Filling a hole in the UML architecture

Written by
Jakob Sverre Løvstad
jakobsve@stud.ntnu.no

Teaching supervisor/guide
Gunnar Brataas
Gunnar.Brataas@idi.ntnu.no

Subject teacher
Alf Inge Wang
Alf.Inge.Wang@idi.ntnu.no

Fall, 2003
(5th year, 9th semester)
Directed study, subject code: TDT4735

Software Engineering Group,
The Department of Computer and Information Science,
Faculty of Information Technology, Mathematics and Electrical Engineering

NTNU
Abstract

In this project, I mainly consider the two modelling languages UML and SP. I compare the two languages within the Development and Physical view of Philippe Kruchten’s 4+1 view model. For the actual comparison, I use the quality framework for conceptual modelling languages developed by John Krogstie and Arne Sølvberg.

The report employs a main software system example to illustrate briefly how modelling in UML and SP works. The architecture design languages SADL and ACME are also explained since these are compared to SP later in the report. And, of course, the quality framework is also presented, explained and tailored for this project’s needs.

The evaluation shows how SP compares to the other three languages for the various aspects of the quality framework. There are quite many details here on all the positive and negative properties of the languages. The comparison between SP and SADL and ACME is less comprehensive than the comparison between SP and UML because the underlying goal of this paper is to show that SP should extend UML in the Development and Physical view of the 4+1 view model.

The conclusion shows that SP deals very well with complexity and offers a very tidy, structured way of expressing domains. It does have some flaws in visual design as well as being not very good for expressing more vague parts of domains. It’s also more difficult to learn. All in all, though, it’s a better solution than UML’s corresponding modelling methods (namely the component and deployment diagrams).

The last part of the report suggests further work on the SP language to deal with problems expressed in the conclusion. This section is full of ideas for things that could be done to improve SP. What will actually be done about this is left for further research work…
Acknowledgement

Thanks to Gunnar Brataas who’s been my guide through this project and also a partner for discussion on the various topics. Also thanks to Peter Hughes for his work on the Structure and Performance Architectural Design Language and for his input on the report.

Finally I would like to thank in general those who’ve taught the classes at NTNU that have given me the foundation of knowledge needed to go through with this project.
Preface

This paper is meant for software engineers with basic knowledge of software architecture and the Unified Model Language (UML). All the other concepts in this report will be explained in adequate detail to understand what is being said and there are references available to relevant in-depth articles on the different subjects should you desire to learn more about them.

I hope you find this report as interesting to read as it was to write!

Jakob Sverre Løvstad
Index

Abstract......................................................................................................................... 2
Acknowledgement........................................................................................................... 3
Preface............................................................................................................................ 4
1 Introduction ................................................................................................................ 7
  1.1 The problem ........................................................................................................ 7
  1.2 Motivation ............................................................................................................ 8
    1.2.1 Expanding knowledge about the domain ..................................................... 8
    1.2.2 Preventing misunderstandings ..................................................................... 8
    1.2.3 Facilitate implementation ............................................................................. 8
    1.2.4 Reviving SP and finding good use for it ....................................................... 9
  1.3 Goals ................................................................................................................... 9
  1.4 Approach ............................................................................................................. 9
  1.5 Scientific method of this paper .......................................................................... 10
  1.6 Structure of this report ...................................................................................... 10
2 State-of-the-art......................................................................................................... 11
  2.1 The 4+1 View Model of Architecture ................................................................. 11
    2.1.1 Logical view ................................................................................................ 12
    2.1.2 Process view .............................................................................................. 12
    2.1.3 Development view ...................................................................................... 13
    2.1.4 Physical Architecture .................................................................................. 13
    2.1.5 Scenarios .................................................................................................... 13
  2.2 UML within the 4+1 View Model ....................................................................... 13
    2.2.1 The Case example ..................................................................................... 13
    2.2.2 UML in the Logical view ............................................................................. 14
    2.2.3 UML in the Process view ............................................................................ 17
    2.2.4 UML in the Development view ................................................................... 19
    2.2.5 UML in the Physical view ........................................................................... 20
    2.2.6 UML in Scenarios ....................................................................................... 21
  2.3 SP within the 4+1 View Model .......................................................................... 24
  2.4 Quality Framework for comparing UML and SP ............................................... 28
  2.5 A brief introduction to ACME ............................................................................ 32
  2.6 A brief introduction to SADL ............................................................................. 35
3 Comparison of UML and SP.................................................................................... 36
  3.1 Domain appropriateness ................................................................................... 37
    3.1.1 Domain appropriateness for UML .............................................................. 37
    3.1.2 Domain appropriateness for SP ................................................................... 38
  3.2 Participant knowledge appropriateness ............................................................ 38
    3.2.1 Participant knowledge appropriateness for UML ....................................... 39
    3.2.2 Participant knowledge appropriateness for SP .......................................... 40
  3.3 Comprehensibility appropriateness ................................................................. 40
    3.3.1 Comprehensibility appropriateness for UML ............................................. 40
    3.3.2 Comprehensibility appropriateness for SP ............................................... 41
  3.4 Technical actor interpretation appropriateness ................................................ 42
    3.4.1 Technical actor interpretation appropriateness for UML ......................... 42
    3.4.2 Technical actor interpretation appropriateness for SP ............................... 43
  3.5 Other things to consider ................................................................................... 44
    3.5.1 Tool support ............................................................................................... 44
    3.5.2 Community acceptance .............................................................................. 44
1 Introduction

This first chapter describes the Whats and Whys of my project. Hopefully, after reading this chapter, you will have a clear understanding of what the problem is and why it should be solved. Afterwards is should be easier to follow the rest of this report as I start discussing the actual research work, present results and come to conclusions.

I have structured this chapter into seven subchapters. This should be sufficient to give you some basic knowledge on the origins of my task.

1.1 The problem

In this section I would like to talk about the problem itself and the background for it.

The project at hand is called “Filling a hole in the UML architecture”. The best starting point for explaining the background for it is to quote the text from the assignment given as a summary of the assignment:

“Structure and Performance (SP), developed by Peter Hughes at the Department of Computer and Information Science, has the potential to complement UML so that they better can express different perspectives of an architecture. UML is basically a collection of various partial languages that are more or less integrated. The strong parts of SP seem to cover the weak parts of UML. It is desirable to study to what degree SP fulfils the requirements for an Architecture Description Language and potential adjustments needed in SP. It is furthermore desirable to document that SP fills a hole in UML by modelling the same architecture using only existing UML and using UML extended with SP. Finally, one may study how SP can be integrated as well as possible with the other partial languages that make up UML.”

-Gunnar Brataas

For those of you who are unfamiliar with UML, it’s an acronym for Unified Modelling Language and it’s one of the most popular modelling languages available. There are numerous tools and software development systems supporting UML and it’s taught to a lot of students in Computer Science. I refer you to [1W] if you would like to learn more on this topic (there will be a brief introduction in chapter 2.2, however). Be advised that this report does not deal with UML 2.0 since this version isn’t yet in common use.

SP is short for Structure and Performance and is a modelling language made by Peter Hughes in 1988. It’s not widely known yet, but deals with hierarchical models that often span from software on the top down to hardware at the bottom. I refer you to [1A] for more information on SP (a short introduction will be given in chapter 2.3 for convenience).

The principle is rather simple as may have been clear from the quote from the assignment text. We want to see if SP, possibly somewhat refined, can be better for some parts of the modelling UML is used for, thereby strengthening UML through integration with SP. We want to do this from a software architecture point of view.
1.2 Motivation

Doing a project such as this and trying to fill a conceptual hole in UML is all well, but most readers with some curiosity will ask “why”. Is the hole there because there isn’t, in fact, any need for it to be filled? Or is UML underdeveloped in that area for some reason? In this chapter, I will explain some reasons for why this project is interesting and justify its existence.

1.2.1 Expanding knowledge about the domain

SP is, as further explained later, quite reliant on a high degree of domain knowledge. Having to build a proper SP model of the system you wish to develop will force you to ask yourself a lot of “unpleasant” questions about the domain and you will have to find some answers that puritans in software development can’t always answer.

This may seem like a lot of work, but it’s quite necessary that developers have a sound understanding of what they’re building. It will often become apparent through SP that there are important answers to be answered before implementing the system.

You will also gain a more proper understanding of how the software fits into the underlying layers of both lower level software (like CORBA, JVM and so on) and hardware. And proper understanding is what modelling is really all about.

1.2.2 Preventing misunderstandings

It’s often the case that several teams or independent developers are working together on projects to develop various systems. The people making the software (for example a e-commerce system) are most likely not the people setting up the hardware (for example the database servers that stores all orders). If the software people are disregarding the hardware setup, they may be in for a surprise when they see how the hardware people have decided to build the network, servers and whatever other components they are using.

To clarify for all parties involved how the system is supposed to look in a structural layout, SP is possibly a good way of bridging the gaps between layers of software and between software and layers of hardware. A lot of money can be lost on having to change things later on or spending valuable time with the different teams explaining things that should be clear. Making a proper model from the beginning is a better strategy.

1.2.3 Facilitate implementation

Today the trend in software development is that most of the effort in building a system is directed purely at software. This might seem quite natural and the way it’s supposed to be, but it often presents a problem when it’s time to actually implement the system. Being blind to the realities of lower level software and hardware limitations and structure can become quite disastrous.

