py tool just "skips" over this part of a message. As ZCS requires most messages to include a header, for security (authentication) purposes, we had to find another way to get headers rendered in the SOAP message.

With some help of the ZSI developers, we found a way to overcome this issue. The solution is to create an additional complexType in the WSDL file and modify ZSI to render a SOAP object between the header tags (minor modification). After instancing this object, describing the header, we supply this to the modified client stubs.

*Header object (non-WSDL compliant!):*

```
  <xsd:element name="AuthRequestHeader">
       <xsd:complexType>
          <xsd:sequence>
             <xsd:element name="tag one" minOccurs="0" maxOccurs="1"
                nillable="True" type="xsd:string"/>
             <xsd:element name="tag two" minOccurs="0" maxOccurs="1"
                nillable="True" type="xsd:string"/>
          </xsd:sequence>
       </xsd:complexType>
    </xsd:element>
```

The header object specified above is a "regular" WSDL element. The modifications we made to ZSI make it possible to render WSDL elements in SOAP message headers.

To get these header objects rendered, we need to make a some changes to the generated source code (client types) too. These changes are required (notation is in CVS style; the "-" stands for "line delete", the "+" for "line add")

(-) kw = {}

(+) kw = {'soapheaders': soapheaders}

(+) outheaders = self.binding.ps.ParseHeaderElements((NAMEOFTHECALL.typecode,))

(-) return response

(+) return outheaders,response

### Problem 2. Tag attributes are not rendered

As with the headers, also tag attributes are not rendered by WSDL2Py. That is, if the tag attributes are mentioned in a WSDL conforming to the WSDL standards. By re-authoring the WSDL, with an alternate construction for the tag attributes, we can get ZSI to render attributes, if specified.

*Attributed object (non-WSDL compliant!):*

```
    <xsd:complexType name="accountreqoptions">
       <xsd:simpleContent>
          <xsd:extension base="xsd:string"/>
          <xsd:attribute name="by" type="xsd:string"/>
```

```
        </xsd:simpleContent>
     </xsd:complexType>
```

By using the object described above as type for a message element, we can specify an attribute. In this case, the “by” part of the message is the attribute, which can be called from the client stubs and code.

To add the attribute to the message, we need to make another change to the by wsdl2py generated client stubs. This is only one line of code, again in CVS notation:

(+) self.attribute_typecode_dict["by"] = ZSI.TC.String()

### Problem 3. Defect SOAP tags

SOAP envelopes consist of several parts. Two important parts are the message header and message body part. ZSI encodes these message parts with <header> and <body> tags; Zimbra however expects these parts to be <soap:Header> and <soap:Body>. By changing the string value representing the contents of these tags, this problem was solved.

### Problem 4. Namespaces in header and body tags

Zimbra expects namespace information in the header and body tags. These namespaces tell Zimbra to what part of the SOAP implementation the data in the header or body tag should be provided.

For example, Zimbra may expect a message formed like this:

```
<soap:Header xmlns:=”urn:zimbraAccount”>
      header information
</soap:Header>
<soap:Body xmlns:=”urn:zimbraMail”>
      body information
</soap:Header>
```

ZSI does not support any data in the header and body tags. Without this extra data, Zimbra cannot determine for what part of the Zimbra suite the data is intended.

So, we need to add this information to our SOAP calls. ZSI uses the DOM component to compose the XML/SOAP objects, which more or less supports what we need. The implementation of the DOM model provided by ZSI can add prefixes to opening tags of a XML document. By slightly changing that implementation, we can turn the prefixing function into a postfixing function. Providing the namespace as an argument to the modified DOM method will create the required tags.

Although the underlying DOM implementation has support for postfixing for namespaces now, ZSI still cannot handle this. The ZSI methods calling the DOM methods (or the ZSI methods calling other ZSI methods which call DOM methods) were never intended to call this part of DOM.

By extending the DOM implementation with an option to postfix the opening tag of an object, we can add namespaces to the SOAP header and body tags.

For testing purposes, we created some static calls addressing the DOM extension, but we must admit that this solution is not very desirable. We have contacted the ZSI developers to find a better solution for this problem. Until then, the indicated bypass must be used.

### Problem 5

Upon receiving messages, ZSI does checks on namespaces. The way JAX embodies namespaces in the body and header tags is not supported by ZSI and confuses the program. As we know beforehand what namespaces will be submitted (as this is documented by Zimbra), we can safely remove this check.

### 4.2.4 Consequences of the modifications

Of course, modifying ZSI has a few consequences. We tried to keep the consequences to an absolute minimum, so ZSI can be used for other purposes too. Modifications to other part of the system may have consequences too.

- The WSDL we are writing is no standard WSDL (not WSDL 1.0 or 1.1 compliant). For our ZSI application this is no problem, but most likely our WSDL can not be used with any other WSDL-enabled SOAP implementation.
- The additional objects which describe the headers will not cause any problems with any other WSDL parser, as these are just additional objects without any dependencies on them in that WSDL. The modifications introduced for the attribute rendering will cause problems though, as they describe both the attribute and the message in another way than specified in the WSDL standard.
- A client cannot be developed in a straightforward way. After running a WSDL2Py conversion, changes have to be made to the generated code. If we update a WSDL, we have to re-convert that WSDL to Python stubs too. That means that all changes to the stubs which were made prior to the conversion will be gone.

  We tackle this drawback by generating a separate WSDL for each Zimbra call, resulting in per-call Python stubs. All these Python stubs can again be called from a single client. This solution is 100% transparent to all clients.
- Quite a few modifications had to be made to ZSI. We tried to keep ZSI in the best possible shape; instead of modifying ZSI functions, we added functions in which we modified their original implementations. However, there is no guarantee that this modified version will flawlessly support other applications using ZSI running on that same server (which can be the case on a SWAP enabled server)

  Also, ZSI cannot be updated as normally would be possible with a “regular” installation. At the time of writing, ZSI 2.0, the version we are using, is a non-final version. This implies that the maintainability of the server (as ZSI is a part of that) may become problematic. So, modifying ZSI may hurt its compatibility and does hurt its maintainability for sure.

Despite the issues described above, we decided to go along with this approach. Doing things this way should give us better insight in the features ZSI lacks, the possible pros of JAX over ZSI and the feasibility of communication between SWAP and other SOAP interfaces (as JAX is a very prominent SOAP implementation). The gained knowledge and experience is very worthwhile for both Neuberger & Hughes and the developers of the ZSI project.

Examples of the Zimbra WSDL, converted (modified) client files and the modified ZSI distribution are available on the CDROM this document is on.

### *4.2.4 Conclusion on ZSI with Zimbra/JAX*

Without modifying ZSI and pulling some tricks, communication between ZSI and JAX would not be possible. Our efforts have led us to the conclusion that ZSI lacks quite some important SOAP aspects and that the interoperability between ZSI and JAX is not as easy as we suspected it to be. Although this does not really disable ZSI (and thus SWAP) to be used with SOAP webservices on different platforms, the ability to quickly develop and deploy SOAP clients, as SOAP was intended for, is clearly out of reach.

If Neuberger & Hughes decides to develop a migration tool for Exchange4Linux users, these notes should, along with the additional information the CDROM, aid them in deploying a ZSI client for Zimbra collaboration suite.

## 5. Conclusions

The core activity of this project was the reviewing of SWAP. We did a requirements study and checked SWAP against its requirements. We argue that with SWAP, Neuberger & Hughes hit the right direction, but that there is still work to be done.

We are pleasantly surprised by SWAP being implemented safely; so far, we only found one serious error. Of course, we may have overseen problems, despite the fact we have good faith in the quality of the SWAP source code and our research.

SWAPs architecture could use some attention, especially if SWAP is to be used with a lot of simultaneous connections or for mobile use.

Also, SWAP reveals that some parts of Exchange4Linux need attention. As future versions of Exchange4Linux will probably resolve this issue, SWAP will benefit from this too. Together with enhancements described in this document, SWAP can enter its next stage.

All in all, we think that SWAP has a lot of potential. So, we conclude that Neuberger & Hughes hit a good direction with SWAP. There is still work to be done though. Especially the compatibility with SOAP implementations other than ZSI and the performance could be increased.

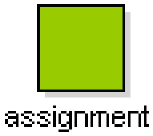
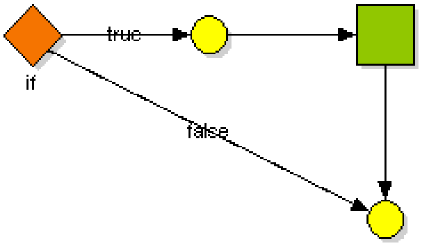
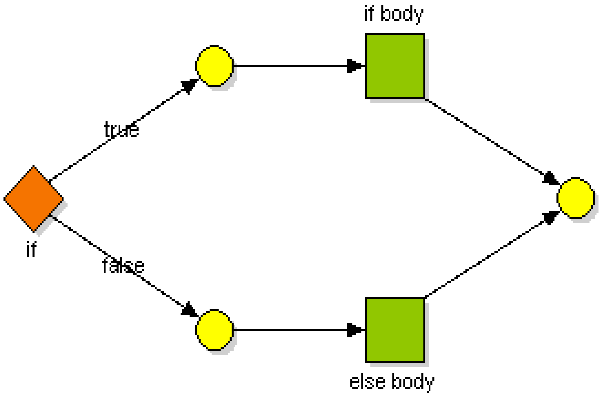
Besides the verification of the SWAP protocol, some research in various specific subjects has been done. These specific researches are mainly interesting to either Neuberger & Hughes or the Eindhoven technical university. We did research in methods to create formal models (Petri nets in our case) from source code and testing methods on those models.

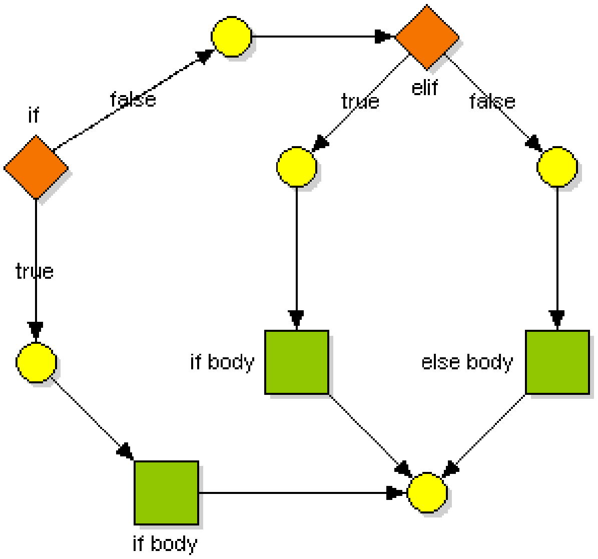
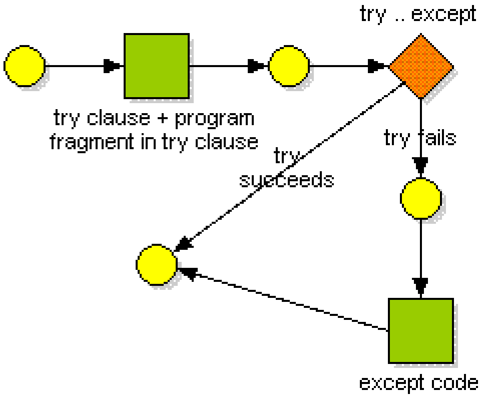
We also did some practical tests with SOAP applications. We tried to use SWAP from PHP, using SOAP, and tried to access a Zimbra collaboration suite server, using Python and SOAP.

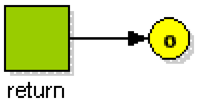
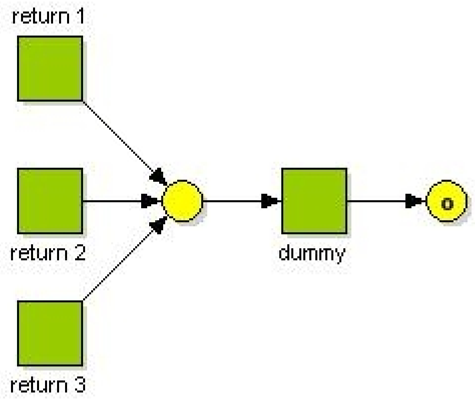
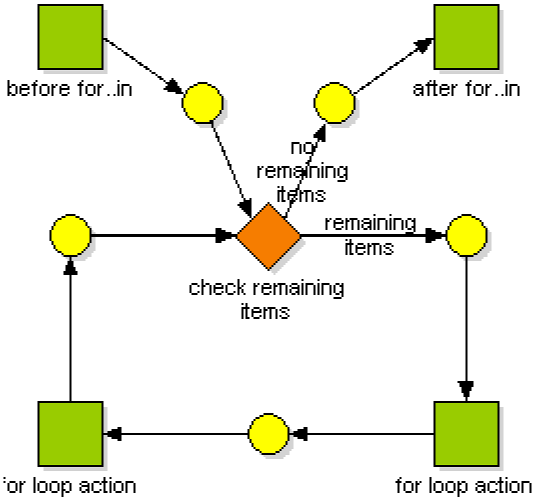
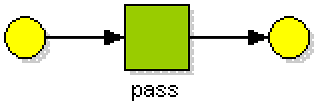
# *Appendices.*

## A1. Mapping source code to Petri nets

The table hereunder, describes how we mapped typical Python statements to Petri net constructs. The set of proposed mappings define the “mknet” conversion. The reasoning behind these mappings can be found appendix A2.

| *Python language construct* | *Petri net equivalent* | *Graphical notation* |
|---|---|---|
| variable assignment / single modification | a single transition |  |
| if | decision diamond with 2 outputs; 1 output will connect to a transition that will be fired if the condition for the if statement holds, the other output of the diamond will connect to the same place the transition that is fired if the if statement holds puts out to; this implies that only a transition is fired if the conditional expression holds |  |
| if .. else | decision diamond with 2 outputs; each output will connect to a (different) transition; one transition is fired is the conditional statement holds; the other is fired if the conditional statement does not hold; both the transitions connect to the same output place |  |

| *Python language construct* | *Petri net equivalent* | *Graphical notation* |
|---|---|---|
| if .. elif .. else | this is basically an if statement with an if..else statement as the “transition” to be fired if the *first* if statement does *not* hold; this will be modeled as a decision diamond with 2 outgoing arcs; if the first conditional statement holds, the code in the body of that if statement will be executed; the other outgoing arc is connected to another decision diamond, the elif diamond, which is a regular if-else decision diamond with two outgoing arcs; all transitions (of both the if- and the elif transitions) have an outgoing arc to one single output place |  if<br>false<br>true<br>elif<br>true<br>false<br>true<br>if body<br>else body<br>if body |
| try .. except | This is basically an if statement, with as main difference that the branching occurs *after* the program fragment in the try-clause has been executed |  try .. except<br>try clause + program fragment in try clause<br>try succeeds<br>try fails<br>except code |

| *Python language construct* | *Petri net equivalent* | *Graphical notation* |
|---|---|---|
| return | Stops the current program (or function) and returns to the calling function; this is modeled as a single transition that is connected to a final place; if the net features multiple return calls (Python allows this), all return calls connect to a single transition (a dummy), as Yasper doesn't allow final places to have more than a single input |  <br>or:<br> |
| for .. in | For each item of a collection of items (Python array, list or dictionary type), some action is performed. After each, and before the first, iteration, a check on remaining items is performed. This is modeled as a XOR transition that either fires an action, which returns to the XOR transition, or it breaks the loop. |  before for..in<br>after for..in<br>no remaining items<br>remaining items<br>check remaining items<br>or loop action<br>for loop action |
| pass | Python statement that doesn't do anything at all; usually used to execute the (empty) body of a loop; this can be modeled by either “nothing” or a single transition (which consumes a token and produces the same token again); we choose to model pass as such an empty transition |  |

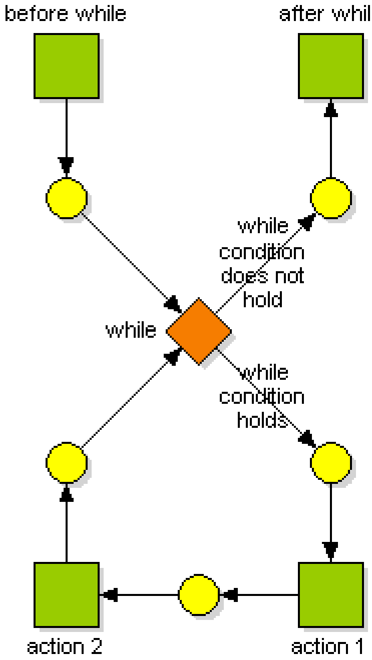
| *Python language construct* | *Petri net equivalent* | *Graphical notation* |
|---|---|---|
| while | decision diamond with 2 ingoing and 2 outgoing arcs; one ingoing arc is used to initiate the while loop; the other is used to reinitiate the while loop after execution of the body; one outgoing arc is connected to a subnet that will be initiated if the condition for the while loop holds; the other outgoing arc is connected to a subnet that will be executed once the while loop is done |   before while<br>after while<br>while condition does not hold<br>while<br>while condition holds<br>action 2<br>action 1 |

*Table 5: Source code to Petri net mappings*

## A2. The reasoning behind program code to Petri net conversions

### The if statement

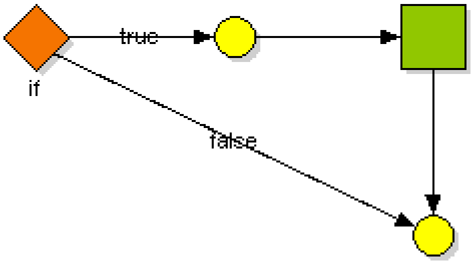


*Illustration 11: IF-statement*

We distinguish two situations:

1. the guard of the if condition holds
   - ➔ the outgoing arc for the “true” condition is chosen by the XOR statement
   - ➔ the code in the body of the if statement, represented by a transition, is executed

     *note:* the body of the if statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition
   - ➔ after the execution of the if body, the if-net puts a token in its final place
2. the guard of the if condition does not hold
   - ➔ the outgoing arc for the “false” condition is chosen by the XOR statement
   - ➔ there is nothing to be executed so the if-net puts a token in its final place

### The if .. else statement

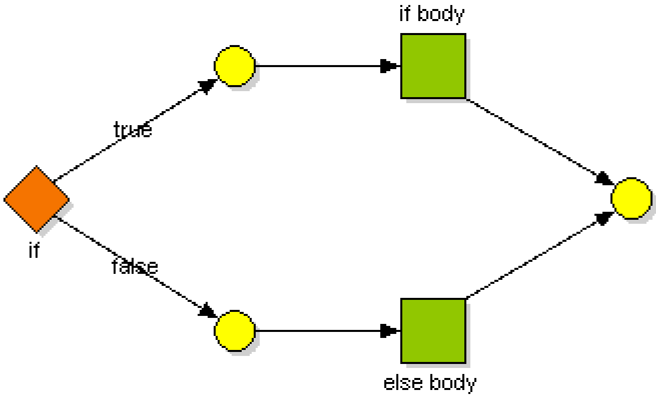


*Illustration 12: If ... else statement*

We distinguish two situations:

1. the guard of the if condition holds
   - ➔ the outgoing arc for the “true” condition is chosen by the XOR statement
   - ➔ the code in the body of the if statement, represented by a transition, is executed

     *note:* the body of the if statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition
   - ➔ after the execution of the if body, the if-net puts a token in its final place
2. the guard of the if condition does not hold
   - ➔ the outgoing arc for the “false” condition is chosen by the XOR statement
   - ➔ the code in the body of the else statement, represented by a transition, is executed

     *note:* the body of the else statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition
   - ➔ after the execution of the else body, the if-net puts a token in its final place

### The if .. elif .. else statement

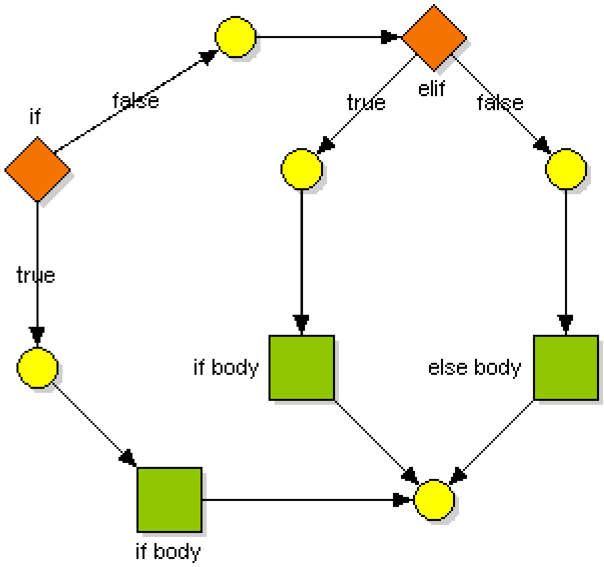


*Illustration 13: If .. elif .. else stament*

We distinguish three situations:

1. the guard of the if condition holds
   - ➔ the outgoing arc for the “true” condition is chosen by the XOR statement
   - ➔ the code in the body of the if statement, represented by a transition, is executed

     *note:* the body of the if statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition
   - ➔ after the execution of the if body, the if-net puts a token in its final place
2. the guard of the if condition does not hold, but the guard of the elif statement holds
   - ➔ the outgoing arc for the if statement's “false” condition is chosen by the XOR statement, the elif statement can fire now
   - ➔ the outgoing arc for the elif statement's “true” condition is chosen by the XOR statement
   - ➔ the code in the body of the elif statement, for the “true” condition”, represented by a transition, is executed

     *note:* the body of this statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition
   - ➔ after the execution of the elif (true) body, the if-net puts a token in its final place
3. the guard of the if condition does not hold and neither does the guard of the elif statement
   - ➔ the outgoing arc for the elif statement's “false” condition is chosen by the XOR statement

- ➔ the code in the body of the elif statement, for the “false” condition”, represented by a transition, is executed

  *note:* the body of this statement can be a program too; in that case, we might want to use a subnet (or its definition) instead of a single transition

- ➔ after the execution of the elif (final) body, the if-net puts a token in its final place

### The try .. except statement

The proof for the this statement is a bit tricky. The problem lies in how the source code is written and how this source code is executed. This code snippet provides a little example:

```
try
        <some program code here>
except
        <in case something goes wrong>
```

The try clause, including its body, is always completely executed. If the code in the body of the try clause fails, the except clause is initiated. This construct can cause the program to jump; in case something in the body of the try clause goes wrong, the program jumps to the body of the except statement; if not, the program jumps to the code after the entire try/except block.