SP can ease the transition from design to implementation. It lets you specify the environment for the software system in detail and help clarify the problems and finding solutions to them.
Especially when designing systems that run on several different platforms and operating systems, it’s important to have a good grasp of how it’s all supposed to fit together. If the server is running Linux and the client Windows NT, you need to know how that affects your system.

1.2.4 Reviving SP and finding good use for it

Quite aside from the positive aspects of using SP, part of the motivation for making this report and continuing work on this language is that it’s been dormant for a long time even though it has a lot of potential. Extending UML with SP can help get SP into more common use and increase its popularity. Solving what seems to be a problem in UML at the same time is a good way of doing this.

1.3 Goals

So far, I have only brought up somewhat vaguely what I wish to do on this project. In this chapter, I will define what it is I want to achieve in a more detailed manner. It’s important to have this in memory as I get more into the actual solution to the problem, so we don’t lose track of what I’m trying to achieve.

I will list the goals as a checklist in the order I will try to achieve them:

1. Find an appropriate architectural framework.
2. Find an appropriate example to model.
3. Model the example and structure the model according to the architectural framework chosen.
4. Find or design a framework for comparing UML and SP.
5. Compare UML and SP.
6. Compare SP to other ADLs with respect to the tasks we feel it is better at than UML.
7. Indicate that SP is in fact better than UML for some parts of the architectural framework.
8. Suggest modifications and further work for SP.

1.4 Approach

Having discussed the problem, reasons for solving it and the more detailed goals I wish to achieve, the next step is to take a look at how to go about actually making it happen. This will be more or less a more elaborate version of chapter 1.3.

Since I am working in the context of software architecture, the first part of attacking the task is choosing an architectural framework to work within. This will also help structure the examination of UML versus SP a lot as opposed to just pointing to random parts of the languages. This way, the project and the report should be much easier to follow as well.

Secondly, I will use the framework to model an example system using first UML and then modelling the appropriate parts of the system in SP. This should show some differences that will help me find out how SP fits into the system modelling picture. The important part in this respect is to choose a system that is relevant, so I have proposed some criteria that should be fulfilled for our example:
1. The system should be easily related to by other people in the modelling and software architecture field. Choosing an example system that is very unusual and unknown will hardly prove anything.

2. The example chosen should already be as completely modelled in UML as possible. This is important to not spend too much work inventing the wheel – the goal of this project is not getting training in UML modelling. It’s also good to use UML modelled by someone else as one may fall into the trap of doing less than optimal UML modelling when trying to show the usefulness of another modelling language in certain areas.

3. The system shouldn’t be very complex. It will be hard for readers of this report to spend lots of energy getting a good understanding of a complex system and it takes the focus away from what’s relevant (the UML versus SP topic).

4. The system should be distributed. Having a system based on a single workstation (that is, a computer with one CPU and one memory) doesn’t take into consideration all the modelling areas that I wish to consider. Very few common systems are based on only one workstation these days, so this point is also related to the first argument in this list.

Thirdly, after having two somewhat differing models to work with, I will have to look at the view or views of the architectural framework where I have used SP to complement UML and prove that SP is actually better suited for the job. This is perhaps one of the most difficult tasks involved in my project as it depends on whether it’s possible to see actual improvements using SP. The basis for this comparison will be the quality framework described in [1B].

Fourthly, since there are many ADLs in use, SP should be compared to some of these to see if it is the best choice for our chosen task among its peers. This comparison has to be done according to some preset criteria and within the parts of the architectural framework we feel SP is useful.

Finally, I have to actually do something useful with the results I have found throughout the project.

1.5 Scientific method of this paper
This report has been largely created on the basis of literature studies. Articles, books and web pages have played central roles in forming an understanding of the concepts involved. In addition, my guide in this project, Gunnar Brataas, is also working with SP and has been a partner for discussions as well as a source of information and ideas. Peter Hughes, the man responsible for making SP, has also given his input to some parts of the project.

In addition to this, a lot of the methods used for comparisons and structuring the concepts come from courses I have previously been involved with. Having had experience with using architectural frameworks and language evaluation methods was rather crucial.

1.6 Structure of this report
My project differs somewhat from many others in the course it is part of (course nr. SIF8094) in that it discusses an existing technology in a new context/paradigm rather
than inventing something new altogether. So this report is structured to accommodate this.

After this introduction, I will have a chapter describing the state-of-the-art within software architecture, the languages involved in this project, the evaluation framework I will use for SP as well as any other necessary concepts that need to be introduced.

The next part consists of most of my own contribution to the project. This is where most of the goals presented in chapter 1.3 will be looked into and it’s the most challenging part of my work both for myself and readers of this report.

After having completed the main part of the report and performed all the actual research work, it’s time to tie up all the loose ends through two chapters dealing with the actual results, evaluation and conclusions first and then further work.

That’s it for the introductory chapter. Let’s move on to discussing state-of-the-art of concepts involved in this project!

2 State-of-the-art

This chapter deals with state-of-the-art concepts within the different fields that this report touches on. It’s basically a prestudy, or background research, that I have done to build up knowledge needed to complete this project.

I will present the different concepts in separate subchapters in the order I have used them (the concepts, that is) in the actual solving of the problem, which is also the order they will appear in the part of this report that deals with the actual solution (as explained in chapter 1.4).

2.1 The 4+1 View Model of Architecture

To put this project into a more specific frame, I need to use an architectural framework to discuss how UML compares to SP. I have chosen the “4+1 View Model of Architecture” from [3A], hereafter referred to only as the “4+1 model” or the “4+1 view model”. I chose this framework because it is established and well documented. By having chosen the architectural framework, I have achieved my first goal for this project (see chapter 1.3).

Goal 1: Find an appropriate architectural framework

As the name of the 4+1 model hints at, it consists of four main views with a fifth which is supposed to more or less knit the four others together. Please have a look at illustration 2.1 to see Philippe Kruchten’s overview of the views as shown in [3A]. The arrows show the order in which to model the different views. You can’t, for example, start out with the Physical view and end up with the Logical view of the model you’re working on.
I will give a brief introduction to the different views in the following chapters. The only real problem with Kruchten’s approach is that he uses the Rational Rose modelling language to model the 4+1 views. That notation is inappropriate for this project (if you wish to learn more about this notation, see [2W]). In chapter 2.2 I will show how to utilise the 4+1 view model with UML for more intuitive and relevant results.

2.1.1 Logical view

The logical view is meant to show the functional requirements for the system. This means modelling objects or object classes and the resulting model is some kind of class diagram. Such a class diagram displays a set of classes and their logical relationships modelled through various connectors that signify association, usage, composition and inheritance (these are, at least, the usual types).

The style for this view is object oriented. There are several modelling languages that support class diagram, in fact, it is quite common. Example languages include UML (class diagrams), E-R and EEML to name a few.

2.1.2 Process view

The process view takes into account some non-functional requirements. Examples include performance, availability, security, modifiability, time and so on. In this view we model threads of control within the program.

Modelling a process means grouping tasks to form an executable task. For my purpose, a process model will be some kind of sequence diagram meant to show how a thread, or several threads, forms a task or tasks that the program executes.
It’s often necessary to do this in software development to show communication in a program and the order in which the program does its work.

2.1.3 Development view

This view is concerned with showing the organization of the software development environment in terms of modules. All software of any size can be split, or categorized, into such modules to get a better overview of the system. The connectors in such a diagram shows what other modules each module uses in some way.

2.1.4 Physical Architecture

This view deals with mapping software to hardware. In models belonging to this view, we try to see how our software system is distributed on hardware devices in different ways. We may use a flat model to just show how a program is spread out on different servers and clients or a hierarchy to show how different hardware devices support applications.

2.1.5 Scenarios

This view is meant to put together the other four previously described views. Kruchten describes these sets of scenarios as instances of use cases with corresponding scripts. As the name of the 4+1 view model hints at, this view is redundant – the scenarios make up the “+1”.

Scenarios describe interaction between objects. They can be used as a help to discover architectural elements during design as well as a validation and illustration tool when the design is complete.

2.2 UML within the 4+1 View Model

In this chapter, I will get into a very brief introduction to UML and how it fits into the 4+1 model described in the previous chapter. We discovered how this could be done in a course at NTNU taught by Letizia Jaccheri described in [4A].

I will go through the different parts of UML by putting them into the different parts of the architectural framework from the previous chapter. To add some structure to the explanation and form a basis for comparison between UML and SP, I want to introduce an example that will be modelled in the different views. I will explain the case example first and then go on to handling the UML part.

2.2.1 The Case example

Choosing an example was quite a challenge. As mentioned in chapter 1.4, I chose a few important criteria for the example I wanted for this project and they were hard to fulfill. A common problem is that few examples are complete when it comes to being modelled in UML, so the second criteria was a big hinder. Many examples from books, articles and websites had a lot of modelling done on class diagrams,
sequence diagrams and the other most common UML features, but lacked diagrams describing the system’s physical architecture and component organisation.

This led to some investigation in trying to find a standard example used in information modelling, but common example systems such as the IFIP conference, the library system and so on, proved to be either too simple or inappropriate for this project (meaning they didn’t fit in well with the UML versus SP problem).

The result was that the second criteria was partly sacrificed for the sake of the other three. I have chosen an example that is a very standard system in software development, but ended up having to model most of it myself. The example case for this project goes like this:

“**A hotel reservation system consists of a client application and a server application.** People who want to either check availability of rooms, reserve a room or cancel a reservation use the client application embedded in a webpage. The client application has to collect information about what the customer wishes to do as well as information about the customer if he or she wants to reserve a room or cancel a reservation.

The server application receives requests from the client application, translates these requests and fetches the needed information from an SQL database. This information is then returned to the client application for presentation to the customer.

Both client and server applications are programmed in Java 1.4.2. The GUI is structured according to the Model-View-Controller Architecture (MVC).”

Although the case may be interpreted to various levels of complexity, it gives a good basic example to use as a starting point. Note the absence of any reference to security in this case. I have chosen to ignore security since it only adds complexity without adding anything that might be useful to model in the context of this project.

✔ Goal 2: Find an appropriate example to model.

### 2.2.2 UML in the Logical view

As described in chapter 2.1.1, the logical view is commonly modelled as some sort of class diagram. UML is indeed one of the most well known languages for class diagrams. Illustration 2.2 shows a simple metamodel for UML class diagrams. The actual information is largely taken from [1B] and [3W].

![Illustration 2.2 – Metamodel for the UML class diagram (taken from [3W])](image)
Each class in the diagram is described what it is, not what it does. Every class contains at least its name, and normally also attributes and/or methods (sometimes also referred to as variables and operations). Illustration 2.3 shows the basic class structure and an example of a simple class.

Illustration 2.3 – Classes in UML (taken from [3W])

All classes in a UML class diagram are connected through three types of connectors describing: association, aggregation and generalisation. Association can be said to be the weakest link (pardon the pun), while generalisation is the strongest.

Association describes an “is-a-member-of” relationship. So if we have a class men and a class women, they would have an association type relationship with a class gender-groups. Associations may also have N’ary relations that add information to the relations such as one car has only one engine or one car has many wheels. It’s also possible to show which class contains what other by use of arrows instead of lines – such as a student is contained by a college (and not the other way around). Illustration 2.4 shows association conceptually and with N’ary relations and with “containership”.

Illustration 2.4 – Association in UML (taken from [3W])

Aggregation describes an “is-part-of” relationship. So if we have a class wheels and a class engine, they would have an aggregation type relation to a class car. Like association, one may also add N’ary relations, but aggregation can also in some cases be a so called composition relation. Composition implies a constraint on the multiplicity of the aggregate side: it can only take the values 0 or 1. Illustration 2.5 shows aggregation both in normal cases and as composition (shown by using a black diamond on the aggregation end instead of a unfilled one).

Illustration 2.5 – Aggregation in UML (taken from [3W])
Generalisation describes an “is-a” relationship. So if we have a class whale and a class human, they would have a generalisation type relation to a class mammal. This concept is often referred to as inheritance where the class mammal would be called a superclass. Illustration 2.6 shows an example of generalisation.

![Diagram of generalisation in UML](taken from [3W])

Now that I have explained the main parts of class diagrams in UML, it’s time to show the class diagrams for our case example from chapter 2.2.1. Illustration 2.7 shows the client side class diagram, while 2.8 shows the server side. Note that in illustration 2.7, the “Serializable Client” class is just a consequence of thinking in Java.

![Client side class diagram for the case example](taken from [3W])
2.2.3 UML in the Process view

Process modelling in UML isn’t the most supported part of the modelling language, but it’s done through the UML activity diagram. These diagrams show what processes are done by different threads in a program. The main entities in the model are activities and arrows that flow between them to show transitions. Illustration 2.9 shows the simplest form of an activity diagram.

In addition to this structure, there’s an optional, but commonly used, decision box to show how the program decides which activity to go to next. The common notation for this is shown in illustration 2.10.
There is also often a need to synchronize events in this type of diagram, so there’s something called a synchronization bar available that signifies that two things start or end simultaneously. Illustration 2.11 shows both events starting and ending at the same time (“Cool down” has to happen before “Switch off heating” and “Open windows” which both have to happen before “Measure the temperature”).

Lastly, it’s common to use a filled circle to signal the start of a program and a filled circle with a white border to signify the end of a program. This is good if one wants to have several places the program can end in a program and it makes the model tidier. Illustration 2.12 shows a complete diagram based on the example case from chapter 2.2.1.
2.2.4 UML in the Development view

The development view becomes a component diagram in UML. We can in these diagrams see how the software is distributed into subsystems using components as well as the interfaces and dependencies that connect them. This part of UML is not very well described in general, but I will briefly introduce the basics.

The component diagram consists primarily of components with interfaces that are bound together by dependencies. The components are boxes, the interfaces are lines with circles attached and the dependencies are arrows trailing dash lines. Illustration 2.13 shows the component itself and two components connected through an interface and a dependency.
As with the previous chapters, I will finish this one with a diagram made from the example case in chapter 2.2.1. Illustration 2.14 shows this component diagram.

Illustration 2.14 – Component diagram for the example case

2.2.5 UML in the Physical view

UML in the physical view is basically just an extension of UML in the development view which is called a deployment diagram. Instead of just looking at the components, we look at how to group the components to make shallow, and possibly interacting, hierarchies that end up in the high level hardware descriptions of our systems.
It’s common to group components into packages. Components sensor and decision logic may be part of a package robot control. These packages may again be grouped into nodes. The nodes are usually more encompassing entities like workstations, servers and other software carrying things. Have a look at illustration 2.15 for the basics of a deployment diagram.

Illustration 2.15 – The basics of a deployment diagram

Once again I will show what the diagram in question will look like for the example case in chapter 2.2.1. Illustration 2.16 shows this deployment diagram.

Illustration 2.16 – Deployment diagram for the class example

2.2.6 UML in Scenarios

The scenarios become sequence diagrams in UML. These show the sequence of communication between different parts of the program, normally the instances of the classes in the class diagram (see chapter 2.2.2). This is modelled as objects interacting through method calls and return values as well as object instantiations and terminations.

Each object is represented as a box with a dotted line hanging down from it. All the objects are lined up on top of the diagram, normally in the order they are instantiated for easy reading. At the point the object is instantiated, the dotted line
becomes a slim box trailing downwards until the object terminates, at which point it goes back to being a dotted line. Illustration 2.17 shows an object in a sequence diagram and how the life of an object is modelled.

Illustration 2.17 – An object in a sequence diagram and an object’s life in the diagram (taken from [3W])

The method calls are modelled as arrows and they are also named according to which method they are calling at the target object. You can also differ between synchronous and asynchronous messages by using a normal arrow for synchronous messages and half an arrow for asynchronous messages. See illustration 2.18 for clarification.

Illustration 2.18 – The difference between synchronous and asynchronous messages (taken from [3W])

When modelling return messages (return values) from the various objects in the diagram, it is also necessary to think of implicit and explicit returns. If it’s implicit, that is if it’s for a synchronous method call, it’s not usually necessary to model it at all. If it’s a return for an asynchronous call, it needs to be explicit and modelled. See illustration 2.19 for clarification.
Illustration 2.19 – Implicit and explicit returns (taken from [3W])

I will finish the whole UML section by showing the sequence diagrams for the example case in chapter 2.2.1. Illustration 2.20 shows the client side sequence diagram while 2.21 shows the server side.

Illustration 2.20 – Client side sequence diagram for the example case
2.3 SP within the 4+1 View Model

SP does not cover the entire 4+1 model. Rather, SP fits into part of the framework and I hope, through this project, to show that it is better than UML in the areas that it does cover. The areas of the 4+1 view model that SP might fill are the development and the physical view.

SP doesn’t consist of that many modelling entities – it’s mainly just a collection of square boxes and straight lines. But it’s still quite difficult to use simply because it’s demanding on the modeller’s skills in finding the proper entities to use and also his or her knowledge and understanding of the system.

The basic entity in SP diagrams is the square box. This box can be pretty much everything from a hardware device or software application to an organisational unit like “Secretary”. It really depends on the granularity chosen for the diagram and what you’re trying to model. Example boxes for different granularities can be “SCSI controller”, “Client application”, “Disk transfer logic” or even “Sales administration”.

For practical purposes (to make the models tidier mainly), you can also use a box with a dotted border instead of a solid one. This symbolizes that the entity consists of a network of sub entities modelled elsewhere. This means that you can cram a whole model of some important part of the system into a higher level hierarchy by saying that a certain box is really, for example, the whole file handling system. This is a better solution than having to actually model every part of the system in one big unreadable model.

By putting one box behind another, so that it looks like the foremost box has a shadow, you can signify that the box is, in fact, several equal entities. If, for example, you have several instances of the same entity, like a CPU meant for a specific purpose, you can model them into one box this way instead of having to draw every single one. Drawing one box several times with the same name would also be wrong for this purpose in SP, since doing this signifies that all those boxes are the same entity. This is another practical feature of SP since it means you can make duplicates instead of having several crossing lines to a box at one position in the diagram. Illustration 2.22 shows the entities in SP.
Illustration 2.22 – The basic entities of SP

In addition to the basic boxes, there are also three basic connectors: communication, process and memory operation. The communication connector is a dotted line from one box to another. This signifies a message of some kind being sent down the hierarchy.

The process connector is a thin, unbroken line from box to box that signifies a process request of kind. For example, a process request might be sent from an application down the hierarchy to a CPU.