After each statement in the body of the try clause, it's possible to jump to the except code. That would mean that for every clause in this code, we would have to model a jump to the body of the except clause.

We have some luck here; in the SWAP program code, the body of all try and all except statements are rather short and plain (no branching by if statements etc there). This implies that these blocks of codes can safely be replaced by a single transition, which makes modeling the try .. except clause easier.

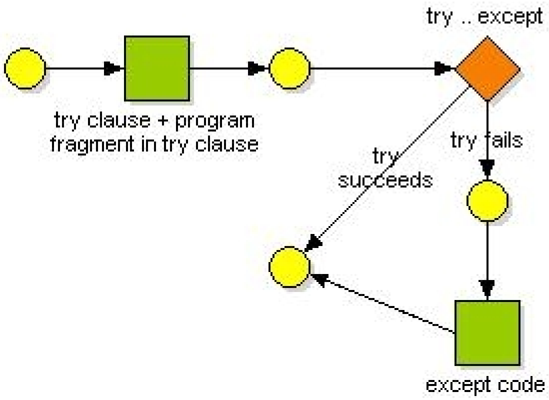


*Illustration 14: Try .. except statement*

So, we distinguish two situations:

1. The try clause is initiated; the code in the body of the try clause is executed and the code in the body of the try statement successfully terminates

- ➔ After the last of the code in the body of the try statement is executed, a token is put into the final place of the try/except net

2. The try clause is initiated; the code in the body of the try clause is executed and the code in the body of the try statement fails at some point
   - ➔ After the failing line of code (or, in our case, after the transition representing the body), the program jumps to the except clause
   - ➔ The body of the except cause is executed (usually, this body is used for generating error messages)
   - ➔ After the last of the code in the body of the except statement is executed, a token is put into the final place of the try/except net

## Return statement

The return statement is mandatory for all functions in Python; functions are allowed to have more than a single return statement.

In case a function has a single return statement, we have the following net:

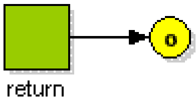


*Illustration 15: Single outpoint*

This is modeled as a single transition; it's nothing more than a single transition.

However, in case a function has more than one single return statement, we have to be a bit creative:

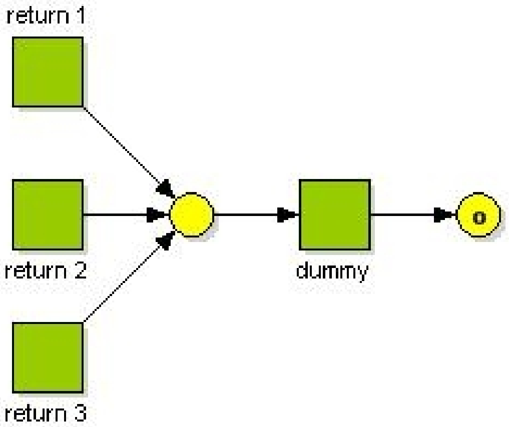


*Illustration 16: Return statement - multiple returns*

Yasper does not allow (sub)nets with more than one transition to be connected to a final place (of a subnet), so we have to come up with some solution. Our solution is to make use of a dummy transition, which works like this:

- presume we have 3 return statements (which of course are modeled as transitions): r1, r2 and r3
- r1, r2 and r3 share an output place connected as input place to a dummy transition
- the dummy transition outputs to the final place of the subnet

As the name suggests, the dummy transition does not do a thing, but collecting a token from one of the return statements and putting it out to the final place of the subnet. Which return statement is chosen depends on how the program branches.

### The for .. in statement

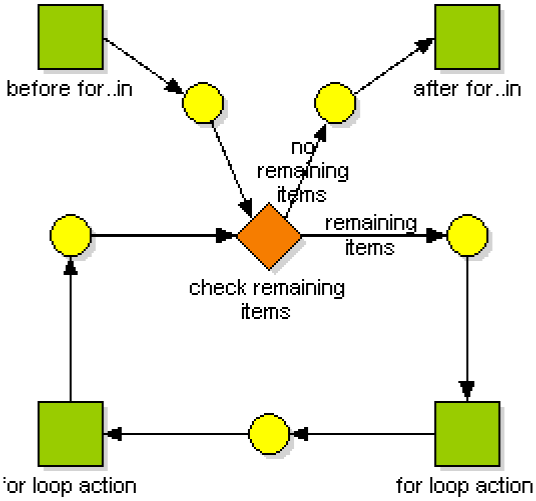


*Illustration 17: For .. in statement*

This Python construct allows program code to be executed for a series of items. After entering the net, we distinguish two situations:

1. There are remaining items in the series
   - ➔ The body of the for .. in statement is initiated
   - ➔ After the code in the body has been executed, we return (again) to the check for remaining items
2. There are no remaining items in the series
   - ➔ A token is put in the final place of the for .. in-net

Each time, when checking for remaining items (that is, upon initiation of the for .. in loop, or, after an iteration of the loop), Python shrinks the list of remaining items by one item. This implies that this loop will always end (even if there is no data provided). We do not model this "shrinking" explicitly. As our testing methods are non deterministic testers, we know that during the testing of this, this loop will finish terminate (and that all both cases will be tested).

### The pass statement

This statement does not do anything; it is used to fill empty bodies, because empty bodies might be disallowed by certain implementations. This can be modeled as a single transition, (a dummy) or just be left out, or as single transition.

### The while statement

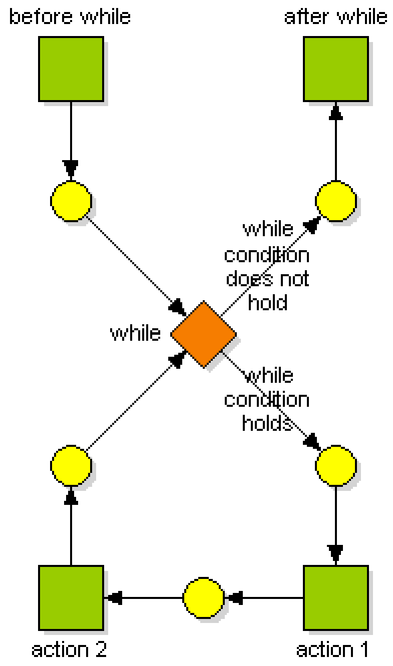


*Illustration 18: While statement*

This statement is almost identical to the for .. in statement, with one exception: the for .. in statement is executed for a series of items, while the while loop is executed as long as certain condition holds. After initiating the while loop, we distinguish 2 situations:

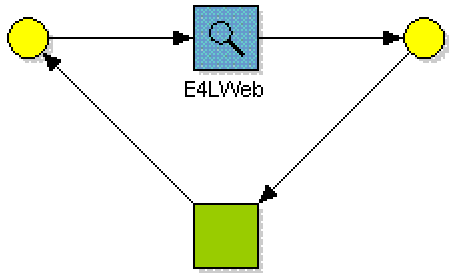
1. The while condition holds
   1. The body of the while loop is initiated
   2. After the body of the while loop has been executed, the loop is reinitiated
2. The while condition does not hold
   1. A token is put in the final place of the while-net

Just as with the for .. in statement, the while statement is tested non deterministically. The same guarantees apply here.