The memory operation connector is a thick, unbroken line from one box to another. This connector signifies a request sent to a storage device of some sort. For example, a memory operation can be sent from an application through a database system to a physical disk. The three connector types are illustrated in illustration 2.23.

Illustration 2.23 – The three connector types in SP

There is also a basic set of rules to help the SP programmer. These were formulated by Minkowitz in 1991 and are as follows (taken from [5A]):

1. Any non-primitive SP component must have at least one processing and one memory subcomponent.
2. If neither processing or persistent memory (or both) of a non-primitive SP component is distributed, a communication subcomponent is required. Only one communication subcomponent for each non-primitive SP component is allowed.
3. A non-primitive SP component cannot be a communication component if one of its ancestor components is a communication component.
4. If processing in a non-primitive SP component is distributed, there should be one processing subcomponent for each distributed processing unit, e.g. CPU.
5. If persistent storage in a non-primitive SP component is distributed, there should be one memory subcomponent for each distributed storage unit.

In addition, Vetland made some SP rule extensions in 1993 that are as follows (taken from [5A]):
1. A non-primitive SP component may have a communication relationship with more than one subcomponent.
2. There may be more than one communication link on any path between a top-level component and a primitive component.

So all in all, SP consists of three boxes, three connectors and two rather complicated rule sets. Let’s finish by looking at SP diagrams for the case from chapter 2.2.1. Keep in mind that this diagram is supposed to fill the same role as the UML deployment and physical diagram. Illustration 2.24 shows a simple, less detailed model of the system, while illustration 2.25 shows a more extensively detailed model of the same system.

Illustration 2.24 – Simple SP diagram for the example case

For the more complicated model in illustration 2.25, I feel some more explanation is needed. This model is a very important part of this report and should be understood at least to some degree.

On the client side the three entities “User Strategic”, “User Interaction” and “Keyboard/Display” represent three granularity levels of mental activity and are therefore connected to the body’s processor: “User Brain”. “User Strategic” is the general wish to order a room at the hotel in the case. “User Interaction” is the user actually booking a room with certain number of beds at a certain date and so on. “Keyboard/Display” is the body interacting with the system at the most basic neurological level where the fingers manipulate the keyboard and the eyes see what happens on the display (screen).

The MVC (Model-View-Controller) pattern from the case is represented in the illustration through the entities “Screen Model” (Model), “View Screen Manager” (View) and “(Main) Request Proc/Controller” (Controller). It’s actually interesting that patterns can become so evident and easily visible in an architectural model.

In addition, there are also two transfer objects that function as caches called “Client Transfer Object” and “Server Transfer Object”. The rest should be understandable.
In addition to the SP diagram in illustration 2.25, I will have to take a closer look at the “HW platform” modules in the diagram. These boxes are surrounded by broken lines which mean they’re actually diagrams in their own right that are hidden for readability and tidiness. Let’s have a look at illustration 2.26 for the contents of the subsystems. Note that in this subsystem you don’t really need to include the entity “Virtual Memory”, but the model places emphasis on paging, so it becomes necessary (or at least of interest).
Goal 3: Model the example and structure the model according to the architectural framework chosen.

2.4 Quality Framework for comparing UML and SP

To compare UML and SP, I have chosen the approach designed by Krogstie and Sølvberg described in chapter 3 of [1B] as a basis. This is a quality framework for evaluating conceptual modelling languages and consists of five categories that the language is considered in. These categories are:

- **Domain appropriateness:**
  To quote [1B], “there are no statements in the domain that cannot be expressed in the language”. This basically means that you should ideally be able to express everything in the domain you’re modelling in while not being able to express anything outside it. Questions we might like to ask about a language to evaluate this quality include:
  - Can you model both hard facts and vague knowledge about the domain (is it flexible in precision)?
  - Does every statement in the language have a unique representation?
  - Can you model everything in the domain and nothing outside it?

- **Participant knowledge appropriateness:**
  This category states that the language should correspond as much as possible with how those who wish to use it perceive reality. This has a lot to do with how intuitive the language is, how aesthetically pleasing it is and how the symbols used to represent various statements and entities are designed. One might like to ask the following questions to help determine this quality aspect:
    - Does the language have a natural feel to it?
    - Do the various symbols represent what you expect them to?
    - Are there better choices for the various symbols?
    - Are the symbols similar to standard symbols seen elsewhere (recognition through association)?
• **Knowledge externaliability appropriateness:**
  This quality aspect is somewhat similar to domain appropriateness, but is considered from a participant’s point of view. So here one evaluates if it’s possible to model all explicit knowledge from the participant. This very briefly described in [1B] and is mostly discarded (possibly due to debate on whether this type of quality is really applicable). I will not use this since I feel that domain appropriateness covers this type of quality more sensibly (this category doesn’t really add much to the picture).

• **Comprehensibility appropriateness:**
  This has to do with how well participants in the modelling effort understand the language’s statements. This involves pretty much anything that makes the language comprehensible. [1B] mentions aspects like level of detail, simplicity of symbols, making important statements more visible and so on. Possible clarifying questions include:
  - Is it easy or at all possible to emphasize more important parts of the model?
  - Are the important statements easy and brief to express?
  - Can one manipulate size, colour, fonts and so on to bring attention to certain parts of the model?
  - Can one group symbols to make the model more readable?

• **Technical actor interpretation appropriateness:**
  The basis for this aspect is that for the language to be reasoned about automatically, it requires a formality in syntax and semantics (both must be operational and/or logical). On the other hand, the language also needs to be efficient enough to be of practical use (executability). One might ask the following questions to evaluate this category of quality:
  - Is it possible to hide and/or encapsulate parts of the model to make it more readable and create independent modules?
  - Do the rules allow illogical modelling (loopholes in the rules)?
  - Are the rules consistent with each other (no contradictions)?
  - Do the rules forbid modelling that should be possible?
  - Is it easy to interpret the language rules?
  - Is it possible to organize data hierarchically to avoid having to look at everything at once?

So all in all, we’re left with four categories for the evaluation of languages. Illustration 2.27 shows the relations between the different quality aspects.
Illustration 2.27 – The quality framework for conceptual modelling languages (taken from [1B])

It is also helpful to look at [2A] which explains Krogstie’s and Jørgensen’s framework for evaluating conceptual models (as opposed to conceptual modelling languages). This framework consists of eight categories:

- **Physical quality:**
  This is basically well the model expresses the domain. That is, how easy it is for those reading the model to make sense of it. The actual definition of physical quality is somewhat more complex, but this is the gist of it.

- **Empirical quality:**
  This quality category deals with how easy it is to interpret the model for users of it. The deciding factors here is how good the model looks ergonomically and aesthetically. So here we have to consider shapes of boxes, how easy it is to misunderstand the different drawn entities, whether the model looks very untidy and so on.

- **Syntactic quality:**
  Syntactic quality is how correct the model is in regard to the modelling language. To quote [2A]: “This includes aspects of lexicon correctness, syntax correctness and structural quality”.

- **Semantic quality:**
  This is how well the model fits into the modelling domain. This means that the model has to be correct (all elements in the model exist in the domain) and that the model has to contain everything in the domain (all elements in the domain have to be in the model).

- **Perceived semantic quality:**
  Perceived semantic quality is how well the participants’ interpretation of the model coincides with their knowledge of the domain. After all, the model is the result of a modeller knowledge of the domain and how well he or she manages
to model that knowledge. This can’t be checked directly, but we can compare the different participants’ interpretation of the model.

- **Pragmatic quality:**
  This is the correspondence between the model and its interpretation by those reading it. So this often relies on how clear and concise the model is.

- **Social quality:**
  Social quality is the agreement of the participants’ interpretations of the model. Basically, one can say that a very ambiguous model doesn’t have high social quality.

- **Organisational quality:**
  This is a somewhat hard to understand category. It evaluates the model’s ability to directly or indirectly help fulfil the prioritised goals of modelling and that all its prioritised goals are addressed.

It can be very helpful to use these categories in some way as well. The slight problem is that since these categories are originally meant for the models themselves, one has to consider how the language supports these categories! So for, say, the physical quality category, one might ask “Does the language enforce a well arranged model?” instead of “Is the model well arranged?”.

I also have to consider the fact that since I’m considering languages and not models, some categories may be improper because they depend upon a model being viewable by some audience. Perceived semantic quality and organisational quality are two such categories that I will have to disregard.

The remaining six categories fit nicely in under the four original ones from [1B]. By that I mean that they can be used as related categories and will help in understanding what I’m looking for when I start comparing UML with SP. You’ve probably already noticed similarities between some of the original four categories and the six taken from [2A]. Table 2.1 summarizes the framework I have constructed from [1B] and [2A].