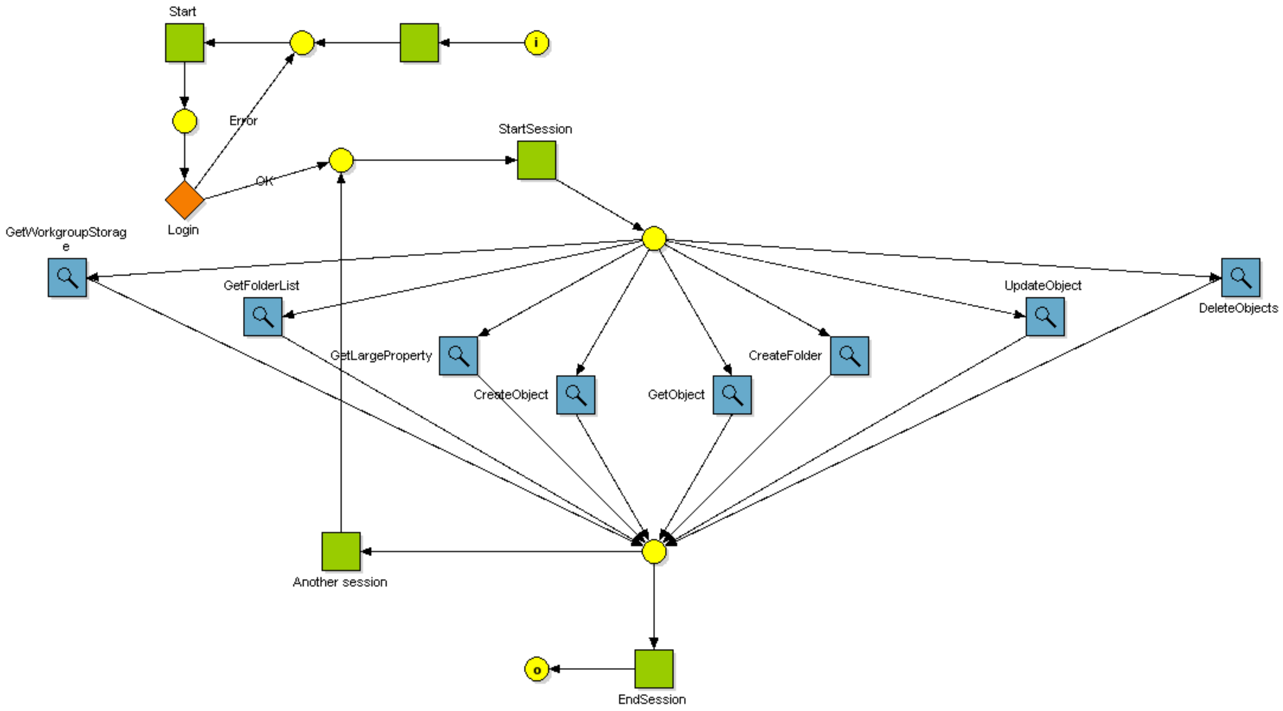
# A3. SWAP server Petri nets

## *1. Source code models*

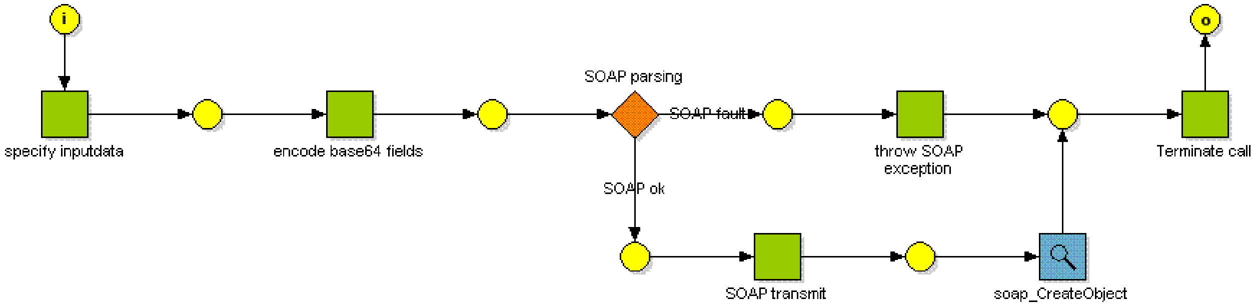
Level 1: system overview



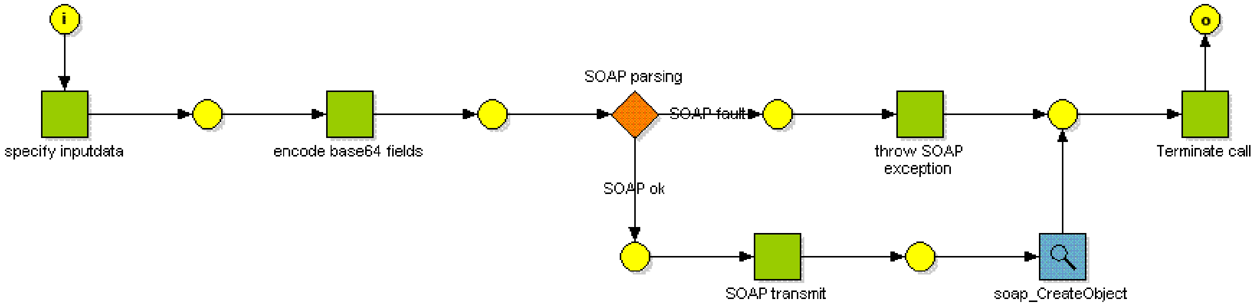
Level 2: SWAP session view



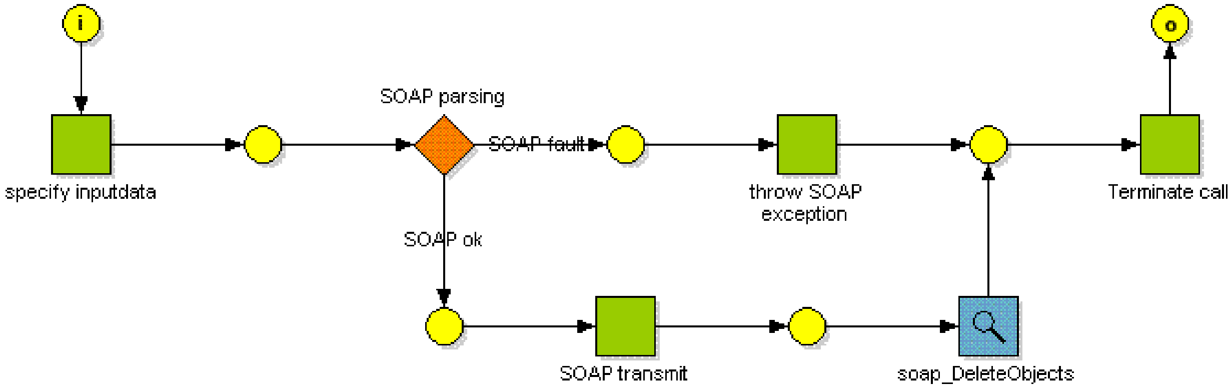
## Level 3: CreateFolder



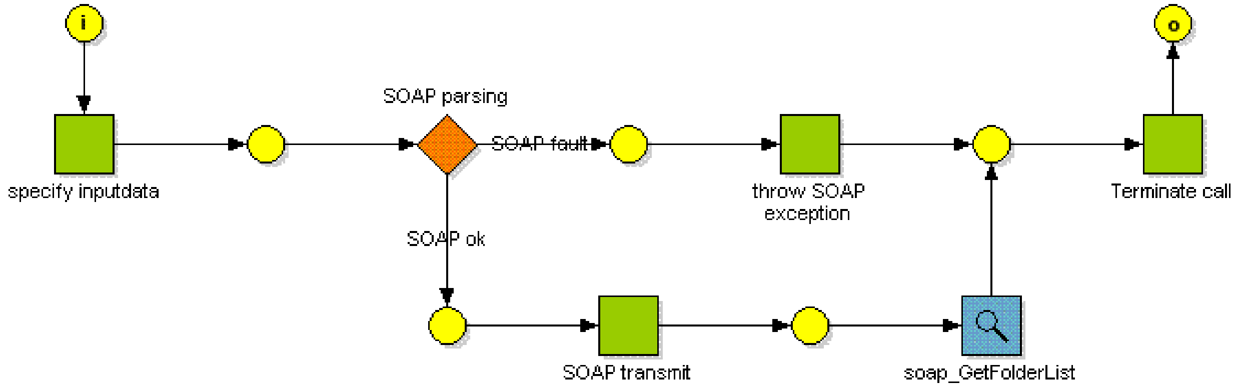
## Level 3: CreateObject



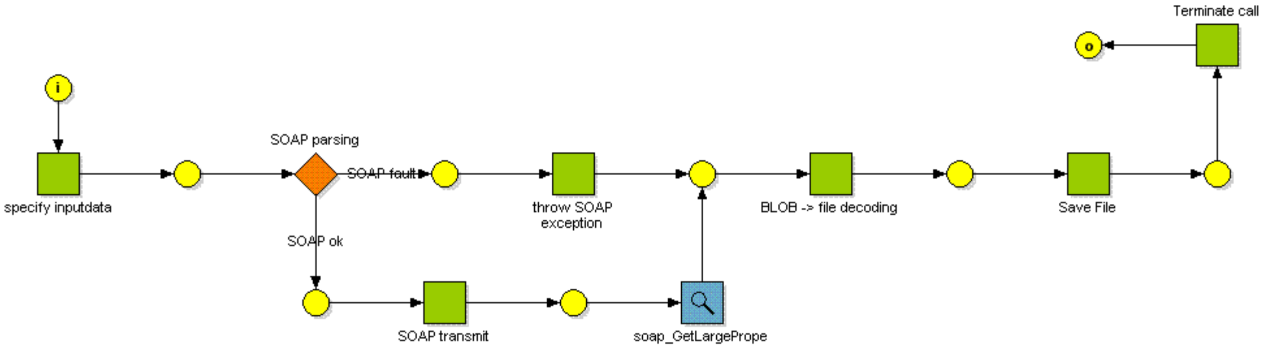
## Level 3: DeleteObjects



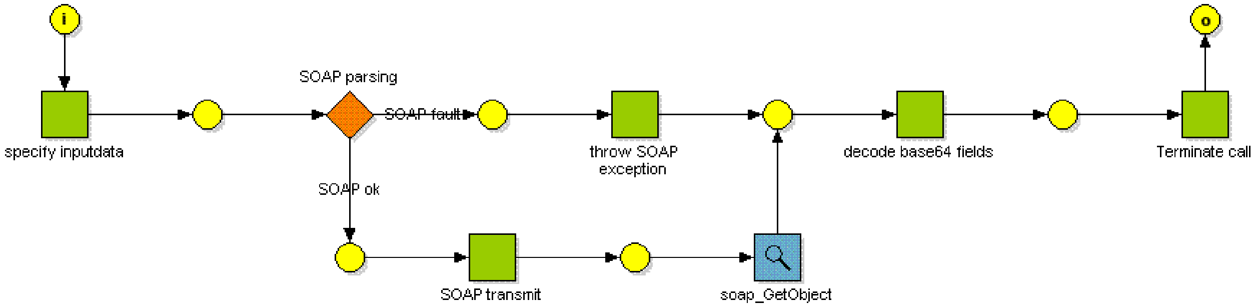
## Level 3: GetFolderList



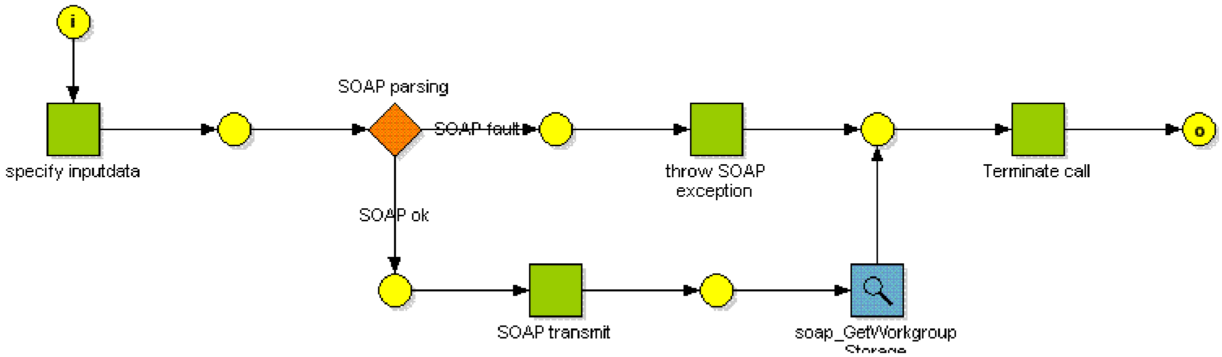
## Level 3: GetLargeProperty



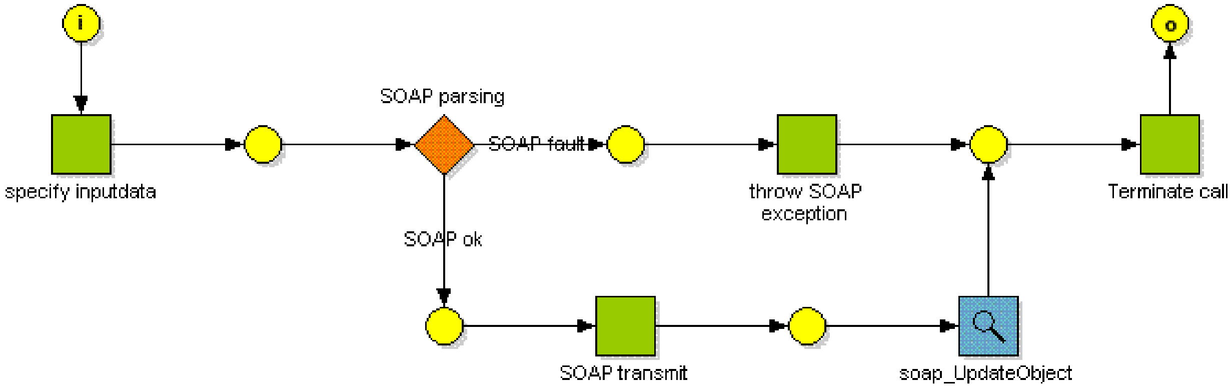
## Level 3: GetObject



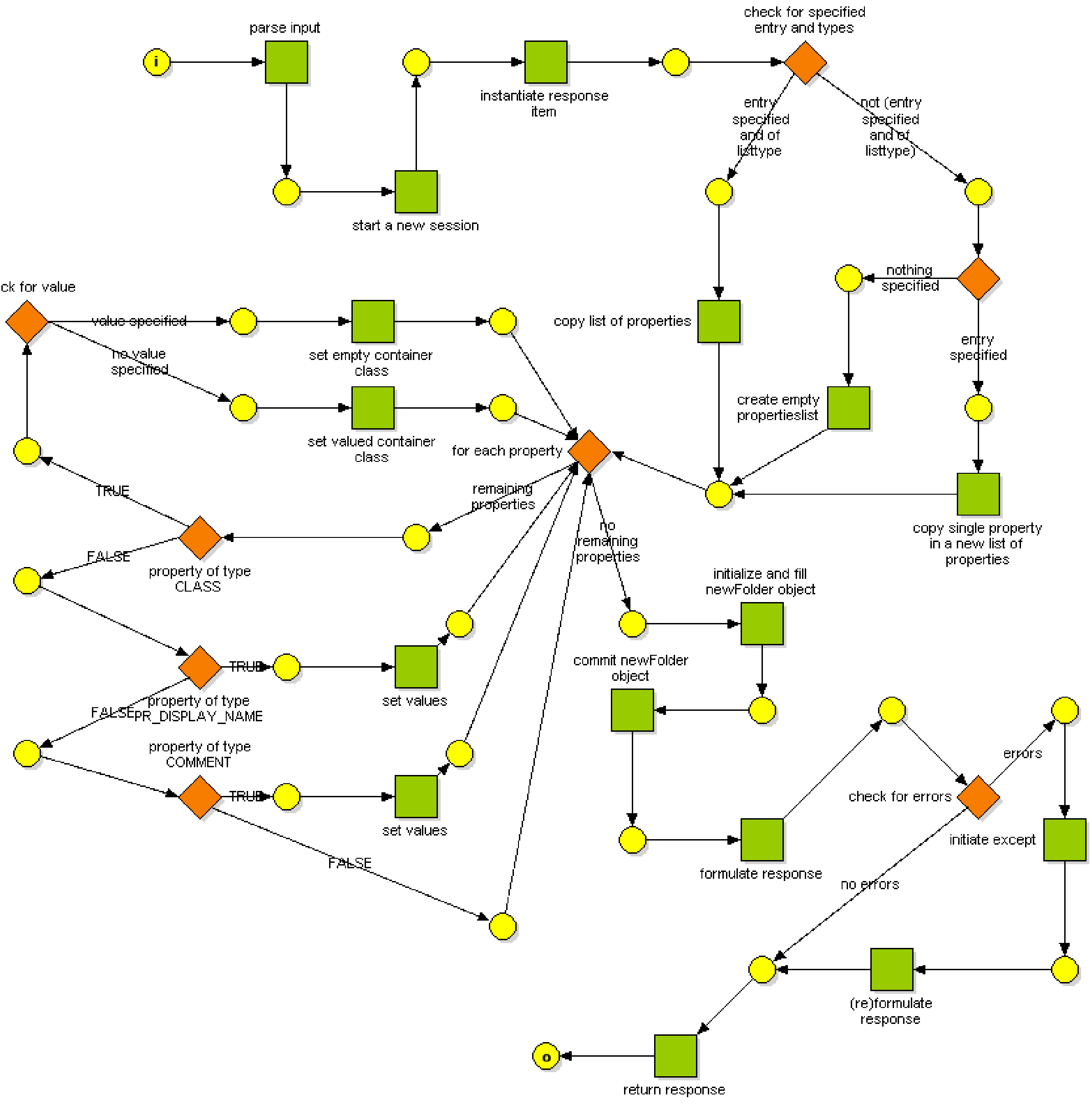
## Level 3: GetWorkgroupStorage



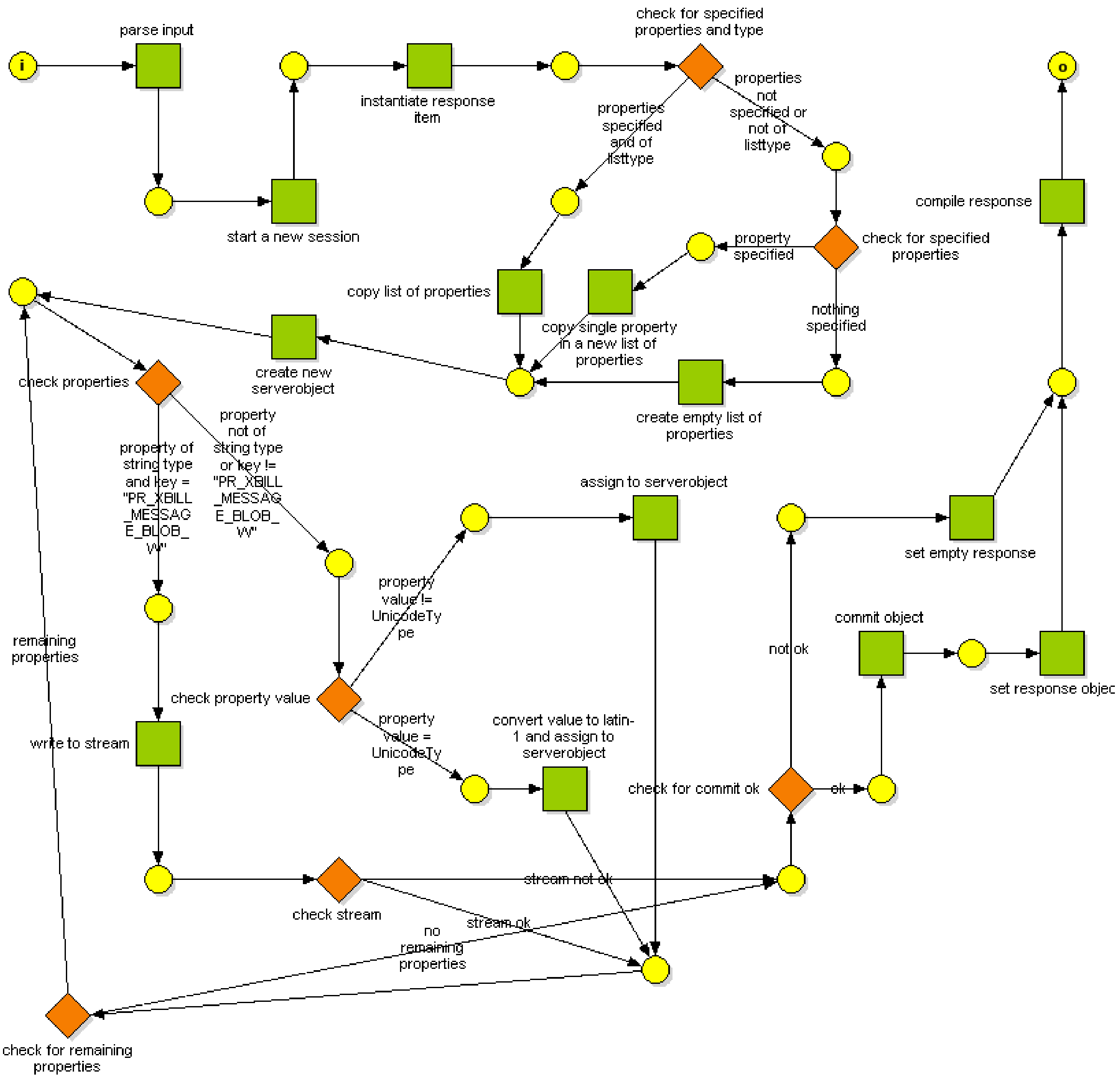
## Level 3: UpdateObject



Level 4: soap_CreateFolder

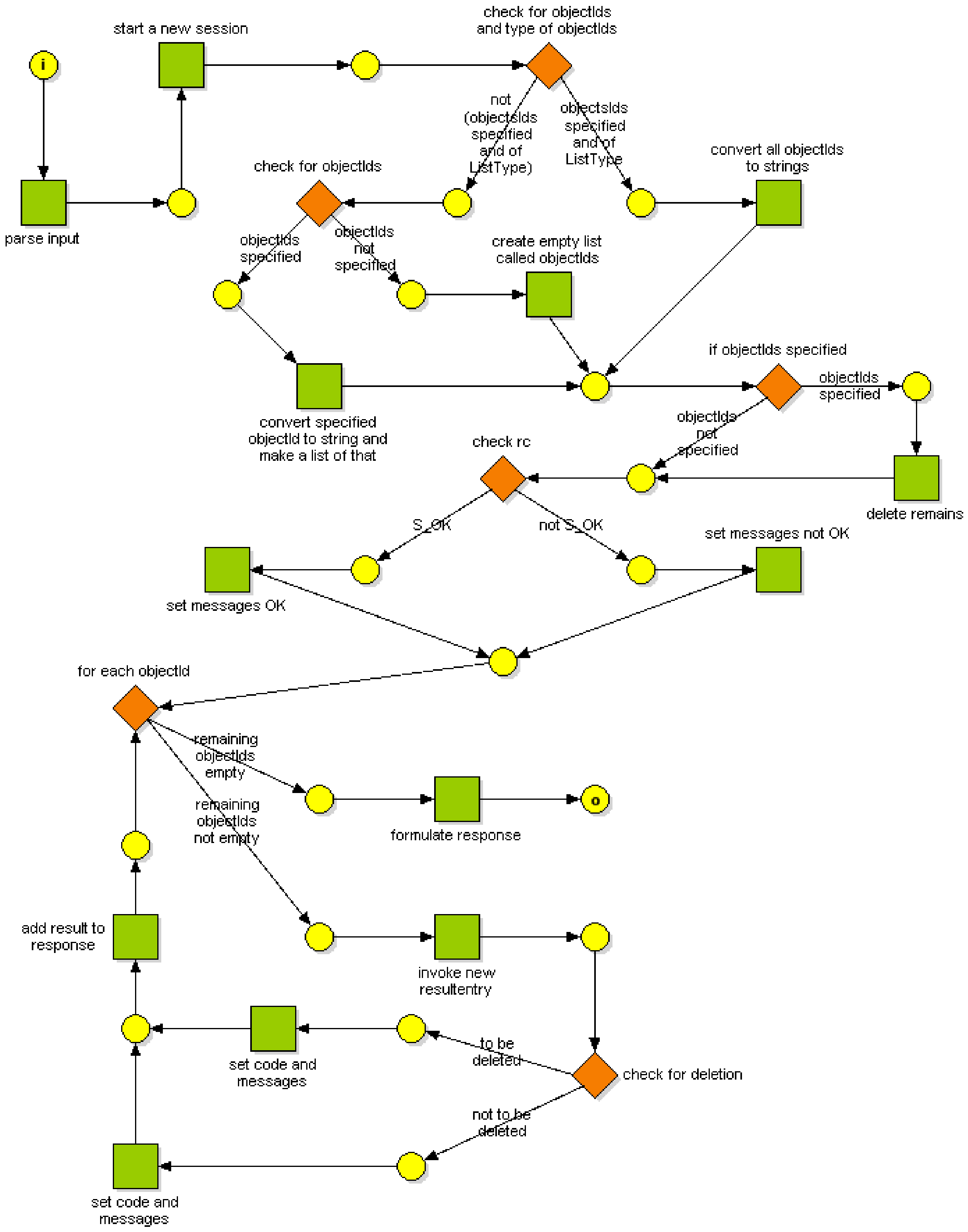


## Level 4: soap_CreateObject

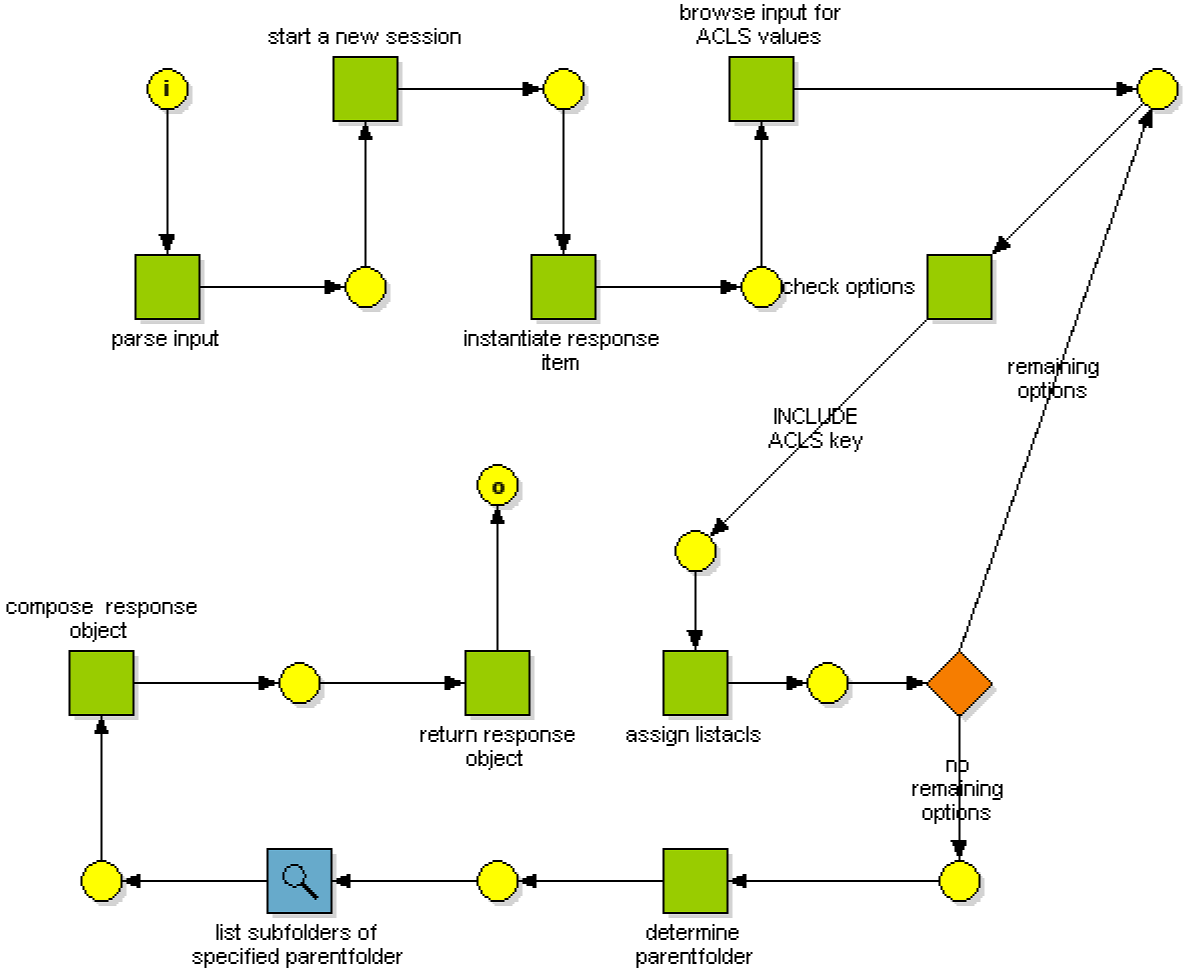


i
parse input
start a new session
instantiate response item
check for specified properties and type
properties specified and of listtype
properties not specified or not of listtype
o
compile response
check for specified properties
property specified
copy list of properties
copy single property in a new list of properties
nothing specified
create new serverobject
create empty list of properties
check properties
property of string type and key = "PR_XBILL _MESSAG E_BLOB_ W"
property not of string type or key != "PR_XBILL _MESSAG E_BLOB_ W"
assign to serverobject
set empty response
property value != UnicodeTy pe
remaining properties
commit object
not ok
set response objec
check property value
write to stream
property value = UnicodeTy pe
convert value to latin-1 and assign to serverobject
check for commit ok
ok
stream not ok
check stream
stream ok
no remaining properties
check for remaining properties

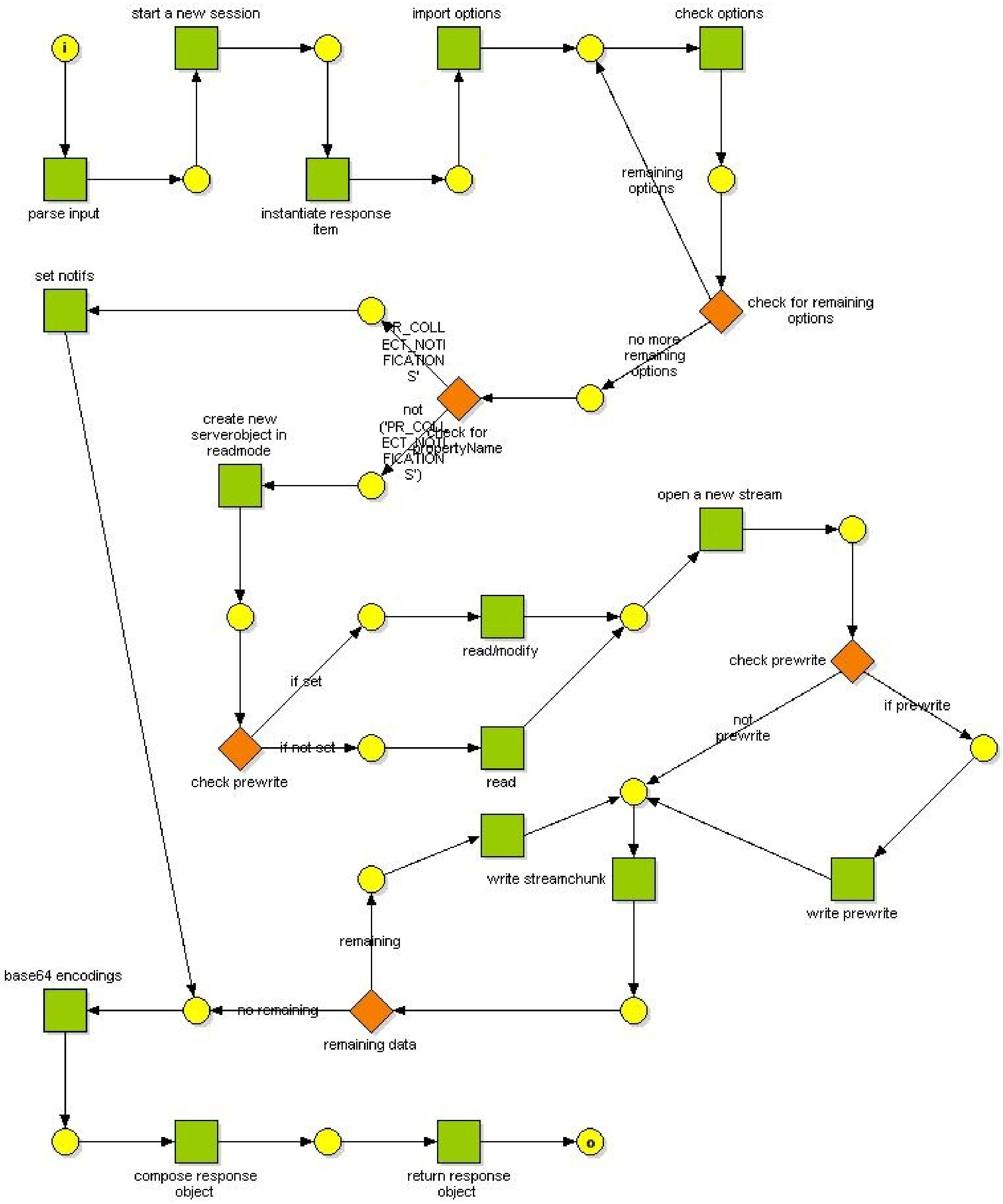
## Level 4: soap_DeleteObjects



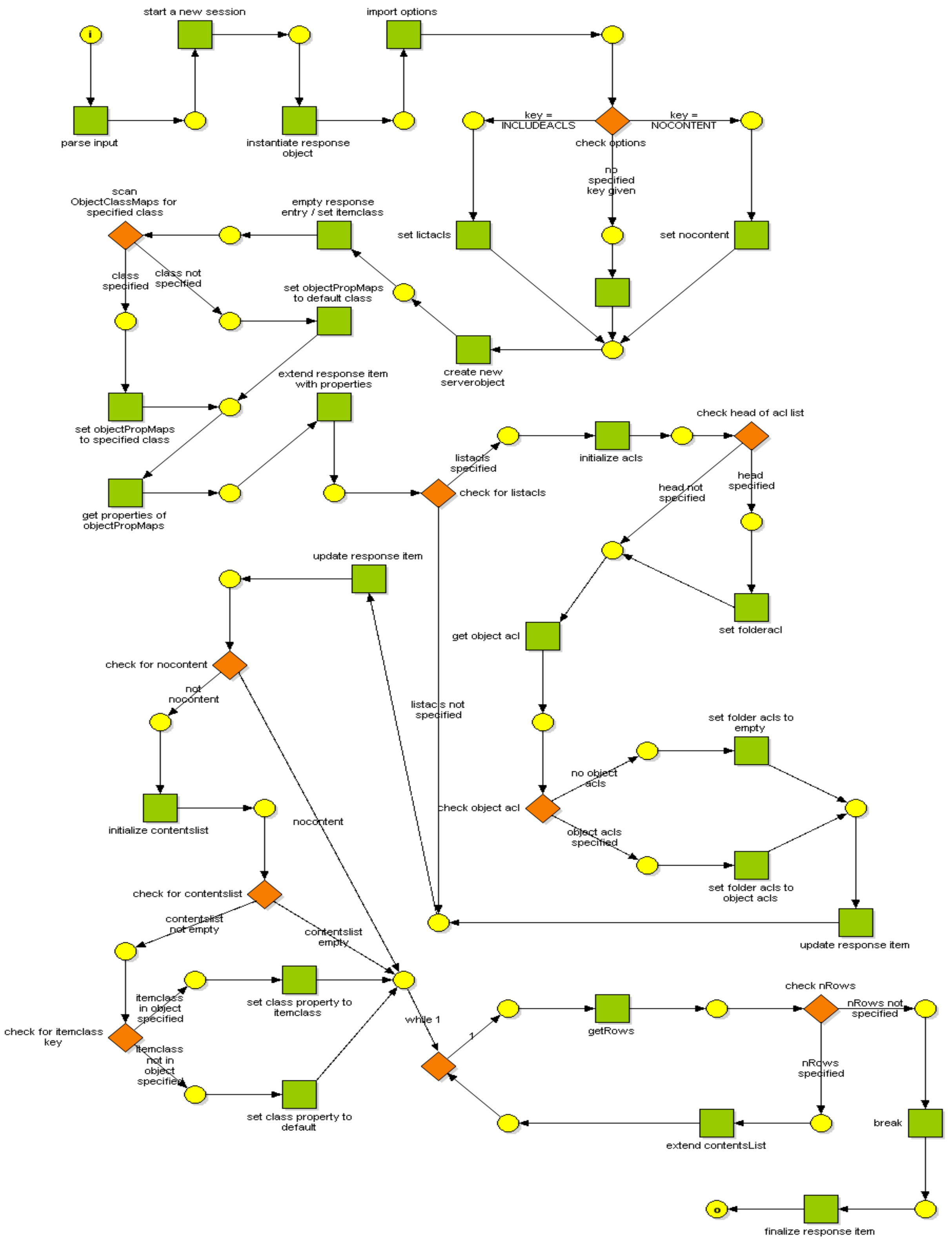
Level 4: soap_GetFolderList



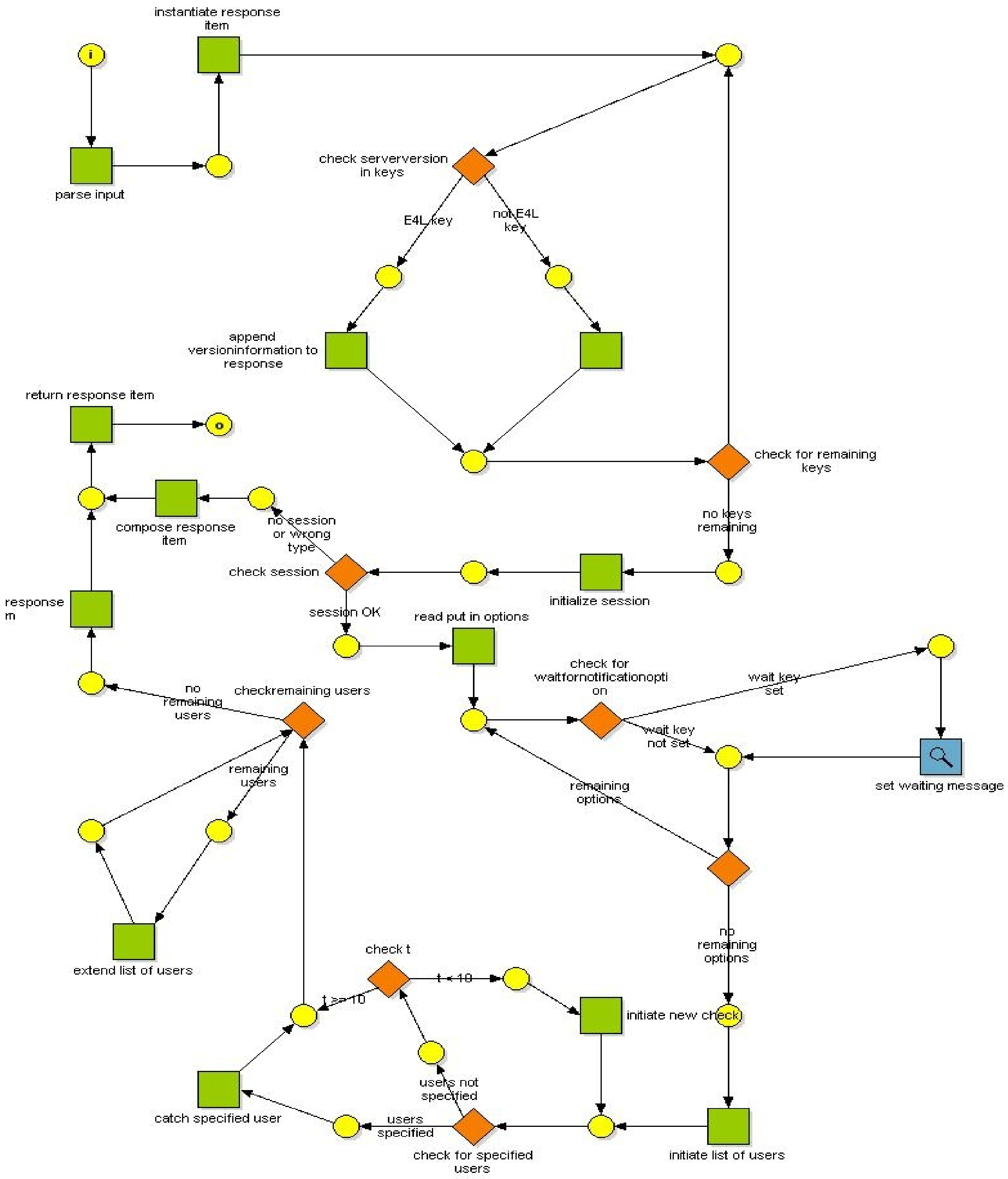
## Level 4: soap_GetLargeProperty



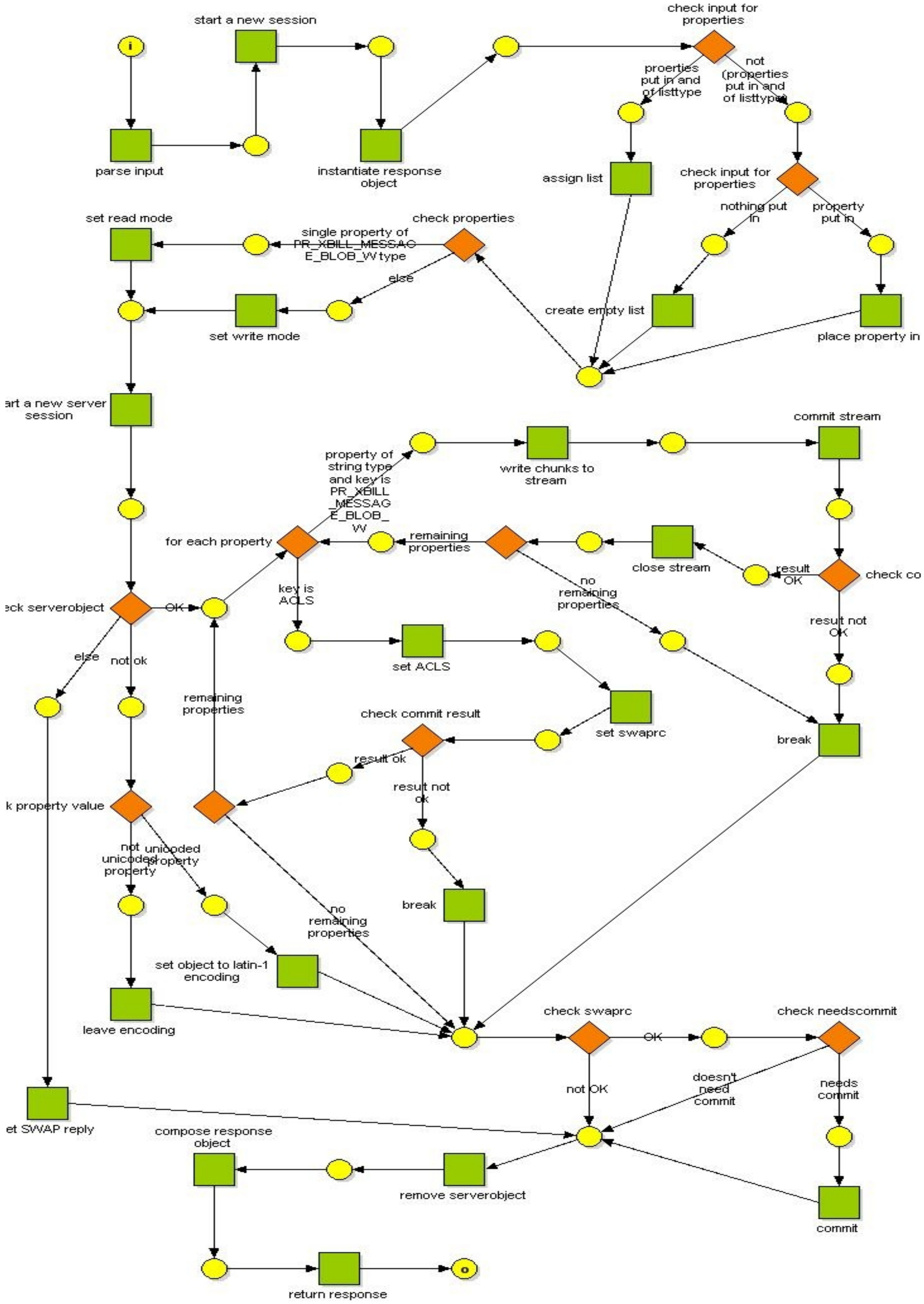
## Level 4: soap_GetObject



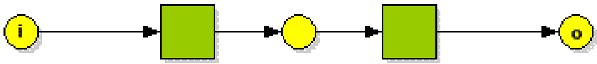
Level 4: soap_GetWorkgroupStorage



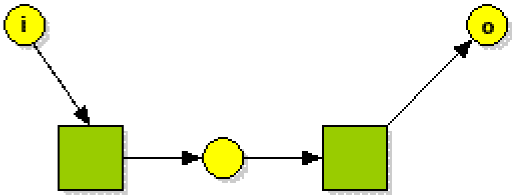
Level 4: UpdateObject



Level 5: list subfolders



Level 5: set waiting message

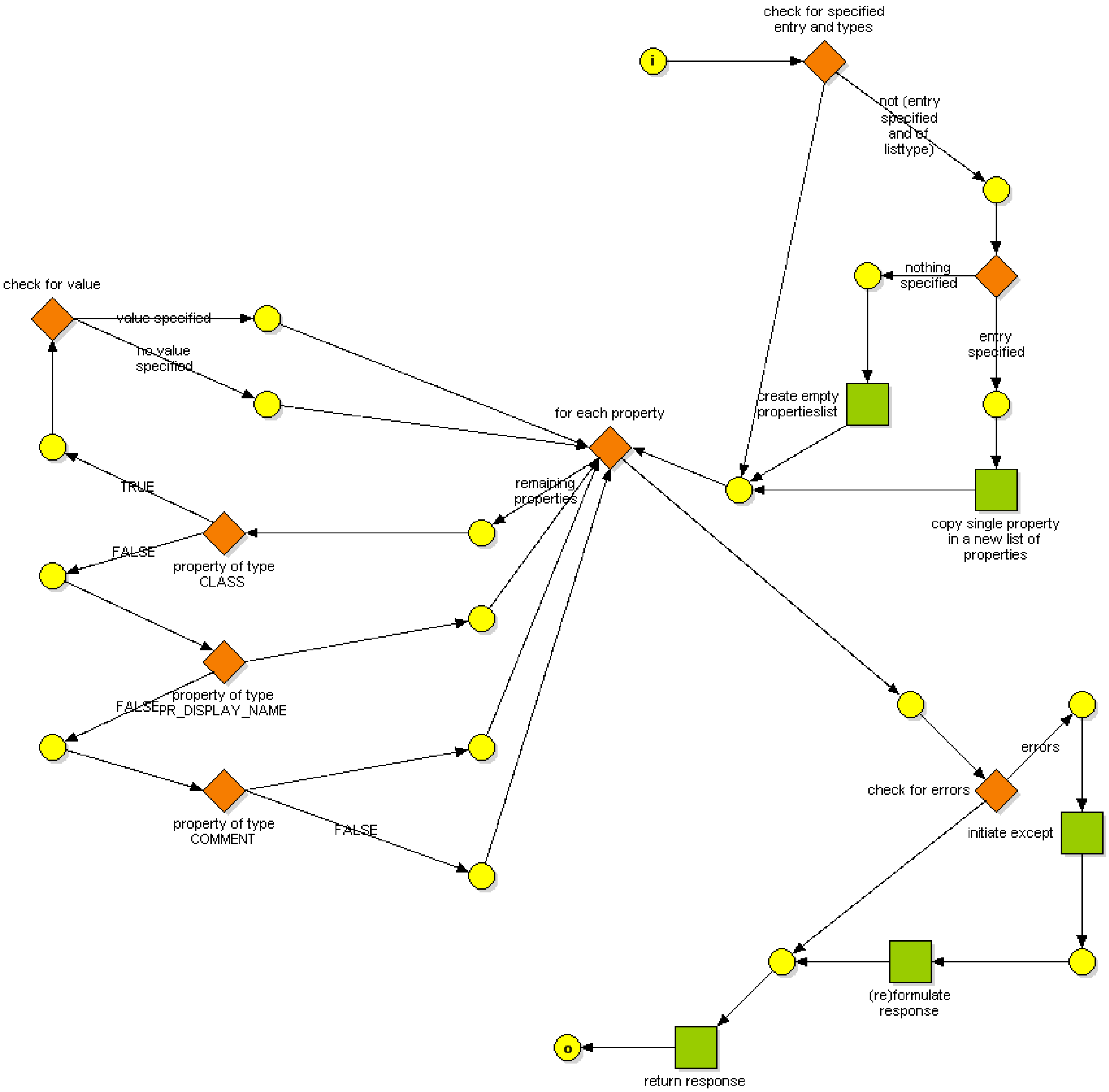


The level 5 models are “fake” models. The given Petri nets are sound workflow nets that do not do anything. This is because the level 5 calls are actually components of the Exchange4Linux server backend that are used by SWAP. Their verification is not in the scope of this research and thus they'll be considered flawless (hence the sound workflow nets).