<table>
<thead>
<tr>
<th>Category</th>
<th>Related category/ies</th>
<th>Evaluates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Domain appropriateness</td>
<td>Physical quality</td>
<td>• Language in accordance with domain</td>
</tr>
<tr>
<td></td>
<td>Semantic quality</td>
<td>• Capability of modelling</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Representation of entities</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Expressiveness</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Flexibility in precision</td>
</tr>
<tr>
<td>Participant knowledge appropriateness</td>
<td>Social quality</td>
<td>• Ambiguity</td>
</tr>
<tr>
<td></td>
<td>Pragmatic quality</td>
<td>• Clarity</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Consistency</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Interpretability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Intuitive representation</td>
</tr>
<tr>
<td>Comprehensibility appropriateness</td>
<td>Empirical quality</td>
<td>• Aesthetics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Ergonomics</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Design</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Readability</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• Symbol discrimination</td>
</tr>
</tbody>
</table>
Table 2.1 – The quality framework constructed for comparing UML with SP

<table>
<thead>
<tr>
<th>Technical actor interpretation appropriateness</th>
<th>Syntactic quality</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Symbol simplicity</td>
<td>• Rules</td>
</tr>
<tr>
<td>• Visual emphasis</td>
<td>• Lexicon correctness</td>
</tr>
<tr>
<td>• Symbol composition</td>
<td>• Syntax correctness</td>
</tr>
<tr>
<td>• Symbol composition</td>
<td>• Structural quality</td>
</tr>
<tr>
<td>• Syntax correctness</td>
<td>• Usability</td>
</tr>
<tr>
<td>• Structural quality</td>
<td>• Usability</td>
</tr>
<tr>
<td>• Usability</td>
<td>• Encapsulation</td>
</tr>
</tbody>
</table>

Goal 4: Find or design a framework for comparing UML and SP.

2.5 A brief introduction to ACME

ACME isn’t really part of the main evaluation of this report, but is an extra language for comparison with SP. Therefore, I will only give a very brief description of this ADL and not go into such detail as I have offered for UML and SP. For some more in-depth information on ACME, have a look at [6A] and [4W].

ACME is referred to by its makers as the Architectural Description Interchange Language. As the name implies, it’s supposed to be a sort of common ground for various ADLs to accommodate easier development of tools and the like. In other words, it’s supposed to be a more generic ADL.

ACME consists of a graphical and a textual part. The graphical part is very straightforward with square component boxes tied together with connectors (symbolised by arrows) to ports in the boxes. In addition, you can add properties and representations. The basics of ACME’s graphical representation is shown in illustration 2.28 and 2.29.
The textual part of the ACME language is a bit like pseudo code where you specify properties of the various entities. Here you can add pretty much what you
want as long as you follow the structure defined in the ADL. An example bit of code is shown in illustration 2.30. I apologize for the lack of readability in illustrations 2.28, 2.29 and 2.30. These are screenshots from articles and the quality was less than hoped for.

```
System simple_cs = {
    Component client = {
        Port send-request;
        Properties { Aesop-style : style-id = client-server;
                     UniCon-style : style-id = cs;
                     source-code : external = "CODE-LIB/client.c" });
    
    Component server = {
        Port receive-request;
        Properties { idempotence : boolean = true;
                     max-concurrent-clients : integer = 1;
                     source-code : external = "CODE-LIB/server.c" });
    
    Connector rpc = {
        Roles {caller, callee}
        Properties { synchronous : boolean = true;
                     max-roles : integer = 2;
                     protocol : Wright = "..." });

        Attachments {
            client.send-request to rpc.caller ;
            server.receive-request to rpc.callee }
    }
}
```

Figure 2.30: Client-Server System with Properties in ACME

Illustration 2.30 – Textual representation of entities in ACME

To clarify ACME a bit, I’ve also made a very simple ACME model with corresponding ACME code that shows the example case used in chapters 2.2 and 2.3. Have a look at illustration 2.31 and 2.32. Keep in mind that the example is a very simple one. It’s possible to expand it a lot in detail, but this seems superfluous since ACME doesn’t play a big role in this report.

Illustration 2.31 – Visual model of the example case
2.6 A brief introduction to SADL

SADL, like ACME, is another language that is included in this report to add more to the analysis of SP, but isn't really part of the main effort here. I will give a brief introduction to SADL, but those who want to get more information on this ADL can have a look at [7A].

SADL is a language with a very specific area of use, namely it's a graphical, domain-specific ADL that facilitates the high-level description of real-time avionics systems. It doesn't get much more specific than that. Being that specialised, it's a language that probably won't end up being a competitor of SP, but it's still used as a tool for seeing positive and negative aspects of SP later in this report.

SADL is a graphical language that uses nodes and directed arcs to represent simulation components and their interactions. It's a hierarchical language supporting multiple levels of abstraction. The basic nodes of SADL is shown in illustration 2.33.

Illustration 2.32 – Textual model of the example case

```plaintext
System_hotel_reservation_cs = {
  Component Windows_PC = {
    Port send-request;
    Properties {
      throughput: integer = 100 kbps;
      operating_system: os-id = "Windows XP";
      source-code: external = "/hotelsystem/client.class"
    }
  }
  Component Reservation_system_server = {
    Port receive-request;
    Properties {
      throughput: integer = 1000 kbps;
      operating_system: os-id = "Linux";
      source-code: external = "/hotelsystem/server.class";
      max_concurrent_clients: integer = 100;
    }
  }
  Component Java_RMI = {
    Roles [invoker, executor]
    Properties {
      synchronous: boolean = true;
      protocol: string = "tcp/ip";
      max_roles: integer = 2
    }
    Attachments {
      Windows_PC.send-request to Java_RMI.invoker;
      Reservation_system_server.receive-request to Java_RMI.executor;
    }
  }
}
```
To show a more complete example, illustration 2.34 shows a screenshot from a SADL modelling tool. This is a more understandable model that illustrates better the way SADL is used. The model is of a housekeeping system. I haven’t made a SADL diagram of the example case from chapters 2.2 and 2.3 because this ADL doesn’t apply very well to the hotel reservation case. Hopefully, it’s still possible to get an impression of this language.

3 Comparison of UML and SP
Having spent the first two chapters laying the foundation for this comparison, it’s finally time to actually do it. This chapter will deal with comparing UML and SP using
the framework worked out in chapter 2.4 and the modelled example case from chapters 2.2 and 2.3.

I will start out having a chapter for each of the main categories in my quality framework where I will consider both languages in that quality aspect. Then I will have a chapter on some relevant issues that I feel fall outside the quality framework, but still need mention. Keep in mind that I'm just interested in comparing UML and SP for the deployment and physical view from the 4+1 view model. This report is not about trying to prove SP more valuable in the areas UML handles very well.

Each subchapter dealing with the four main categories of the framework will have a list of positive and negative aspects for each of the two languages. I will rate each aspect (H)igh, (M)edium or (L)ow according to its importance. This is to help focus on the more interesting points instead of getting lost in all the various parts of the evaluation. The three degrees for importance carry the following meaning:

- **High**
  The aspect is very significant to the modelling language’s ability to perform the task the modeller wishes to complete and/or the aspect is something that clearly shows a difference between the languages.

- **Medium**
  The aspect is significant to how well, quality wise, the modeller’s task is completed, but is not crucial to its success. The aspect may also show a difference between the language.

- **Low**
  The aspect should be taken into consideration, but will not significantly hinder or compromise the success of the modelling task. The aspect may show some smaller differences between the languages.

With the quality framework and a method for signifying importance in hand, the comparison may start.

### 3.1 Domain appropriateness

Domain appropriateness was explained in chapter 2.4, so I will get right to the actual evaluation. I will use one chapter for each of the languages' evaluation and try to include as many aspects as possible.

#### 3.1.1 Domain appropriateness for UML

Table 3.1 describes the positive aspects of UML in this category and table 3.2 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Symbol uniqueness</td>
<td>All the statements in the language do have unique representations.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>OO-modelling supported</td>
<td>UML supports the OO-modelling paradigm well since it’s based on objects.</td>
<td>L</td>
</tr>
</tbody>
</table>

Table 3.1 – Positive aspects of UML within the Domain appropriateness category
Table 3.2 – Negative aspects of UML within the Domain appropriateness category

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Flexible in precision</td>
<td>It’s hard to model any vague knowledge. You can to a degree model the facts in the domain, but most developers would probably like to put more information in such a diagram.</td>
<td>M</td>
</tr>
<tr>
<td>-</td>
<td>Expressiveness</td>
<td>The ideal is that the language should be able to express everything in the domain. UML is actually rather limited in this regard. You can only model the tip of the iceberg, so to speak, as UML only allows three layers of the hierarchy to be modelled (two software and one hardware level is the best it can do). And even then some might argue that there are domain properties that don’t fit into the model.</td>
<td>H</td>
</tr>
</tbody>
</table>

3.1.2 Domain appropriateness for SP

Table 3.3 describes the positive aspects of SP in this category and table 3.4 takes care of the negative ones.