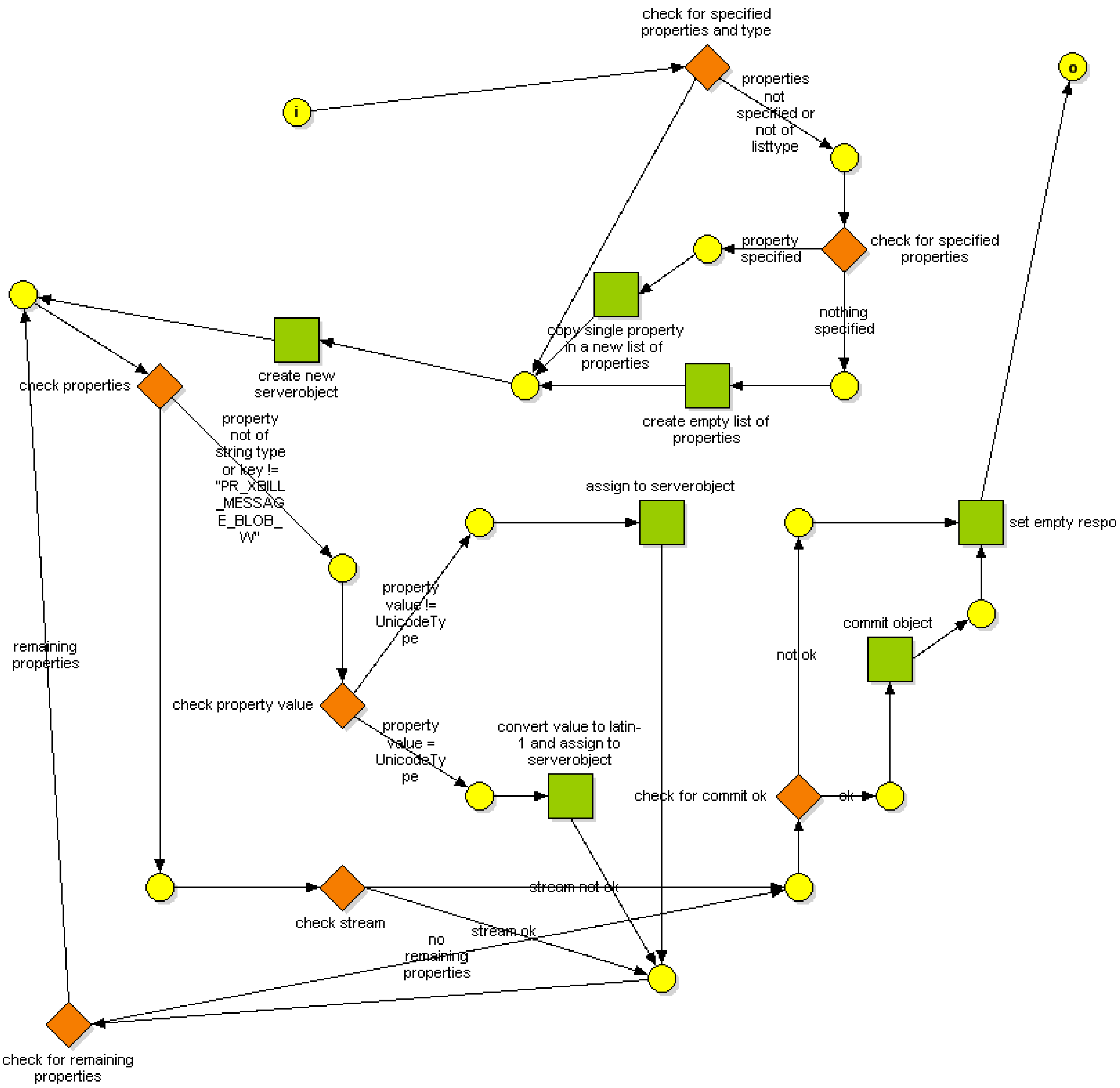
## *2. Filtered model*

Not all subnets will or can be reduced. Only those we reduced will be mentioned here; the ones that can not be reduced are equivalent to those we distilled from the source code.

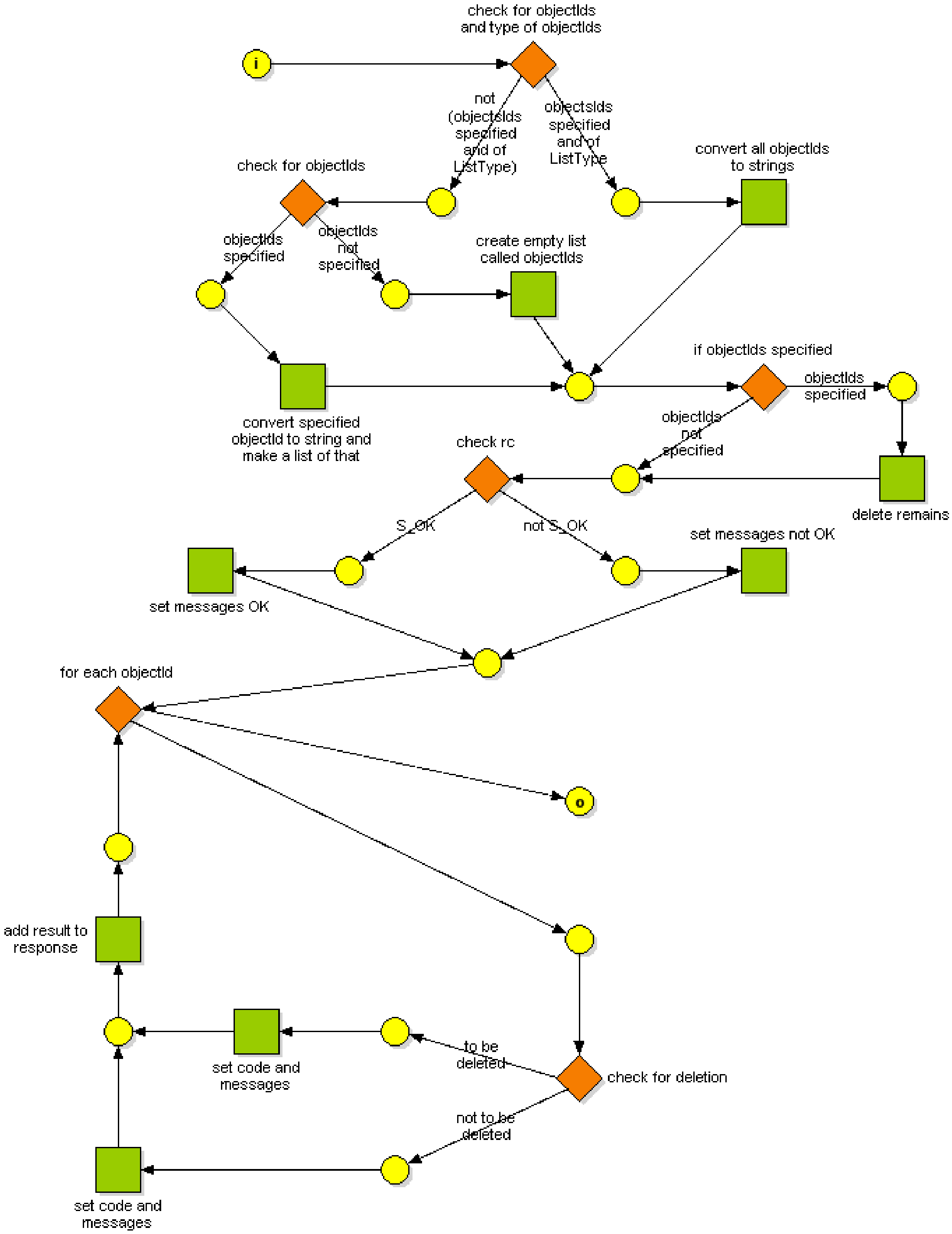
Level 4: soap_CreateFolder


check for specified entry and types
i
not (entry specified and of listtype)
nothing specified
entry specified
create empty propertieslist
copy single property in a new list of properties
check for value
value specified
no value specified
for each property
remaining properties
TRUE
FALSE
property of type CLASS
FALSE
property of type PR_DISPLAY_NAME
property of type COMMENT
FALSE
check for errors
errors
initiate except
(re)formulate response
return response
o

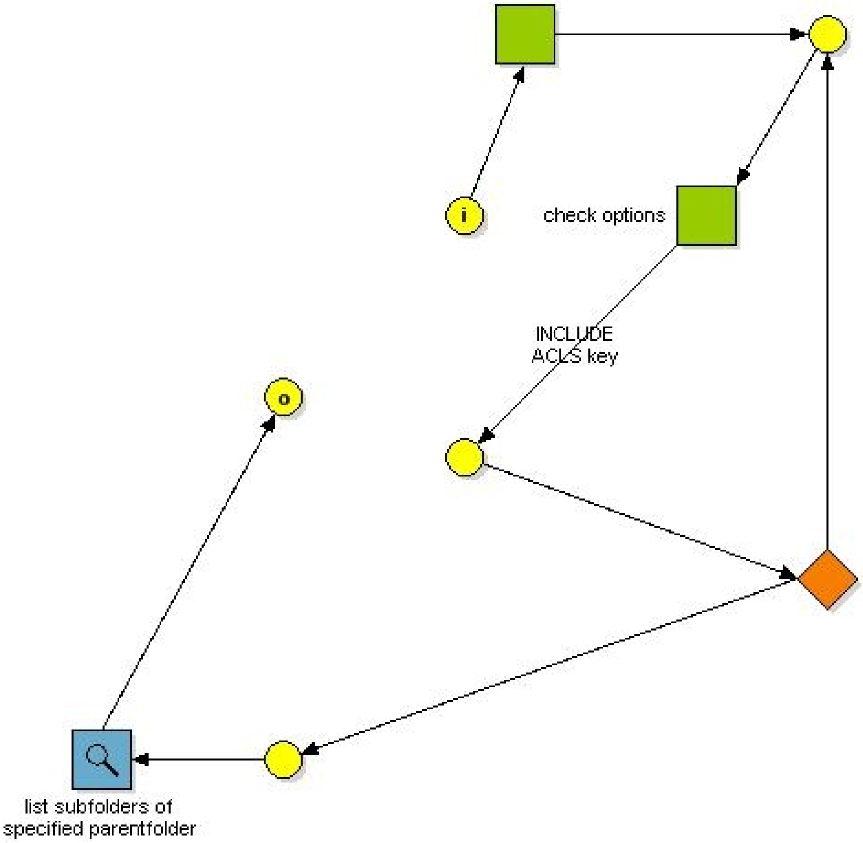
## Level 4: soap_CreateObject


check for specified properties and type
o
i
properties not specified or not of listtype
property specified
check for specified properties
copy single property in a new list of properties
nothing specified
check properties
create new serverobject
create empty list of properties
property not of string type or key != "PR_XBILL _MESSAG E_BLOB_ W"
assign to serverobject
set empty respo
property value != UnicodeTy pe
commit object
remaining properties
not ok
check property value
property value = UnicodeTy pe
convert value to latin-1 and assign to serverobject
check for commit ok
ok
stream not ok
check stream
stream ok
no remaining properties
check for remaining properties

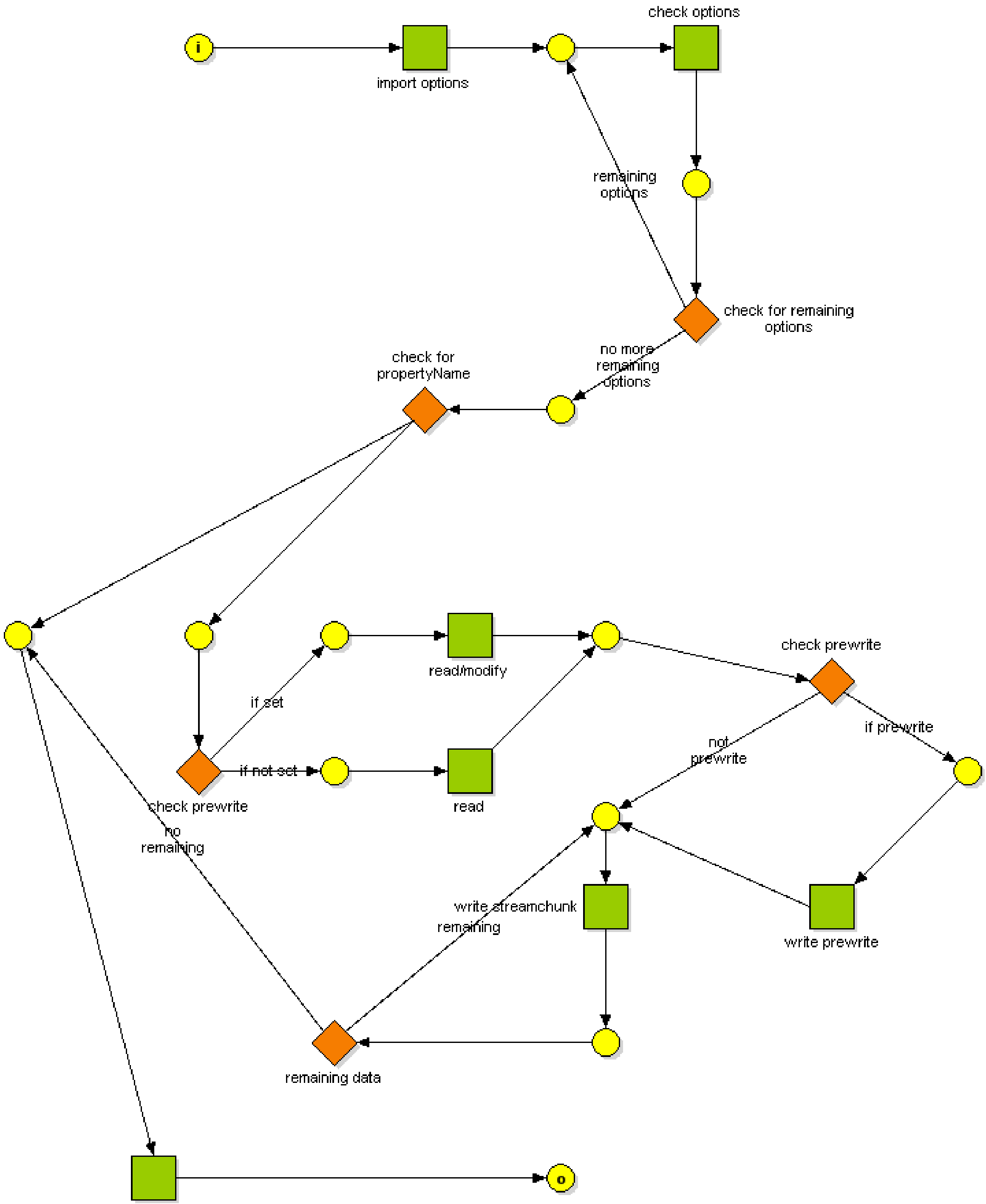
Level 4: soap_DeleteObjects



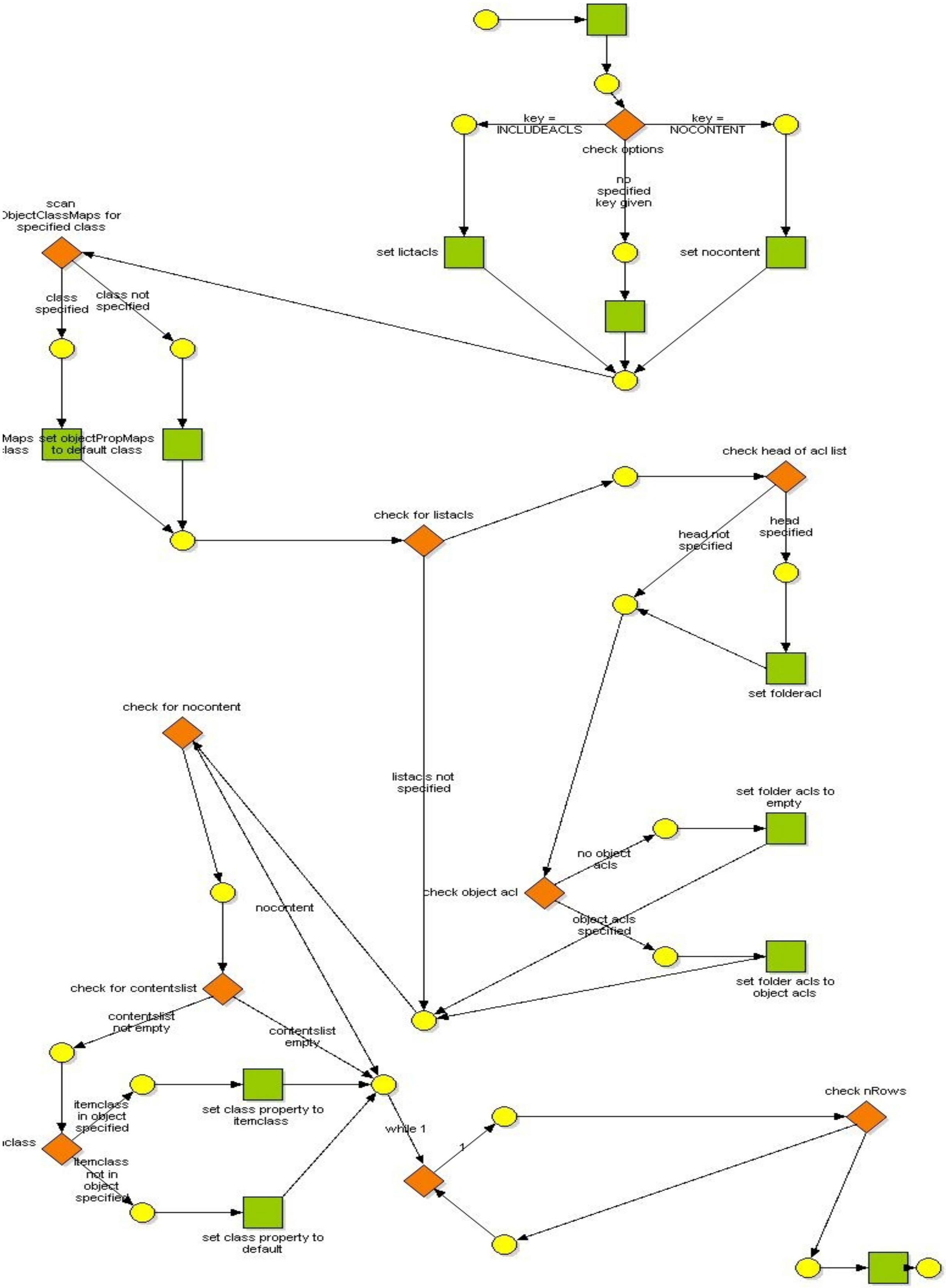
Level 4: soap_GetfolderList



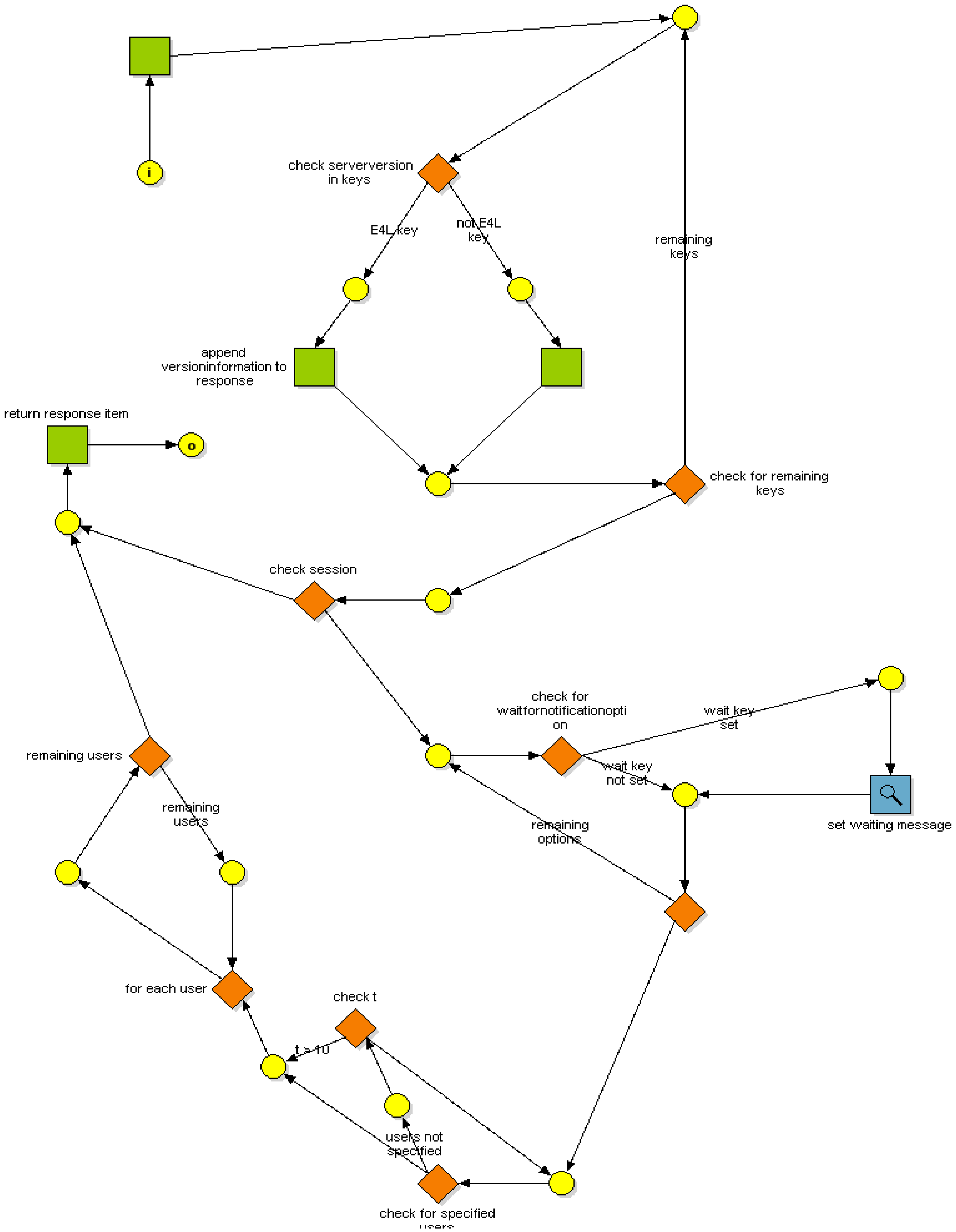
Level 4: soap_GetLargeProperty



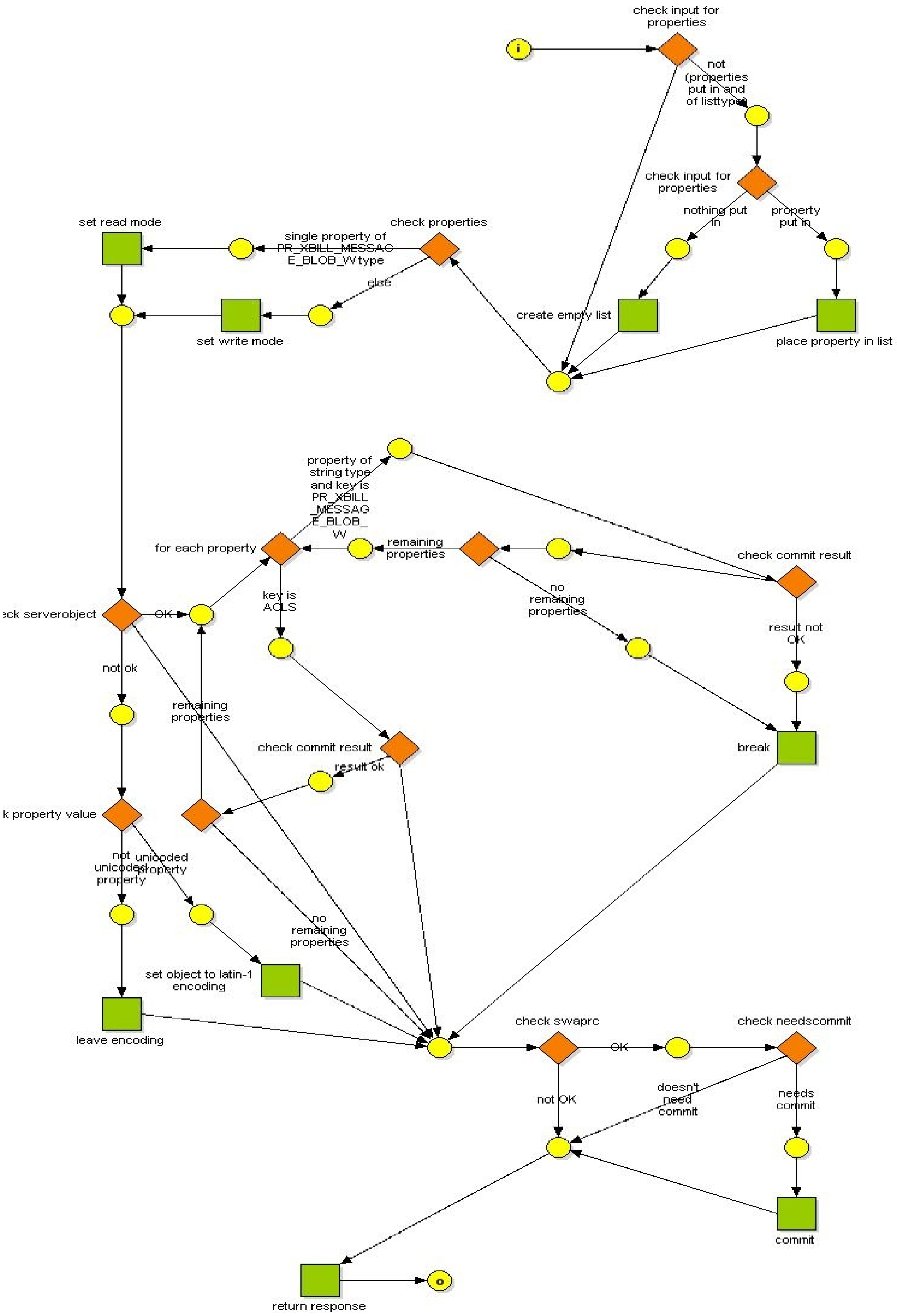
Level 4: soap_GetObject



## Level 4: soap_GetWorkgroupStorage



Level 4: soap_UpdateObject



## A4. WofLan testing issues

After reducing the to be tested Petri nets, we converted these nets into a flat net. After doing so, we did a conversion to the WofLan TPN format. A problem occurred; WofLan crashed upon opening the generated TPN file. The size of the input file seems to be responsible for this issue.

In order to circumvent this problem we chose to split up the input file in multiple parts. To be exact, we made for each subnet, a separate Yasper diagram. After converting these separate Yasper files into classical Petri nets (as there is no hierarchy in these nets, as we use nets of just one level, we only have to do a XOR2AND conversion), we generated TPN files from these nets. Some of these files were not readable by WofLan. It turned out that (converted) PNML files, with as first transition a XOR transition cannot be read by WofLan. By adding a dummy transition in front of the XOR transition (which has no effect on the soundness of the whole) this problem was circumvented. After reapplying the conversions, we acquired files readable by WofLan.

In order to make this way of testing possible, we used a bottom-up approach. We started testing on the lowest level (the lowest level nets, which are trivial SMWF nets, were not tested, as their soundness is trivial, as any SMWF net is always sound) and modeled each sound subnet as a single transition (which of course is sound too).

So, after doing a series of tests, we achieved some nice results.

## A5. Gain of the reduction process

By comparing the file size of the filtered and flattened model to the file size of the flattened non-filtered model, we can get an insight what we gained from the reductions we did. Of course, all descriptional information is erased from both files, before comparing.

The filtered version has a file size of  177740 bytes, the non-filtered version has a file size of 276396 bytes. This sums up to a reduction of (about) 36%. As, when using automated checking tools, the coverability graph grows in an exponential way, compared to the size of the Petri net (or its conversion), we may suspect a serious gain in this reduction.

## A6. Manual checking for deadlocks

In order to manually check for deadlocks, we made use of the unfiltered Petri nets and the program code. This was done by performing these steps:

1. In a Petri net, we identify sensitive points (which is a location where deadlocks related to data might occur) (branching point by either a XOR transition or conflicting places) in a Petri net

   *or,*
   in the SWAP source code, we  identify sensitive points which wait for an event to happen (that event may be the “arrival” of data)
2. Consider the scenarios that are likely / possible to happen, with relation to a sensitive point

3. Analyze, using the source code, how different scenarios are treated, using the Petri nets, how this “treatment” branches.

In case of problems, in the third step of the check described above, we have to find a solution to these issues. This step has to be performed carefully, as changing the program might introduce new errors. So, for every change proposed, we considered its effects on the rest of the program and did re-checks where necessary.

### *Results*

After considering the source code, we notice that there is only one construct requiring some attention. In the *soap_GetObject* call, there is a “while 1” loop. This loop will only terminate upon reaching the “else” clause of the embodied if-else statement. This else clause can only be reached when the environment *either* does not hand over any more data (the result of a “get” call is empty) *or* it keeps on submitting data on which the condition for X holds.

The problematic code:

```
while 1:
            <assignment to variable X, including a poll to the environment>
      if <condition on X>
            <some non relevant code>
      else
            break
```

If the conditions, mentioned above, for breaking the loop are not met, the program would end up in a deadlock situation.

An optimal solution would be to limit the number of times this loop is fired. In that case, we still could make use of this “while 1” construct. The code could look like this:

```
while 1:
      if (current =< limit)
                  <assignment to variable X, including a poll to the environment>
            if <condition on X>
                  <some non relevant code>
            else
                  break
            current += 1
      else
            break
```

Although this construct is safe, presuming a correct limit is provided, we still have a problem: how can we know what the limit is? Ideally, the environment would provide us with the count of items we would be receiving. This cannot be the case here however, as this would require more changes in the CORBA backend and (most) Exchange4Linux clients.

As we have insight in the number of rows that are provided by the environment, we can prespecify the proposed limit. Although that is a bit of a solution, it's not 100% safe either. The problem is that repeatedly polling the environment for data, while there is no data to be delivered, ends us up in undefined territory (which of course, is bad programming too).

A small patch solves this issue. When polling the environment for data, while there is no (new!) data to be delivered, there are two scenarios which can happen:

- the same data as before is delivered

– no data is delivered

So, by adding a check for these two scenarios, we have a safe solution to this issue. Our solution looks like this:

```
limit = <value for limit>
previous = “NoValue”
while 1:
      if (current =< limit)
                  <assignment to variable X, including a poll to the environment>
            if (!=X) or (X==previous)
                  <exception handle>
            if <condition on X>
                  previous = X
                  <some non relevant code>
            else
                  <exception handle>
            current += 1
      else
            <exception handle>
```

This solution is safe. In case one of both cases of repeatedly polling the environment for data applies, this is detected and an exception in thrown.

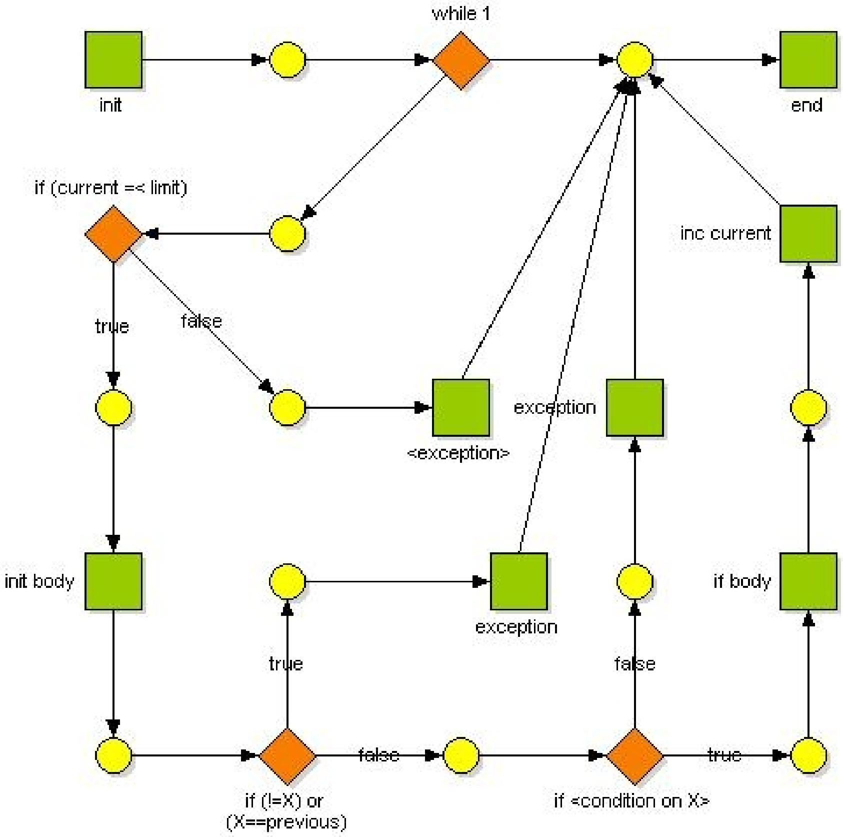


*Illustration 19: Safe Petri net for while-1 loop*

## A7. Performance optimizations

Here we describe a few optimizations to the SWAP architecture. These optimizations can be considered to be implemented in SWAP, in order to achieve a better performance.

### ***Optimization 1:*** *Client-side validation*

When a message is submitted, the SWAP backend (omniORB or the PostGreSQL database) checks the submitted data for (in)correct information. A number of those checks are rather trivial, which could easily be done at the client side, even on ultra-thin clients.

For example: suppose we want to submit a new contact (object which contains personal information) to the database and we enter a misspelled email address (no @, no subdomain, no top level domain etc). With this simple communication pattern, the whole object, including the SOAP control data has to be submitted to the server, parsed by the server and an error has to be replied back to the client. This implies that the whole path for accessing an object, has to be walked twice, just for generating an error message. We consider this to be very inefficient. An error like this could've easily been detected at the client side, before submitting the message.

This technique, which we'll refer to as client-side validation, is one of the important aspects of the AJAX concept. By doing a client-side validation, we can lower the number of exchanged messages, including the load on the connection, Like mentioned above, that is just what we need. There is another gain in applying this technique too: the responsiveness of client applications will become better, as validation results can almost be generated and displayed instantly, after submitting a message (which only will be done if the client-side validation succeeds successfully).

A good thing about this client-side evaluation is, that if we know how the SWAP server validates incoming information (that does not need any database checks), we can integrate these checks into our clients, without making any changes to SWAP. Of course, this will yield a double check for these fields. Without modifying SWAP, the server will still check for errors on fields we know they'll confirm to the requirements. As these checks do not put a high load on the server (and because of these checks, the server load will be lower anyhow), we do not see any problems in using this technique.

### ***Optimization 2:*** *Batched messaging*

In case a user wants to do a series of operations at once, doing this in a batched way usually gives the highest performance. If this is not done in some batched mode, for each operation, the entire path for setting up a connection, encoding, computing and decoding has to be walked. By doing as much as possible in a single session, the load on the connection to the server and the server itself will become better.

With SOAP, this can be done in a fairly easy way. SOAP supports multiple messages/operations per SOAP message. This implies that in a single session and a single SOAP envelope, a series of operations can be performed.

There is a drawback to this technique too; we have to take certain failures and problems into account. In single-operation SOAP messages, if some operation fails, the initiater (usually a user) receives an error message, acknowledging that the requested operation went wrong.

However, when submitting multiple operations at once, an error message could lead to confusion. For example, depending on how the receiving component for batched messages is implemented, some parts of the batch that should be computed, after the error occurred, might not be dealt with. This is a very important aspect (amongst others) to consider before allowing batched requests. Also, operations depending on earlier operations in a batch queue may suffer from errors.

For now, NH considers batched messaging as an interesting aspect for future use, but not interesting for the time being (integration issues)

### ***Optimization 3:*** *Smart data handling*

Optimizing the amount of data tripping the wire can be done from the architecture too. By adjusting the data that is transferred as good as possible to the requesters needs, the performance can possibly be increased (depending on the difference between the transferred information and the needed information).

There is always a trade-off point between optimality and generality. More general messages allow a simple messaging pattern; the drawback is that a requester will, most likely, not get the exact data requested, but more (or even worse, less) information.

### ***Optimization 4:*** *Message compression*

As SWAP messages are sent as SOAP messages, which are always plain text messages, the right compression technique can reduce the size drastically. Although this is no fix for redundant message fields (including their values), applying a compression algorithm before sending out a message (and, of course, upon deflating that same message upon receiving), can lead to much smaller messages. The drawback of this approach is the higher load message compressing/decompressing will cause on both the server- and clientside.

## A8. Petri net reduction techniques explained

### ***Technique 1:*** *Subnet substitution*

Due to the nature of our Python-to-Petri-net mappings, some constructions appear more than once. These constructions, which we will refer to as subnets from now on, would be checked more than once, which may be unnecessary.

In certain cases, we can replace a subnet by a single transition. The following rules apply:

1. The subnet should be sound
2. The initial place of the subnet to be substituted should be safe

An formal definition can be found in [10] (theorem 3, rule 3).

### ***Technique 2:*** *predefined reduction rules*

By replacing (sub)nets by simpler variants, we can reduce Petri nets easily to smaller nets. This subject has been researched [12] and yielded some very useful results. We use results from this research as distilled in [13].

This technique is somewhat similar to the substitution technique. Despite, there is a difference. For these substitutions, stronger requirements on the non-reduced nets apply. Although the substitution techniques are more powerful, the use of these predefined reduction rules is tempting. The gain in applying them is less, but so is the time it costs to apply them at all. The diagram (next page) shows those techniques.

The reduction rules as depicted in illustration 20 may need some additional explanation. All these rules apply as depicted, and to “weaker” versions of them (i.e. less incoming or outgoing arrows). However, they might not apply to stronger versions (more incoming or outgoing arrows). In case we want to use stronger versions of these rules, we have to check on their correctness!

### ***Technique 3:*** *parallel transitions*

This reduction preserves all the information from the original net in the reduced net, but makes the net “just smaller” [14]. When applying the parallel transitions reduction, all transitions with the same input *and* output places, are merged into a single transition. Of course, if we have multiple subnets (containing 1 or more transitions), with an identical set of input and output places, these nets can safely be reduced using this technique.

### ***Technique 4:*** *forced communication pairs*

The forced communication pairs reduction is a lossy reduction; not all properties of the original net will be preserved in the reduced net. The most important property, for the tests we want to perform on the reduced nets although, is that (possible) deadlocks are preserved [14]. This implies that this reduction is fine for deadlock testing.

This technique reduces a subnet (often described as a series of communications) to a single transition. Some constraints apply to this technique. If other transitions communicate with the subnet between the input and output place(s) of that subnet, this technique cannot be applied.