Table 3.3 Positive aspects of SP within the Domain appropriateness category

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Symbol uniqueness</td>
<td>All the statements in the language do have unique representations.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>Expressiveness</td>
<td>SP is very flexible within its domain in that it’s possible to add layers to the hierarchy vertically and horizontally indefinitely. You can even add layers of granularity. In other words, it’s possible to model a lot of the domain information in the language.</td>
<td>H</td>
</tr>
</tbody>
</table>

3.2 Participant knowledge appropriateness

This type of quality attributes was explained back in chapter 2.4, so I will give you the evaluation without further ado.
3.2.1 Participant knowledge appropriateness for UML

Table 3.5 describes the positive aspects of UML in this category and table 3.6 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Ambiguity</td>
<td>One symbol doesn’t have several meanings and there aren’t different symbols that mean the same thing.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>Interpretability</td>
<td>Interpreting the UML component and deployment diagrams is rather easy provided you have some basic understanding of software systems.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>Intuitive representation</td>
<td>UML uses pretty intuitive external representation of its entities. Most people who have some knowledge of computer systems can understand what it means that two nodes “Client workstation” and “Server” are connected by connector “LAN”.</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 3.5 Positive aspects of UML within the Participant knowledge appropriateness category

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Clarity</td>
<td>Although the UML diagrams have little ambiguity, there’s a lot of room within the rules for making very different diagrams with respect to organisation of symbols and the exact type of meaning some of them carry (for example, a connector may be both “LAN” or “TCP/P”).</td>
<td>L</td>
</tr>
</tbody>
</table>

Table 3.6 Negative of UML within the Domain Participant knowledge appropriateness category
3.2.2 Participant knowledge appropriateness for SP

Table 3.7 describes the positive aspects of SP in this category and table 3.8 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Ambiguity</td>
<td>One symbol doesn’t have several meanings and there aren’t different symbols that mean the same thing.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>Clarity</td>
<td>The rules are very strict on how the diagrams should be built and all SP diagrams look approximately the same with respect to organisation and there’s little room for varying what the symbols mean on any level.</td>
<td>L</td>
</tr>
</tbody>
</table>

Table 3.7 Positive aspects of SP within the Participant knowledge appropriateness category

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Interpretability</td>
<td>It can be a bit of a challenge to interpret SP diagrams even for people with basic understanding of software systems. There’s a higher threshold of knowledge present to fully grasp the meaning of the diagram.</td>
<td>H</td>
</tr>
<tr>
<td>-</td>
<td>Intuitive representation</td>
<td>SP doesn’t have a very intuitive representation. It can offer some difficulty for those who aren’t familiar with the language to understand why there’s a thick black connector from a box called “GUI controller” to a box called “Client disk” or a dotted line from “Ethernet” to “Client LAN”.</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 3.8 Negative of SP within the Domain Participant knowledge appropriateness category

3.3 Comprehensibility appropriateness

Like the attributes in 3.1 and 3.2, this one was also described back in chapter 2.4, so let’s get into the evaluation process.

3.3.1 Comprehensibility appropriateness for UML

Table 3.9 describes the positive aspects of UML in this category and table 3.10 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Model complexity</td>
<td>UML is very good for simple models.</td>
<td>M</td>
</tr>
<tr>
<td>+</td>
<td>Aesthetic symbols</td>
<td>The various boxes and connectors in UML’s component and deployment</td>
<td>M</td>
</tr>
</tbody>
</table>
The various symbols in the diagram are very different and easy to tell apart.

In UML it’s more common to use visual means such as fonts, size, solidity, colour and so on to emphasize and bring out certain parts of the model. As for the symbols themselves, some are emphasized by being more elaborate than their peers.

UML diagrams in the views considered in this report have symbols that are relatively aesthetically pleasing in composition, with few crossing edges, short edges and little or no redundancy.

The diagrams can quickly get quite crowded since the entities are stacked on top of each other like a pyramid viewed from above. Slightly complex diagrams will be less than pleasing to read.

The UML symbols are a little more complex than necessary. When making complex models, this will hinder comprehension.

SP is better at structuring more complex systems while maintaining readability.

The design of symbols and connectors is pretty straightforward and works well enough. The notation for subnets and multiple instances of an entity is also intuitive.

The symbols in SP can’t get any simpler. This is useful when one has to make complex models. Complex models with complex symbols are harder to read than complex models with simple symbols.
<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Aesthetic symbols</td>
<td>The symbols in SP are just square boxes and the connectors are just simple lines. Not very pleasing to the eye.</td>
<td>M</td>
</tr>
<tr>
<td>-</td>
<td>Symbol discrimination</td>
<td>All the connectors look similar and all the boxes look similar. This might be a source of confusion for people with little skill in modelling and reading models.</td>
<td>M</td>
</tr>
<tr>
<td>-</td>
<td>Visual emphasis</td>
<td>There’s really no clear way to emphasize different parts of the model in SP and no part of the symbol library stands out as more important visually than the rest.</td>
<td>L</td>
</tr>
<tr>
<td>-</td>
<td>Symbol composition</td>
<td>In SP, it’s hard to avoid crossing edges, long edges and some redundancy. This is especially true once the model gains complexity. The nature of the SP models makes it so. Keep in mind that such a complexity is impossible to reach in UML at all, so it’s a bit unfair to drag this into a comparison between the two languages.</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 3.12 Negative of SP within the Domain Comprehensibility appropriateness category

3.4 Technical actor interpretation appropriateness

The last of the four quality categories is Technical actor interpretation appropriateness and is also explained in chapter 2.4. Let’s continue with the evaluation.

3.4.1 Technical actor interpretation appropriateness for UML

Table 3.13 describes the positive aspects of UML in this category and table 3.14 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Good meta-model</td>
<td>The structure and definition of UML is clear and complete.</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 3.13 Positive aspects of UML within the Technical actor interpretation appropriateness category

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Mathematical/operational</td>
<td>UML has no support for purely mathematical and operational rules. There’s no way to enforce or do any calculations on parts of the model.</td>
<td>M</td>
</tr>
<tr>
<td>-</td>
<td>Hierarchy</td>
<td>UML lacks a strong hierarchy. You can only have two software (components and packages) and one hardware layer (nodes).</td>
<td>H</td>
</tr>
</tbody>
</table>
This is very restrictive.

<table>
<thead>
<tr>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>There isn’t a convenient way to hide parts of the UML diagrams. There doesn’t seem to be any good abstraction mechanisms to make a hierarchy in granularity.</td>
<td>H</td>
</tr>
</tbody>
</table>

Table 3.14 Negative of UML within the Technical actor interpretation appropriateness category

### 3.4.2 Technical actor interpretation appropriateness for SP

Table 3.15 describes the positive aspects of UML in this category and table 3.16 takes care of the negative ones.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
<th>Importance</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Hard to find loopholes</td>
<td>The rules of SP cover the language well. It’s hard to find ways to model things SP isn’t made for without breaking these rules.</td>
<td>M</td>
</tr>
<tr>
<td>+</td>
<td>Hierarchy</td>
<td>In SP, you can model as many layers of software and hardware as you please basically. The language is very strong in this regard.</td>
<td>H</td>
</tr>
<tr>
<td>+</td>
<td>Mathematical/operational rules</td>
<td>SP has some mathematical and operational rules connected to it. This is for calculating performance mainly by having matrices filled with numbers or formulas describing operations. You can then use the model and the matrices to calculate various interesting performance evaluation related results. This isn’t the focus of this report, so I won’t get into it – it’s just good to know it’s possible.</td>
<td>M</td>
</tr>
<tr>
<td>+</td>
<td>Encapsulation</td>
<td>SP offers the box surrounded by a dotted line as an abstraction mechanism. As seen in illustration 2.25 and 2.26, it’s possible to just call a box “Network subsystem” and then have a separate diagram showing what it contains/represents.</td>
<td>M</td>
</tr>
</tbody>
</table>

Table 3.15 Positive aspects of SP within the Technical actor interpretation appropriateness category
Table 3.16 Negative of SP within the Technical actor interpretation appropriateness category

3.5 Other things to consider

In this chapter, I will suggest some other things to consider in the UML vs SP debate I have started in this report. The various things discussed in this chapter either falls outside the other four categories or belong to several of them (which makes them hard to place).

3.5.1 Tool support

A pretty important part of the quality of modelling languages falls outside the languages themselves. Tool support is crucial to how much the language will be used by the common developer.

UML has an incredible amount of modelling software available. Some are simple and free while some are very advanced and has a price tag only that make them only suitable for larger organisations. Either way, this has led to a lot of popularity for UML since everyone can easily get their hands on a tool that makes the modelling task a lot easier. The UML tools also usually look good which draws attention from the casual observer looking for a way to solve a modelling problem. Have a look [5W] for some UML tools.

SP, on the other hand, has very little support in modelling tools. There are, in fact, only a few prototype tools specifically made for SP modelling purposes. The problem with these prototypes turned out to be that they were too strict in enforcing SP and didn’t allow any sketching and playing around with the models. Since SP is so simple in its symbolism, however, it’s entirely possible to model the language in any program with simple drawing functions. There is a need, however, for a tool that allows sketching and less “serious” modelling while letting the user know when the SP rules are broken (the program will play the role of a guide rather than an enforcer).

3.5.2 Community acceptance

Another aspect one should look at is how well the language is accepted in developer and academic communities. Influential people supporting a modelling language have
a big effect on users. Especially if these people are known to have expertise in the field.

UML has very much become the language of choice for a lot of developers and is also commonly used to teach students in academic institutions. It fits well with the current popularity of OO-programming (for more on UML, patterns and OO-programming, check out [2B]) and is an easy way in to the information systems modelling field.

SP has had much less widespread popularity as it is somewhat restricted to a few experts in the field. A lot of information on the language is still only in existence in the heads of those using and developing the language, most notably Professor Peter Hughes. It seems paramount that this information soon makes it into articles and instructional books.

3.5.3 Available literature and documentation

A key to using any technology is the literature and documentation one can get hold of to learn and understand that technology. Otherwise, one will have a hard time getting any kind of understanding of whatever one wants to gain use of.

UML is documented in the extreme. OMG, the people behind UML, have a whole lot of information on their site (see [6W]) and any search for UML on most search engines online will give you more hits than you’ll ever bother to go through. The problem with this, as I have experienced while working on this report, is that it’s hard to find a quick and readily understandable fact or answer about the language.