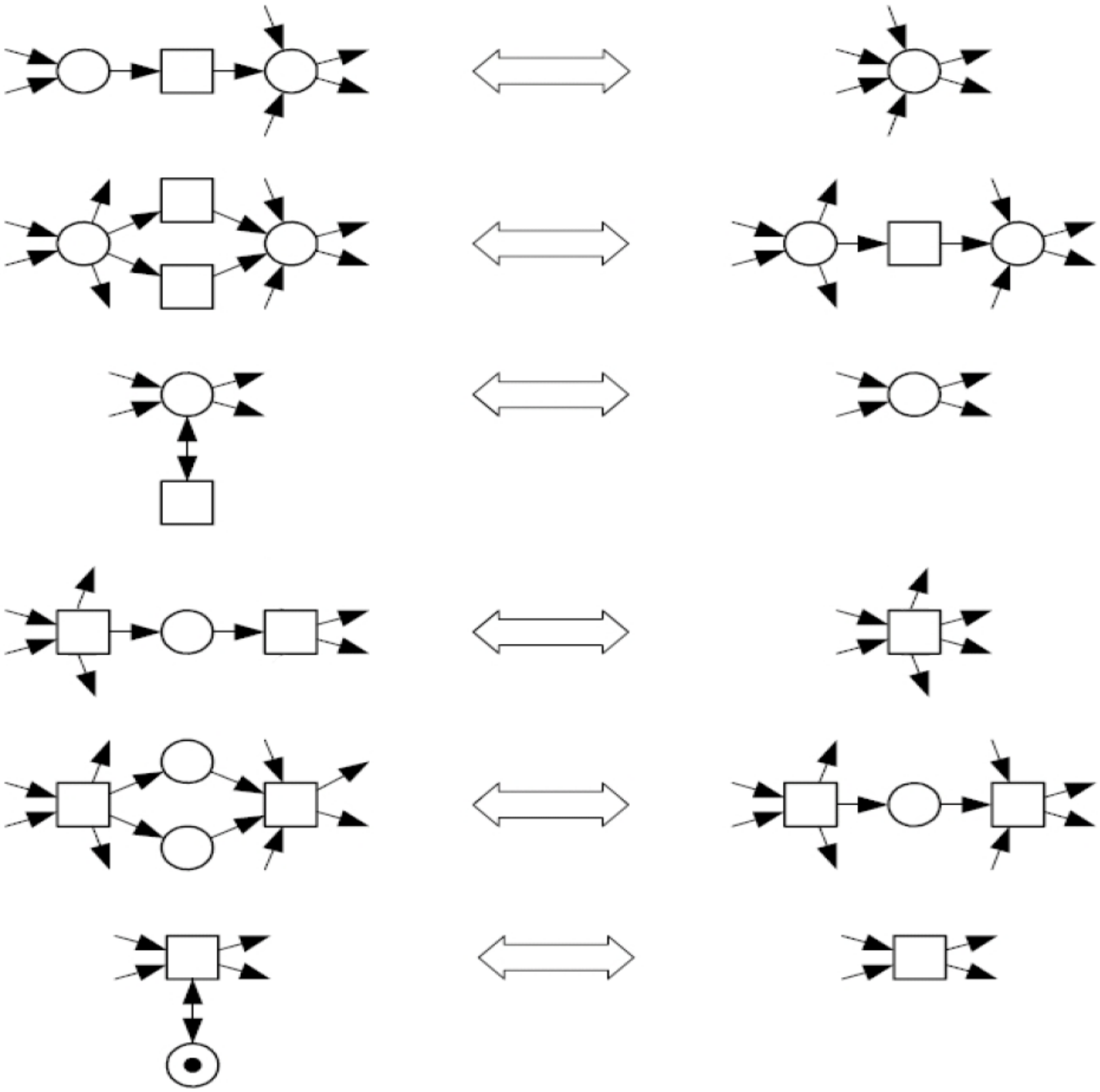


*Illustration 20: Predifined reduction rules*

### Application strategy

Our greedy strategy looks (as an algorithm, in pseudo-code style) like this:

```
REDUCE_PETRI_NET(PN)
    List subnets of PN which can be substituted
        Replace those subnets by the corresponding substitutions
    List subnets of PN which can be reduced by the Parallel transitions technique
        Replace those subnets by the a single transition, while keeping their input and output places
    List subnets of PN which can be replaced by equivalents from the reduction rules
```

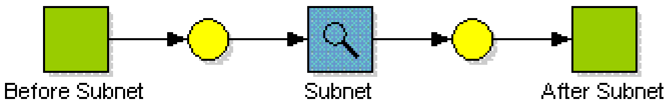
```
      Replace those subnets by the equivalents of the reduction rules
List subnets of PN which can be reduced by the Forced communications
techniques
      Replace those subnets by a single transition, while keeping their
      input and output places
```

We reckon the fact that this greedy algorithm is not 100% safe. It may happen that a certain reduction yields a less optimal solution than theoretically possible. If we notice this, during the reduction  of our Petri nets, we will deviate from the proposed algorithm, in order to gain a more optimal solution than the greedy algorithm would give us. As a reduction is not absolutely necessary (soundness checks on non-reduced nets checks should give the same results on reduced nets, if the reduction is done well), we can allow some sloppiness.

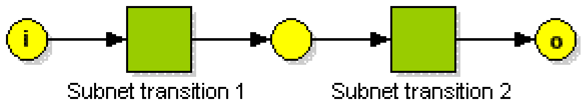
## A9. Petri net flattening rules

### *Hierarchical to non-hierarchical:*

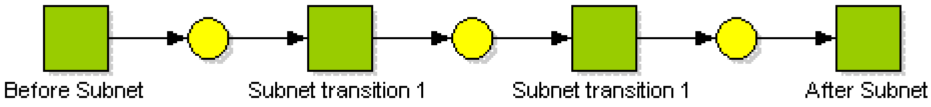
The net



with subnet



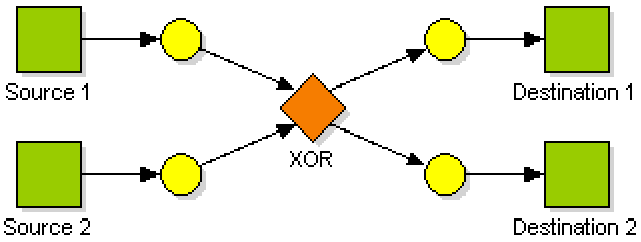
becomes



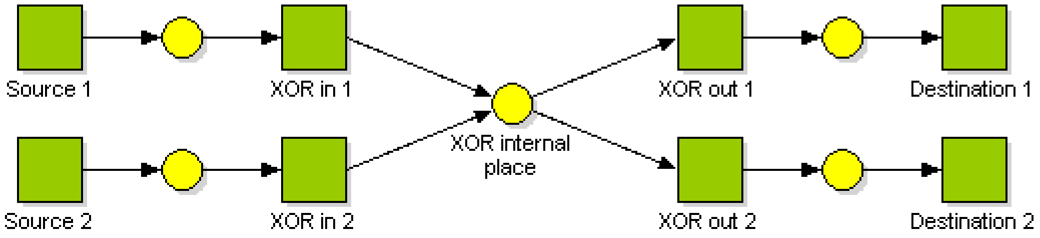
This flattened net is bisimilar to the original, hierarchical net.

### *XOR to non-XOR:*

A net with the following fragment (containing a XOR transition)



will be turned into this equivalent



This partial net is bisimilar to the fragment containing the XOR transition.

## A10. SWAP protocol overhead calculations

In a perfect world, not a single protocol would have overhead. In the real world, more (or even all) protocols have some overhead. Keeping overhead to a minimum is good practice to ensure maximum performance. In this appendix, we calculate the amount of overhead of the (initial) SWAP protocol and do some recommendations on how to force back the amount of protocol overhead.

We will not give a full specification of the overhead of all possible messages. SWAP itself is not responsible for message overhead, but the underlying CORBA layer is. We will, however, give some examples proving that message overhead should be taken seriously. And, we will prove that with SWAP, we can take care of this overhead in a reasonable manner.

We will calculate the overhead for *some* SWAP calls. This is done by making a SOAP call to Exchange4Linux, through SWAP, and evaluate the SOAP response. For all polymorphic calls (those calls can be instanced for multiple purposes or with multiple object types), we only evaluate a few important calls. After looking at each field in the SOAP response, we judge its necessity and its size. From our initial analysis, we know that redundant data is transmitted and we expect some information to be bulky.

For each SOAP message we evaluate, we make a table which mentions each key in the message. The table is used for checking duplicates. Each request & response message contains some basic information on the called process and the result of the call. As these parts of the message are mandatory to the correct working of the system, we will not put these keys in our tables.

**Note:** we do not model this table after the structure of the SOAP messages. The hierarchy in these XML documents is neglected, as this is of no importance to the amount of overhead in a message.

### 1. GetWorkgroupStorage (monomorph)

The GetWorkgroupStorage call is used to get information on the store (database) of the workgroup server. This call can be regarded as an initial call to the system. This call is not mandatory though; SWAP can be used fully without making a GetWorkgroupStorage call first.

Request message:

| Tag/key | Comment | Duplicate |
|---|---|---|
| options | | |

The body of the global GetWorkgroupStorage call is almost empty. There are no duplicates.

Response message:

| Tag/key | Comment | Duplicate |
|---|---|---|
| Exchange4Linux_SWAP_VERSION_MINOR | | |
| Exchange4Linux_SWAP_MIN_VERSION_MINOR | | |
| Exchange4Linux_SWAP_MIN_VERSION_MAJOR | | |
| Exchange4Linux_SWAP_VERSION_BUILD | | |
| Exchange4Linux_SWAP_MIN_VERSION_BUILD | | |
| Exchange4Linux_SWAP_VERSION_MAJOR | | |
| Exchange4Linux_SWAP_MIN_VERSION_RELEASE | | |
| Exchange4Linux_SWAP_VERSION_FULL | | |
| Exchange4Linux_SWAP_VERSION_RELEASE | | |
| USERS | | |

This response does not contain any duplicates.

## 2. GetObject (polymorph)

The GetObject call is used to retrieve any object from the database. The request message for all types is the same (as only the ID of the requested object has to be put in); the response message varies per objecttype.

Request message (email):

| Tag / key | Comments | Duplicate |
|---|---|---|
| storageId | | |
| objectId | | |
| options | | |

This request message does not contain any duplicates.

| Tag / key | Comments | Duplicate |
|---|---|---|
| ObjectId | | |
| ParentId | | |

| ***Tag / key*** | ***Comments*** | ***Duplicate*** |
|---|---|---|
| Class | | Class, ObjectClass, class, itemclass |
| Type | | |
| PR_CREATION_TIME | | |
| PR_LAST_MODIFICATION_TIME | | |
| Class | | Class, Class, ObjectClass, class, itemclass |
| Comment | | comment, PR_COMMENT |
| Created | | created |
| Delivered | | delivered |
| Flags | | flags, PR_MESSAGE_FLAGS |
| ItemNo | | |
| Modified | | modified |
| Name | | name |
| ObjectClass | | Class, Class, ObjectClass, class, itemclass |
| Parent | | parent |
| Serial | | serial |
| Status | | status, PR_MSG_STATUS |
| Title | | title, PR_DISPLAY_NAME |
| Type | | type |
| class | | Class, Class, ObjectClass, itemclass |
| comment | | Comment, PR_COMMENT |
| created | | Created |
| delivered | | Delivered |
| flags | | Flags, PR_MESSAGE_FLAGS |
| itemclass | | Class, Class, ObjectClass, class |
| modified | | Modified |
| name | | Name |
| parent | | Parent |
| seq_no | | |
| serial | | Serial |
| status | | Status, PR_MSG_STATUS |
| title | | Title, PR_DISPLAY_NAME |
| type | | Type |
| PR_0x36D00102 | | |
| PR_0x36D10102 | | |
| PR_0x36D20102 | | |
| PR_0x36D30102 | | |
| PR_0x36D40102 | | |
| PR_0x36D70102 | | |
| PR_ASSOC_CONTENT_COUNT | | |

| *Tag / key* | *Comments* | *Duplicate* |
|---|---|---|
| PR_ATTACH_NUM | | |
| PR_COMMENT | | Comment, comment |
| PR_CONTAINER_CONTENTS | | |
| PR_CONTAINER_HIERARCHY | | |
| PR_CONTENT_COUNT | | |
| PR_CONTENT | | |
| PR_CONTENT_UNREAD | | |
| PR_DISPLAY_NAME | | Title, title |
| PR_FOLDER_ASSOCIATED_CONTENTS | | |
| PR_FOLDER_TYPE | | |
| PR_LONGTERM_ENTRYID_FORM_TABLE | | |
| PR_MAPPING_SIGNATURE | | |
| PR_MDB_PROVIDER | | |
| PR_MESSAGE_DELIVERY_TIME | | |
| PR_MESSAGE_FLAGS | | Flags, flags |
| PR_MSG_STATUS | | Status, status |
| PR_RECORD_KEY | | |
| PR_SEARCH_KEY | | |
| PR_STATUS | | |
| PR_STORE_ENTRYID | | |
| PR_STORE_RECORD_KEY | | |
| PR_STORE_SUPPORT_MASK | | |
| PR_SUBFOLDERS | | |
| PROPNAMES | | |
| ObjectId | | |
| ItemNo | | |
| PR_MESSAGE_FLAGS | | |
| PR_SUBJECT | | |
| PR_SENDER_NAME | | |
| PR_MESSAGE_DELIVERY_TIME | | |

Out of 71 items in the message body, 19 of them are duplicates. This implies that the protocol overhead on the response of the GetObject call for email items is *at least* 27%. As all of the items in the message body are key ⇔ value pairs, the overhead may grow, depending on the length of the values of the duplicate keys. If, for example, the length of  a duplicate value is large compared to the rest of the message body, the overhead may become much worse (theoretically speaking, the overhead may grow up to 99,9%).

| *Tag/key* | *Comment* | *Duplicate* |
|---|---|---|
| storageId | | |
| objectId | | |

| *Tag/key* | *Comment* | *Duplicate* |
|---|---|---|
| options | | |

There are no duplicates here.

Response message (note):

| *Tag/key* | *Comment* | *Duplicate* |
|---|---|---|
| ObjectId | | |
| ParentId | | |
| Class | | Class, ObjectClass, class |
| Type | | |
| PR_MESSAGE_SIZE | | |
| PR_CREATION_TIME | | |
| PR_LAST_MODIFICATION_TIME | | |
| Class | | Class, ObjectClass, class, itemclass |
| Comment | | comment, PR_COMMENT |
| Created | | created |
| Delivered | | delivered |
| Flags | | flags, PR_MESSAGE_FLAGS |
| Itemno | | |
| Modified | | modified |
| Name | | name |
| ObjectClass | | Class, Class, class, itemclass |
| Parent | | parent |
| Serial | | serial |
| Status | | status, PR_MSG_STATUS, PR_STATUS |
| Title | | title, PR_DISPLAY_CONVERSATION_TOPIC, PR_NORMALIZED_SUBJECT, PR_SUBJECT, PR_DISPLAY_NAME |
| Type | | type |
| class | | Class, Class, ObjectClass, itemclass |
| comment | | Comment, PR_COMMENT |
| created | | Created |
| delivered | | Delivered |
| flags | | Flags, PR_MESSAGE_FLAGS |
| itemclass | | Class, Class, ObjectClass, class |

| ***Tag/key*** | ***Comment*** | ***Duplicate*** |
|---|---|---|
| modified | | Modified |
| name | | Name |
| Parent | | parent |
| seq_no | | |
| serial | | Serial |
| status | | Status, PR_MSG_STATUS, PR_STATUS |
| title | | Title, PR_DISPLAY_CONVERSATION_TOPIC, PR_NORMALIZED_SUBJECT, PR_SUBJECT, PR_DISPLAY_NAME |
| type | | Type |
| PR_0x10800003 | | |
| PR_0x340F0003 | | |
| PR_ALTERNATE_RECIPIENT_ALLOWED | | |
| PR_ATTACH_NUM | | |
| PR_BODY | | |
| PR_CLIENT_SUBMIT_TIME | | |
| PR_COMMENT | | Comment, comment |
| PR_CONVERSATION_TOPIC | | Title, title, PR_DISPLAY_NAME, PR_NORMALIZED_SUBJECT, PR_SUBJECT |
| PR_DELETE_AFTER_SUBMIT | | |
| PR_DISPLAY_BCC | | |
| PR_DISPLAY_CC | | |
| PR_DISPLAY_NAME | | Title, title, PR_DISPLAY_CONVERSATION_TOPIC, PR_NORMALIZED_SUBJECT, PR_SUBJECT |
| PR_DISPLAY_TO | | |
| PR_FOLDER_TYPE | | |
| PR_HASATTACH | | |
| PR_IMPORTANCE | | |
| PR_LONGTERM_ENTRYID_FROM_TABLE | | |
| PR_MAPPING_SIGNATURE | | |
| PR_MDB_PROVIDER | | |
| PR_MESSAGE_DELIVERY_TI | | PR_MESSAGE_DELIVERY_TI |

| *Tag/key* | *Comment* | *Duplicate* |
|---|---|---|
| ME | | ME |
| PR_MESSAGE_DELIVERY_TIME | | PR_MESSAGE_DELIVERY_TIME |
| PR_MESSAGE_FLAGS | | Flags, flags |
| PR_MSG_STATUS | | Status, status |
| PR_NORMALIZED_SUBJECT | | Title, title, PR_DISPLAY_CONVERSATION_TOPIC, PR_DISPLAY_NAME, PR_NORMALIZED_SUBJECT |
| PR_ORIGINATOR_DELIVERY_REPORTED_REQUESTED | | |
| PR_RECORD_KEY | | |
| PR_RTF_COMPRESSED | | |
| PR_RTF_IN_SYNC | | |
| PR_SEARCH_KEY | | |
| PR_SENSITIVITY | | |
| PR_STATUS | | Status, status |
| PR_STORE_ENTRYID | | |
| PR_STORE_RECORD_KEY | | |
| PR_STORE_SUPPORT_MASK | | |
| PR_SUBJECT | | Title, title, PR_DISPLAY_CONVERSATION_TOPIC, PR_DISPLAY_NAME, PR_SUBJECT |
| PR_SUBJECT_PREFIX | | |
| MAPI_000046E9 355840x0003 | | |
| MAPI_000046E9 355860x0003 | | |
| MAPI_000046E9 355870x0003 | | |
| MAPI_000046E9 355880x0003 | | |
| MAPI_000046E9 355890x0003 | | |
| MAPI_0000E6C1 340490x0003 | | |
| MAPI_0000E6C1 340510x000B | | |
| MAPI_0000E6C1 340540x000B | | |
| MAPI_0000E6C1 340620x000B | | |
| MAPI_0000E6C1 340640x0003 | | |
| MAPI_0000E6C1 340720x0003 | | |
| MAPI_0000E6C1 341300x0003 | | |
| MAPI_0000E6C1 341320x001E | | |
| MAPI_0000E6C1 341440x0040 | | |
| PROPNAMES | | |

Out of 86 items in the body of the message, 22 are duplicates. The *minimal* overhead is about 26%.

Response message for a contact:

| ***Tag/key*** | ***Comment*** | ***Duplicate*** |
|---|---|---|
| ObjectId | | |
| ParentId | | |
| Class | | Class, ObjectClass, itemclass, class |
| Type | | Type, type |
| PR_MESSAGE_SIZE | | |
| PR_CREATION_TIME | | |
| PR_LAST_MODIFICATION_TIME | | |
| Class | | Class, ObjectClass, itemclass, class |
| Comment | | comment |
| Created | | created |
| Delivered | | delivered |
| flags | | Flags |
| ItemNo | | |
| Modified | | modified |
| Name | | |
| ObjectClassParent | | |
| Serial | | serial |
| Title | | title |
| Type | | Type, type |
| class | | Class, Class, ObjectClass, itemclass |
| comment | | Comment |
| created | | Created |
| delivered | | Delivered |
| flags | | Flags |
| itemclass | | Class, Class, ObjectClass, class |
| modified | | Modified |
| name | | |
| parent | | |
| seq_no | | |
| serial | | Serial |
| title | | Title |
| type | | Type, Type |
| PR_0x10800003 | | |

### 3. GetFolderList (monomorph)

The GetFolderList call is used to list all the folders in a store. Upon specifying the store (MORE INFO), all the folders, including the folder hierarchy, are listed.

Request message:

| *Key/tag* | *Comment* | *Duplicate* |
|---|---|---|
| storageId | | |
| store | | |

There are no duplicates here.

Response message:

| *Tag/key* | *Comment* | *Duplicate* |
|---|---|---|
| UID | | |
| PARENTUID | | |
| CLASS | | |
| FolderDepth | | |
| DISPLAY_NAME | | |
| COMMENT | | |
| ACL | | |

There are no duplicates here. This may sound a bit surprising, as the keys in the table occur multiple times in the response message. This is because the keys are *per folder* keys (unique per folder). For each key in the specified folder in the GetFolderList request, these keys are listed (the root folder, called “IPM::SubtreeFolder”, excepted).

### 4. DeleteObjects (polymorph)

Although the DeleteObjects is polymorph (it can be applied to every type of object, including folders), the request and response messages always look the same.

Request message:

| *Key/tag* | *Comment* | *Duplicate* |
|---|---|---|
| objectIds | | |

There are no duplicates here.

Response message:

| *Key/tag* | *Comment* | *Duplicate* |
|---|---|---|
| message | | |
| key | | |
| message | | |

Although the “message” tag appears twice in the response message, the meaning of both is different. So, there are no duplicates.

### Conclusion

Although we did not give a full specification or measurement of the message overhead SWAP inherited, we see that calls which are formed after the object they are embodying, always contain a significant amount of overhead. And that is an important aspect of the system (mainly the Exchange4Linux CORBA layer) which needs to be taken care of.

## A11. WSDL

Generally speaking, there are multiple ways of opening up a SOAP server to the world. One way, which is relatively easy, common and fast (and thus often preferred), is to develop a SOAP server and let the development environment (Microsoft Visual Studio .NET for example), generate a descriptional file, a so-called WSDL file ,from the source code.

This WSDL file describes and specifies the interface to remote procedures (webservices). This interface consists of 6 parts:

1. global definitions
2. types
3. messages
4. ports
5. bindings
6. services

The first part, the definitions section, describes all the namespaces [15] that are used/needed to use this WSDL file. These namespaces define several important aspects, such as (simple) types and messages. Namespaces can be considered as classes which can be instantiated to make use of the definitions they hold. This makes rapid development for commonly used things possible.

The second part, the types section, describes how variables should be typed. SOAP can handle both primitive and complex types. The set of built-in simple types SOAP has are inherited from XML [16]. This section is used for self-defined types as well (which mostly exist of simple types).