SP has the opposite problem of UML. The documentation available is quite sparse and hard to find. It also has a higher threshold of understanding to grasp. Reading a large UML document or a few pages of SP documentation might take equally much of your time.

3.5.4 Knowledge of the domain

Different languages require different amounts of knowledge of the domain the system is being developed in before you can start modelling. It might be good to have a modelling language that allows the developer to start modelling at an early stage of the system and then expand and fill in details in the model as knowledge of the domain increases.

In the component and deployment diagrams of UML, you seldom have to know that much about the system before making a model. You need a rough idea of how you want the system to look and how it should be distributed, but you don’t really get into the finer details of it.

SP is quite different in that it requires quite a lot of detailed knowledge of the domain and system to be used for modelling. This is a drawback in that it takes time and work to acquire said knowledge, but then again the SP model also offers a lot more back in terms of help in implementing the system, performance evaluation and structure (SP is short for “Structure and Performance”, so this should come as no surprise).

✔ Goal 5: Compare UML and SP.
4 SP compared to other ADLs

After having looked at how SP fits into the 4+1 architectural framework and shown an evaluation of the two languages, we’d like to investigate how SP compares to other ADLs for the tasks I have shown it is suitable for.

4.1 Criteria for comparison

The criteria for the comparison will be grounded in the same quality framework used in chapter 4 and explained in chapter 2.4. The difference will mainly be that because this is not the main part of the report, I will concentrate on just showing some of the more notable positive and negative aspects of the various ADLs as compared to SP.

4.2 Possible ADL choices and criteria for choosing

Finding and choosing ADLs for the purpose of comparing them with SP was very much simplified by using [7W]. I have put all the ADLs listed at the website in table 4.1 for your convenience.

<table>
<thead>
<tr>
<th>#</th>
<th>ADL</th>
<th>Organisation</th>
<th>Leader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ACME</td>
<td>Carnegie Mellon University</td>
<td>David Garlan</td>
</tr>
<tr>
<td>2</td>
<td>AESOP</td>
<td>Carnegie Mellon University</td>
<td>David Garlan</td>
</tr>
<tr>
<td>3</td>
<td>CODE</td>
<td>University of Texas at Austin</td>
<td>J. C. Browne</td>
</tr>
<tr>
<td>4</td>
<td>ControlH &amp; MetaH</td>
<td>Honeywell Technology Center</td>
<td>Steve Vestal</td>
</tr>
<tr>
<td>5</td>
<td>Demeter</td>
<td>Northeastern University</td>
<td>Karl Lieberherr</td>
</tr>
<tr>
<td>6</td>
<td>FR</td>
<td>Ohio State University</td>
<td>B. Chandrasekaran</td>
</tr>
<tr>
<td>7</td>
<td>Gestalt</td>
<td>Siemens Corporate Research, Inc.</td>
<td>Bob Schwanke</td>
</tr>
<tr>
<td>8</td>
<td>Modechart</td>
<td>University of Texas at Austin</td>
<td>Al Mok</td>
</tr>
<tr>
<td>9</td>
<td>Rapide</td>
<td>Stanford University</td>
<td>David Luckham</td>
</tr>
<tr>
<td>10</td>
<td>RESOLVE</td>
<td>Ohio State University</td>
<td>Bruce Weide</td>
</tr>
<tr>
<td>11</td>
<td>SADL</td>
<td>SRI</td>
<td>Mark Moriconi</td>
</tr>
<tr>
<td>12</td>
<td>UML</td>
<td>Rational Software Corporation/OMG</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>UniCon</td>
<td>Carnegie Mellon University</td>
<td>Mary Shaw</td>
</tr>
<tr>
<td>14</td>
<td>Wright</td>
<td>Carnegie Mellon University</td>
<td>David Garlan</td>
</tr>
</tbody>
</table>

Table 4.1 – A list of all the ADLs listed in [7W]

The problem here is that I still have too many ADLs on my hands here to deal with properly in a project such as this where the comparison isn’t really the main part, but rather something extra done to clarify SP’s position among its peers. So let’s propose some criteria for choosing what ADLs to use for the comparison.

1. **Consider ADLs that are in common use.** Comparing SP to some obscure ADL will make it hard for readers of this project to relate to the evaluation and its purpose.

2. **Don’t use an ADL that doesn’t offer any solutions in the Development and Physical view of the 4+1 view model.** Keep in mind that these two views are the only really relevant views in this project.

3. **Don’t compare several ADLs from the same institution.** If one institution develops several ADLs, there’s bound to be one favourite. It’s uncommon for
one company or university to try to market several competing concepts. It’s preferred in this report to go with the most acknowledged ADL from each institution that fits with the other criteria.

4. **Consider software oriented ADLs.** Even though SP takes hardware into consideration, a purely hardware oriented ADL won’t be comparable. To make SP and hardware ADLs comparable, one would have to add information on the actual internal structure of the hardware to SP.

5. **There must be enough documentation available for easy comparison.**

This is both because of the time allotted to this project (hard to learn all existing ADLs in a short period of time) and also since poorly documented ADLs are hard to put to the test.

Criteria 1 and 2 are the most important here. Most developers won’t care if SP is better than some language they’ve never heard of and whether an ADL is better than SP outside the domain I’m using isn’t interesting. The third criteria will help narrow the selection down. The fourth criteria is relevant to limit the comparison to languages that are comparable to SP. The last criteria is mostly important because of the time span for this project.

Because there is a time limit for this project, I will just choose two ADLs from the list. Obviously, I won’t choose UML since it’s already been evaluated in chapter 4.

### 4.3 Choosing ADLs for comparison with SP

In this chapter I will go through a short selection process to find the two ADLs I need for my planned comparison. Using the criteria from chapter 4.2, I will narrow the list from table 4.1 down to the needed languages. [8W] was a big help in finding extra information on some of the languages, the rest was done by hunting down documentation via search engines.

By using the third criteria from chapter 4.2 and excluding UML from the list in table 4.1, I would initially like to look at one language from each of the following institutions:

- Carnegie Mellon University
- University of Texas at Austin
- Honeywell Technology Center
- Northeastern University
- Ohio State University
- Siemens Corporate Research, Inc.
- Stanford University
- SRI

If I were to consider a language from every one of these institutions, I would be stuck with eight languages. So I will have to continue by having a look at the second criteria from chapter 4.2. The slight problem here is that most languages have some part that could be said to work within the Development and Physical view of the 4+1 view model. I will have to restrict myself to the languages that have an obvious relevance for these views.

After having sorted through the documentations, it seems that the following ADLs conform well with criteria number two:
Finding enough documentation to support this proved somewhat difficult, so there may be other ADLs that also may fit into the picture, but are exemplified less or don’t apparently have comparable attributes with SP (meaning that they cover a very different aspect of the architectural design or aren’t visual modelling languages at all).

To make a long story short, I then apply the first and fifth criteria from chapter 4.2 and the resulting two languages that stand out are:

- ACME (known as the Architecture Description Interchange Language)
- SADL (short for Simulation Architecture Description Language)

These two languages pop up a lot in different contexts and seem adequately popular to investigate. They also have available documentation in complete enough forms to enable a general comparison with SP. So the next step is the actual comparisons done in the next chapters.

4.4 ACME compared to SP

To simplify things, the comparison will consist of one table for each of the four quality aspects from 2.4. Each table will be in a separate chapter to add structure and enable addition of some comments without making a mess of the report. The various positive and negative aspects of ACME will all be considered in the context of how well it performs in comparison with SP (as opposed to the UML versus SP evaluation that could stand alone for each of the languages).

Note that since this evaluation is less detailed than the one executed previously in chapter 4, I will only include aspects that qualify as High on importance according to the criteria chosen in the introduction to chapter 4. For more information on ACME, have a look at [4W] and [6A].

4.4.1 ACME Domain appropriateness

In this section, I will attempt to find aspects of Domain appropriateness where ACME is better and worse than SP. This is one of the tougher categories to evaluate since most actively used modelling languages have the basics of this category covered. Table 4.2 shows the evaluation.
4.4.2 ACME Participant knowledge appropriateness

Here I will go into the Participant knowledge appropriateness quality aspects of ACME. Table 4.3 shows the evaluation.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Flexible in precision</td>
<td>ACME offers a lot of ways to add both vague and formal information to the seven entities it consists of.</td>
</tr>
<tr>
<td>-</td>
<td>Expressiveness</td>
<td>ACME, although offering a lot of ways to express knowledge, doesn’t quite have as many ways to expand the model with layers of knowledge.</td>
</tr>
</tbody>
</table>

Table 4.2 – Domain appropriateness of ACME

4.4.3 ACME Comprehensibility appropriateness

I will now use table 4.4 to list the relevant aspects of comprehensibility appropriateness for ACME. This is an easier task than evaluating the other categories since I now deal with more superficial aspects that all modelling languages have to include somehow.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Interpretability</td>
<td>Compared to SP, ACME diagrams are easier to interpret with some basic understanding of software systems and a few minutes of introduction to ACME.</td>
</tr>
<tr>
<td>+</td>
<td>Intuitive representation</td>
<td>ACME diagrams are pretty intuitive to read. The various components and connectors are simple and partly self-explanatory.</td>
</tr>
<tr>
<td>-</td>
<td>Clarity</td>
<td>With flexibility comes some lacking clarity. Since it’s possible to model a lot of different information in the diagrams, they can become less clear than the strict SP diagrams.</td>
</tr>
</tbody>
</table>