The third part, the messages section, describes what messages should look like. Here, per type of message, the names and type of the data fields are specified. These fields can both be simple type, or instances of complex types.

The fourth part, the porttype section, specifies what messages are used for a (remote stored) procedure which can be called. Usually, a remote procedure is accessed through a “request” call and answers as a “response”. Invoking a remote procedure is done through a port; the port addresses the messages.

The fifth part, the bindings section, augments SOAP actions (which are in fact ports) with namespaces. This section is used to determine what namespaces are used for the defined port.

The sixth and last part, the service part, specifies what service the WSDL should communicated with. Or, put differently, the connection to the SOAP server is specified here.

Using a WSDL is not required for using SOAP. This WSDL-less mode, often referred to as the non-WSDL mode, is not preferred unless absolutely necessary. Using non-WSDL mode requires much more code to be written and makes the development of SOAP programs slower and more error prone.

All primitive types, supported in WSDL, are actually the standard types supported by XML. Different or more complex types can be defined by using these primitives. The table below lists the XML primitive types.

| | | |
|---|---|---|
| string | Boolean | float |
| double | decimal | binary |
| integer | nonPositiveInteger | negativeInteger |
| long | int | short |
| byte | nonNegativeInteger | unsignedLong |
| unsignedInt | unsignedShort | unsignedByte |
| positiveInteger | date | time |

*Table 6: XML types*

Besides these primitive types, WSDL also supports higher level types, which are derived from object oriented languages. These high level types, always declared as a “complexType” objects, can be accumulated with additional operators, as the table below shows:

| *Type* | *Meaning* |
|---|---|
| restriction | adds restrictions to the allowed data in the base type |
| extension | adds functionality to a specified base type |

*Table 7: SOAP complexType extensions*

Arrays are supported too. In WSDL, there is a notion of a sequence. This sequence can be used on any type, including complex types. By specifying a “minOccurs” and “maxOccurs” value, which can be set to any natural number (taking maxInt into account that is) or ”unbounded”, quite some flexibility is offered [17].

## A12. SOAP error catching: SOAPfault

The SOAP protocol does feature a fault detection mechanism, called the SOAPfault system. SOAPfault knows the following faults: VersionMismatch, MustUnderstand, DataEncodingUnkown, Sender and Receiver. The latter two errors, SOAPfault::Sender and SOAPfault::Receiver, relate, amongst other errors, to networking errors.
From errors generated by the TCP/IP stack, SOAPfault::Sender and SOAPfault::Receiver can generate human readable error messages. More information on the SOAP faults can be acquired from [18].

## A13. Filtering Solutions

### ***Solution 1:*** *creating a new object*

A first solution would be to create a new object which contains only the fields that are required for that type of workgroup object. After composing the new object, we would send that one out, instead of the class generated from the CORBA object.

```
NewObject = InstanceOf TEMPLATE
FOR EACH Attribute IN ReceivedObject
    IF Attribute IN NewObject
        FILL IN NewObject VALUE FROM Attribute
```

With n as the number of attributes in the received object and m as the number of attributes in the new object, this is a $\Theta(nm)$ solution, which is not really good. Another drawback of this solution is that upon creating the new object, the memory load might become quite high, especially when objects with (binary) attachments are involved.

### ***Solution 2:*** *delete unwanted attributes*

For each SOAP message, there is a single Python object which contains all the data (in key/value pairs) for that message. Python features the possibility to modify instanced classes. One of the methods of the class modifiers is the "del" method, which enables us to remove attributes from class instances. By removing all unwanted attributes, an "optimized" version of the class instance will remain. Sending out that object will render a more compact SOAP message.

A reason to prefer this solution is that the delete method will be performed in $\Theta(1)$ time (so for the whole object, containing n key/value pairs, this would be performed in $\Theta(n)$ time) . The drawback is that we would need to keep up a list of unwanted/unneeded attributes per (workgroup) objecttype. This results in limited flexibility and is not really pretty.

### ***Solution 3:*** *filter on wanted attributes*

By defining, per workgroup objecttype, a set of required attributes, we can filter on these attributes. As Python is an imperative language, this would lead to a low-performing solution.

```
FOR EACH Attribute IN PyObj
    IF NOT (Attribute IN RequiredAttributes)
        Delete Attribute
```

The problem with this solution is that for each attribute, we need to check the set of required attributes. This will not perform very well, as, for n received attributes and m required attributes, this would result in a $\Theta(nm)$ solution, because, for each attribute, we need to check a set of required attributes.

From an architectural point of view, this approach seems acceptable, although the performance will most likely be bad, especially for objects with a high count of attributes.

Verifying automatically generated objects against a predefined set of keys is pretty though, as we would be using some kind of template.

Python features some constructs we know from functional programming languages (Haskell for instance) which are programmed to be efficient. The *filter* function would be very useful here. Filter is used to filter elements of a list; after applying filter (on a list of length n, function filter runs in Θ(n) time) to a list, only the elements which satisfy a (specified) predicate will remain. The problem is that this construct *only* operates on list types; we have to filter object attributes though. Still, this filter function can be used.

As we cannot just use the filter function, we need to look into a somewhat more complex solution. Python features a set of operations which can list properties of class instances. The most common functions for listing class properties are the *dir*, *getmembers* and *getattr* functions or the __*dict*__ class built-in property. We decide to go with the __*dict*__ property.

By making a list of all the attributes in the object, we can filter on any attribute not in the list of prerequired keys. After making this list, unwanted keys can be removed. All this operations are done in Θ(n) time, so  this will result in a Θ(n) algorithm, which is pretty.

```
AttributeList = LIST OF (Object.Attributes)
DeleteList = FILTER (NOT (AttributeList, RequiredAttributes))
FOR EACH Item IN DeleteList
    DELETE Object.Item
```

We prefer this solution, because it runs in reasonable time and allows the required flexibility.

### From specification to implementation

The required and generated collection of attributes can be considered as a (mathematical) set. To remove unwanted attributes from the generated set of attributes, we should compute the intersection between the required and generated set. As the required attributes set is a subset of the generated attributes set, we'll end up with an attribute set that is equal to the required attribute set.

Formally expressed:

R = set of required attributes

G = set of generated attributes

F = set of filtered attributes

R is a subset of G: $R \subseteq G$

F is a subset of G and after finishing, equal to R: $F \subseteq G \wedge R \subseteq F \wedge F \subseteq R$

So, for each message with overhead, we have to compute the filtered set. Neuberger & Hughes will define these sets.

### Program code

This predicate can be used for filtering on unwanted objects. In semi-pseudo code, our program will look like this:

```
AttributeList = Object.__dict__.keys()
DeleteList = AttributeList - RequiredAttributes
FOR Item IN DeleteList if not Item.startswith(“_”)
    delattr(Object, Item)
```

This code fragment is a pretty straightforward translation from the pseudo code. The only important difference is the the second condition for the “FOR” loop, which is necessary as we would otherwise be deleting class methods too.

## B1. Vocabulary / abbreviations

The list hereunder contains the definition of less common words and abbreviations used throughout the thesis document. The words / abbreviations explained below are, in the text, once per word / abbreviation, marked in blue. The list is sorted in an alphabetical order.

*Apache HTTP:* open source webserver by the Apache foundation

*BASE64:* a method to denote binary codes / strings in ASCII

*bisimulation:* a relation between two or more transition systems; if the bisimulation holds, the systems should behave perfectly alike

*branching bisimulation:* a stronger variant of the bisimulation equivalence in which for all actions in the first system, there is a corresponding action in the second system

*branching point:* a certain part in a programs code where the chosen path is determined by a case distinction

*colored Petri net:* a variant on regular Petri nets, where data can play a role

*commit:* making a set of tentative changes (here, to the database) permanent

*CORBA:* stands for Common Object Request Broker Architecture, a middleware solution

*CVS:* stands for Concurrent Versions System, a filesystem like environment which can intelligently deal with different versions of (non-binary) files, mainly used in software development

*data abstraction:* replacing the outcome of a test by a nondeterministic choice when converting programming (Python) code into a Petri net

*DOM:* stands for Domain Object Model, an object oriented approach of HTML and XML

*deadlock-free:* a system, program, or model, where no deadlock situation can occur

*groupware:* multi-user environment, used for storing and accessing centralized organizational data such as emails, agendas, notes, task lists and logbooks

*guard:* a condition (often on data, often expressed as a predicate) which can be either true or false

*Hierchical Petri net:* a Petri net in which transitions can be Petri nets too

*HordeMail:* an IMAP webmail client developed with PHP

*HTTP:* stands for HyperText Transfer Protocol, a communication protocol for communication between webbrowsers and webservers

*IMAP:* stands for Internet Message Access Protocol, a centralized manner of storing email on a server, using a live connection to view email items

*inert:* no changes are inflicted by a series of transactions

*Java:* a platform independent, object oriented programming language, originally designed by the SUN company, prominent for webapplications

*JAX:* a Java SOAP implementation

*LaQuSo:* a research facility of TU/e

*Linux:* an open source, POSIX compatible, operating system, created by Linus Torvalds

*mCRL2: stands for micro CRL, version 2, a process algebraic language which can take data into account*

*Microsoft Exchange:* Microsofts (market leading) groupware server

*middleware:* software that stands between applications or application layers, often used for converting communication or data between distinct native formats

*namespace:* set of variables (possibly including their values / definitions)

*NuSOAP:* a PHP SOAP implementation

*OmniORB:* an open source CORBA object request broker (middleware layer), developed in C++

*Open source:* way of software distribution where the software's source code is freely accessible

*overhead:* in our case, the part of a message which is not part of the required data

*PHP:* stands for PHP HyperText Processor, a scripting language originally intended for simple webapplications; nowadays it's an object oriented programming language, mainly used on the internet

*PNML:* Yasper's native fileformat

*POSIX:* stands for Portable Operating-System Interface for uniX, a standard for interfacing to UNIX and UNIX like systems (often referred to as *nix) systems

*PostgreSQL:* a relational database system using the SQL language for access

*Python:* originally an object oriented scripting language, nowadays a prominent language for commercial and scientific software

*rollback:* making a database mutation ((partial) commit) undone

*RPC:* stands for Remote Procedure Call, a messaging technique in which objects restored remotely are accessed through a predefined interface and protocol

*Scalix:* a groupware solution by the SCALIX company

*SMWF:* stands for StateMachine Workflow Net, a Petri net in which every transition has exactly one incoming arc and one outgoing arc. Also, each place has one incoming arc and one outgoing arc, except for the initial and final place; any SMWF is sound

*SOAP:* stands for Simple Object Access Protocol, a XML-based protocol for sending messages over a HTTP network

*SOAP message body:* the most important part of a SOAP message, stored in a SOAP envelope, usually used for storing the request or response data

*SOAP message envelope:* the object containing a SOAP message

*SOAP message header:* a part of a SOAP message, stored in a SOAP envelope, giving information on the SOAP message and its environment; sometimes “misused” to store data too

*Squirrelmail:* an IMAP webmail client developed with PHP

*SWAP:* stands for Simple Workgroup Access Protocol, designed and implemented by Neubergher & Hughes to allow easy access to Exchange4Linux (and possible other workgroup servers)

*tag:* a variable container in both HTML and XML

*TCP:* stands for Transaction Control Protocol, part of the TCP/IP protocol responsible for the transmission of data packets and their checksumming

*W3C:* stands for World Wide Web consortium, an alliance responsible for standardizing web technologies

*webservice:* a service or application which can be accessed over a network (the internet, in most cases)

*whitepaper:* a decision paper shorty documenting a solution of a design

*WofLan:* stands for WorkFlowAnalyzer, a Petri net analyzer by TU/e

*WSDL:* stands for Web Services Description Language, a XML document describing the interface to a web service

*XML:* stands for extensible Markup Language, a both human- and machine readable language that allows to a structured markup for documents

*XML RPC:* a remote procedure call protocol using XML as markup language

*XOR transition:* a Petri net construct used by Yasper, which represents a single place with multiple (zero or more) incoming arcs and multiple (zero ore more) outgoing arcs; often used to model branching points

*XSLT:* stands for XML StyleSheet, allows the rewriting of XML documents in an automated way by following the StyleSheet

*Yasper:* stands for Yet Another Smart Process EditoR, a Petri net modelling and simulation program by TU/e and Deloitte

*Zimbra collaboration suite:* a groupware solution by the Zimbra company

*ZSI:* stands for Zolera SOAP Implementation, a SOAP implementation by the Zolera company (bankrupt), nowadays an open source SourceForge project

## B2. References

[1] http://www.w3.org/TR/soap/

[2] http://www.w3.org/TR/soap/

[3] http://www.w3.org/TR/soap12-part1/#soapfault

[4] Branching time and abastraction in bisimulation semantics, Journal of the ACM, 43(3):555—600, by R.J. van Glabbeek and W.P. Weijland, [ARGlWe96]

[5] http://www.w3.org/TR/wsdl

[6] http://en.wikipedia.org/wiki/Base_64

[7] http://www.capeclear.com

[8] http://www.ietf.org/rfc/rfc2045.txt

[9] SWAP whitepaper: http://www.neuberger-hughes.com/pub/swap/swap-ref.pdf

[10] Workflow verification: Finding control-flow errors using Petri net based techniques by W.M.P. van der Aalst

[11] The modeling, analysis and synthesis of communication protocols by Siyi Terry Dong. [DON83]

[12] Petri nets properties, analysis and applications by T. Murata (IEEE vol. 77 no. 4 1989), [MUR89]

[13] Verification of WF-nets by H.M.W. Verbeek

[14] A compact Petri net representation for concurrent programs by Matthew B.Dwyer, university of Massachusetts

[15] http://www.google.nl/url?sa=X&start=2&oi=define&q=http://www.dpawson.co.uk/xsl/xslvocab.htm

[16] http://www.dpawson.co.uk/xsl/xslvocab.html

[17] http://www.developer.com/services/article.php/10928_1602051

[18] http://www.w3.org/TR/soap12-part1/#soapfault

## B3. Links

These links refer to web pages containing information on the subjects we dealt with in this thesis. Interested readers may want to visit these pages to gather more information on a certain subject. This set is not complete, nor intended to be complete. Despite this, the information on these links will give a very good impression on the matters discussed.

Neubergher & Hughes: http://www.n-h.net/

Nlcom: http://www.nlcom.nl

SOAP (brief): http://en.wikipedia.org/wiki/SOAP

SOAP (in detail): http://www.w3.org/TR/soap/

WSDL (brief): http://en.wikipedia.org/wiki/WSDL

WSDL (in detail): http://www.w3.org/TR/wsdl

ZSI: http://pywebsvcs.sourceforge.net/zsi.html

NuSOAP: http://dietrich.ganx4.com/soapx4/

Python: http://www.python.org/

PHP: http://www.php.net/

Exchange4Linux: http://www.exchange4linux.com/

Zimbra: http://www.zimbra.com/

Scalix: http://www.scalix.com/

Colamo: http://www.colamo.org/

Petri nets (and related subjects): http://www.informatik.uni-hamburg.de/TGI/PetriNets/

Well behaving modules: http://ieeexplore.ieee.org/iel5/373/5730/00218219.pdf#search=%22wbm%20petri%22

Yasper: http://www.yasper.org/

WofLAN: http://is.tm.tue.nl/research/woflan.htm

PNML page: http://wwwis.win.tue.nl/~jmw/pnml/

## B4. Time scheme

| *Week* | *Planned* | *Actual* |
|---|---|---|
| *1* | **S:** SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP |
| *2* | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP |
| *3* | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP |
| *4* | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP, Ex changExchange4Linuxinux, OmniORB, SWAP | **S**: SOAP, WSDL, Python, ZSI, Apache Axis, Apache Ant, Subversion, AJAX, PHP, PHP-SOAP, NuSOAP |
| *5* | **S**: Exchange4Linux, SWAP | **S**: Exchange4Linux, SWAP |
| *6* | **E**: PHP, NuSOAP, **S**: SWAP servercode | **E**: PHP, NuSOAP |
| *7* | **S**: SWAP servercode, Code to Petri net conversion techniques | **S**: SWAP servercode, Code to Petri net conversion techniques, **E**: PHP, NuSOAP |
| *8* | **I**: NuSOAP based SWAP client | **I**: NuSOAP based SWAP client |
| *9* | **I**: NuSOAP based SWAP client | **I**: NuSOAP based SWAP client |
| *10* | **I**: NuSOAP based SWAP client | **I**: SWAP servercode to Petri net conversions |
| *11* | **I**: SWAP servercode to Petri net conversions | **I**: SWAP servercode to Petri net conversions |
| *12* | **I**: Petri net optimizations, **E**: Python, ZSI | **I**: Petri net optimizations |
| *13* | **E**: Python, ZSI | **E**: Python, ZSI |
| *14* | **I**: Petri net optimizations, **E**: Python, ZSI, WSDL | **I**: Petri net optimizations, **E**: Python, ZSI, WSDL |
| *15* | **I**: Python SWAP client, **T**: SWAP message overheads | **I**: Python SWAP client, **T**: SWAP message overheads |
| *16* | **S:** Zimbra, mCRL2 | **S:** Zimbra, mCRL2 |
| *17* | **S:** Zimbra, mCRL2, **I**: PNML mCRL2 conversions | **S:** Zimbra, mCRL2, **I**: PNML mCRL2 conversions |
| *18* | **I**: Petri net optimizations | **I**: Petri net optimizations |
| *19* | **S:** WofLan, **I:** PNML TPN conversions | **S:** WofLan, **I:** PNML TPN conversions |
| *20* | **I**: SWAP WofLan testing | **I**: Petri net optimizations, SWAP WofLan testing |
| *21* | **I**: SWAP WofLan testing, Zimbra WSDL | **I**: SWAP WofLan testing, Zimbra WSDL, **S**: ZSI / WSDL |
| *22* | **D**: Writing of thesis | **D**: Writing of thesis |
| *23* | **D**: Writing of thesis | **D**: Writing of thesis |

| *Week* | *Planned* | *Actual* |
|---|---|---|
| *24* | **S**: SWAP architecture, **I**: SWAP architecture adjustments | **S**: SWAP architecture, **I**: SWAP architecture adjustments |
| *25* | **I**: SWAP architecture adjustments | **I**: SWAP architecture adjustments |
| *26* | **I**: Zimbra WSDL, WSDL to Python conversions | **I**: Zimbra WSDL, WSDL to Python conversions, ZSI modifications |
| *27* | **D**: Writing of thesis | **D**: Writing of thesis |
| *28* | **D**: Writing of thesis | **D**: Writing of thesis |
| *29* | **I**: Zimbra WSDL, WSDL to Python conversions, ZSI modifications | **I**: Zimbra WSDL, WSDL to Python conversions, ZSI modifications |
| *30* | **I**: Python/ZSI Zimbra client | **I**: Python/ZSI Zimbra client |
| *31* | **I**: Python/ZSI Zimbra client | **I**: Python/ZSI Zimbra client |
| *32* | **I**: Python/ZSI Zimbra client | **I**: Python/ZSI Zimbra client |
| *33* | **I**: Zimbra ZSI client, ZSI modifications | **I**: Zimbra ZSI client, ZSI modifications |
| *34* | **I**: Zimbra ZSI client, ZSI modifications | **I**: Zimbra ZSI client, ZSI modifications |
| *35* | **D**: Writing of thesis | **D**: Writing of thesis |
| *36* | **D**: Writing of thesis | **D**: Writing of thesis |
| *37* | **D**: Writing of thesis | **D**: Writing of thesis |
| *38* | **D**: Writing of thesis, **I**: Zimbra ZSI client, ZSI modifications | **D**: Writing of thesis, **I**: Zimbra ZSI client, ZSI modifications |
| *39* | **D**: Writing of thesis, **I**: Zimbra ZSI client, ZSI modifications | **D**: Writing of thesis, **I**: Zimbra ZSI client, ZSI modifications |
| *40* | **D**: Documentation for NH (ZSI, NuSOAP), roundup | **D**: Writing of thesis, Documentation for NH |
| *41* | **D**: Writing of thesis, Documentation for NH | **D**: Writing of thesis, Documentation for NH |

*Legend:*

**D**: Documentation

**I**: Implementation (action)

**E**: Experiment

**S**: Study