Table 4.3 – Participant knowledge appropriateness of ACME

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Aesthetic symbols</td>
<td>The various entities are aesthetically pleasing.</td>
</tr>
<tr>
<td>+</td>
<td>Symbol discrimination</td>
<td>The various symbols in the diagram are very different and easy to tell apart.</td>
</tr>
<tr>
<td>+</td>
<td>Symbol composition</td>
<td>ACME diagrams have symbols that are relatively aesthetically pleasing in composition, with few crossing edges, short edges and little or no redundancy.</td>
</tr>
<tr>
<td>-</td>
<td>Tidiness of diagrams</td>
<td>The ACME diagrams can be packed with a lot of information. Although the textual representations help keep things in order, the model can get a little too full of different pieces of information.</td>
</tr>
</tbody>
</table>

Table 4.4 – Comprehensibilty appropriateness of ACME
4.4.4 ACME Technical actor interpretation appropriateness

Here I will list the relevant aspects of the last category of the quality framework I have chosen, namely Technical actor interpretation appropriateness. This is done in table 4.5.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Good meta-model</td>
<td>The structure and definition of ACME is clear and complete. The documentation on its basis is well written.</td>
</tr>
<tr>
<td>+</td>
<td>Interpretation of rules</td>
<td>ACME is rather easy to understand when it comes to rules. It’s pretty simple to grasp.</td>
</tr>
<tr>
<td>-</td>
<td>Hierarchy</td>
<td>ACME, like UML, doesn’t offer many levels of hierarchy. While it lets you add a lot of information to entities, it’s hard to make a diagram with much depth.</td>
</tr>
<tr>
<td>-</td>
<td>Encapsulation</td>
<td>ACME allows encapsulation to a degree, but doesn’t have SP’s systematic approach to hiding granularity in a hierarchical fashion.</td>
</tr>
</tbody>
</table>

Table 4.5 – Technical actor interpretation appropriateness of ACME

4.5 SADL compared to SP

The comparison of SADL and SP will be executed in the same manner as ACME versus SP in chapter 4.4. So have a look at the introduction to the previous chapter for some general information on the comparison. I will get right to the evaluation here. Note, however, that SADL is a graphical, domain-specific ADL that facilitates the high-level description of real-time avionics systems. So it’s rather specialized and probably isn’t interchangeable with SP in most cases. Still, it can be generally compared to SP in the quality framework. For more information on SADL, have a look at [7A].

4.5.1 SADL Domain appropriateness

I will here describe the positive and negative aspects of SADL within Domain appropriateness in table 4.6. As I mentioned in the introduction

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Flexible in precision</td>
<td>SADL offers at least what seems the necessary amount of ways to model hard and vague knowledge.</td>
</tr>
<tr>
<td>+</td>
<td>Expressiveness</td>
<td>SADL is very good in this aspect considering that it was purposefully built to build systems in its specific domain. It’s hard to beat domain expressiveness of a tool specifically designed for it.</td>
</tr>
</tbody>
</table>

Table 4.6 – Domain appropriateness of SADL
4.5.2 SADL Participant knowledge appropriateness

In this chapter, I will consider SADL’s Participant knowledge appropriateness. It would seem that SADL can, like SP, be somewhat difficult in this category. Perhaps this is due to both languages originally being built for a specific domain which requires some expertise. Anyway, table 4.7 shows the relevant aspects for this category.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Intuitive</td>
<td>SADL diagrams seem essentially pretty intuitive to read because of the simple entities and connectors. There’s also a lot of information stacked onto the models, so it’s not that hard to find out what their various parts represent (at least on an abstract level).</td>
</tr>
<tr>
<td>-</td>
<td>Clarity</td>
<td>SADL diagrams very quickly become quite chaotic to the untrained eye. This may be because of the domain’s systems inherent complexity.</td>
</tr>
<tr>
<td>-</td>
<td>Ambiguity</td>
<td>Some of the symbols are very similar and can be ambiguous. The circle is a frequently used entity in the SADL and it may be a bit hard to remember what’s what.</td>
</tr>
<tr>
<td>-</td>
<td>Interpretability</td>
<td>SADL diagrams are actually a bit harder to interpret than SP models because they are process oriented real-time simulations and as such require a lot of knowledge to interpret.</td>
</tr>
</tbody>
</table>

Table 4.7 – Participant knowledge appropriateness for SADL

4.5.3 SADL Comprehensibility appropriateness

Surprisingly it doesn’t look like SADL is much prettier in appearance and design than SP. Let’s look into the different aspects of Comprehensibility appropriateness and find out if this is at all true. Table 4.8 shows the relevant aspects.
### 4.5.4 SADL Technical actor interpretation appropriateness

In this chapter, I will consider the aspects within the Technical actor interpretation appropriateness. SP and SADL seem very different in this regard. Table 4.9 shows the evaluation.

<table>
<thead>
<tr>
<th>+/-</th>
<th>Aspect</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Good meta-model</td>
<td>The meta-model is pretty well made and reasonably well documented as well.</td>
</tr>
<tr>
<td>-</td>
<td>Interpretation of rules</td>
<td>The rules can be a pretty complex since you have to take real-time systems and their representation into consideration.</td>
</tr>
</tbody>
</table>

Table 4.9 – Technical actor interpretation appropriateness of SADL

✔ Goal 6: Compare SP to other ADLs with respect to the tasks we feel it is better at than UML.

### 5 Evaluation of the results and conclusions

In this chapter, I will look at the main tendencies that can be extracted from the results in chapters 3 and 4. This is very important to create an overview and draw some more general conclusions out of the vast amount of details that you’ve mowed through. This chapter is a merger of evaluation and conclusion since I feel it’s unnatural and cumbersome to draw out information from the work done in the report while postponing the conclusions this information leads up to.
5.1 Rules and meta-model

The basics of SP are pretty well built. From the comparisons between SP and the other three languages, it’s clear that SP is very strict when it comes to rules of the language and how one is supposed to build models in it.

This is a good thing in that it’s relatively easy to stay on track when building diagrams given that you actually have a good understanding of the rules. Modelling things that aren’t supposed to be modelled in SP is hard to do, which is a good thing since it stops modellers from getting themselves into problematic situations.

The downside of SP’s strict rules and rigid way of building models is that it’s hard to get it right and it doesn’t let you gradually increase the detail the model gradually. It forces a very “all-or-nothing” kind of approach which is very dependant on a skilled modeller.

One could wish for a method of softening SP up a bit to give leeway to modellers who need to gradually expand the model in size and detail as their own knowledge of the domain or experience with SP increases. This might be especially interesting since we wish to promote SP as an extension to UML which is a pretty easy and flexible language. Introducing an extension that is in extreme contrast to the rest of UML in difficulty level and use would be hard to do.

5.2 Expressing knowledge

SP is a very good language for expressing hard, factual knowledge. The general image of the language is that it’s meant for systems where you want the essence to be shown without any vague knowledge or unnecessary information on display in the model.

This way of doing things is great when you’re working the systems that SP was originally meant for. When structure and performance are the main issues, you want to keep to the point and add more support for mathematical operations rather than vague knowledge.

The problem with this approach is, naturally, that incorporating SP in UML creates a need to add ways to represent vague knowledge in the diagrams of SP. Modellers will more likely wish to use SP as a replacement for the component and deployment diagrams of UML without really wanting to deal with the performance aspects of it (although, if required, SP has a great way of doing this).

What one might want in this case is a way of adding various vague information in a structured way and create a precedence for this through projects or freestanding examples. Some might argue that you might as well just scribble whatever vague knowledge you want onto the diagrams in an informal manner. This is fine when the modeller is the only one who reads the model, but if it’s supposed to be possible for various actors in the community to exchange models, or even if several modellers on the same project are supposed to work on the same diagrams, it’s important to create a formal and structured way of introducing vague knowledge, so that the reader knows what to expect. This will also help general readability and help adding knowledge and information to the models.
5.3 Dealing with complexity
One of SP’s strong points is its ability to deal with complexity. It takes a very big system or a very unskilled modeller to achieve many crossing lines and confusing entities. SP uses a hierarchy that reaches from higher level software down to lower level hardware if need be. And since it’s possible to use infinitely many layers of abstraction through encapsulation, one can tidy up most models.

The only problem is that getting it all right is hard in SP since it’s a difficult language to learn. Once understood, though, it’s a very nice tool for modelling large system structures.

5.4 Design
SP has a very simple and functional design. As I have mentioned, it consists of three boxes and three connectors essentially. There are no superficial intricacies and it all pretty much looks the same.

The good thing about the current design is that it’s very easy to draw SP models (no drawing or modelling software necessary), the models are very straightforward and they look very clean.

On the other hand, the models in SP are not very pleasing to the eye. Every SP model looks essentially like the another and the symbols are not very descriptive. It might help SP to add a little visual complexity to increase comprehensibility.

It would also be interesting to add simple mechanisms for emphasis or to signal other relevant details for modellers. Sometimes one might want to use a different font or colour to signal something about a part of the system (like that the part isn’t implemented yet, that it’s not changeable or that there’s a problem with it). In short, SP could do with some visual improvements.

5.5 Support for SP
Since SP has been dormant for a while before interest once again has arisen, there’s not much documentation on the language available yet. Considering the vast amount of documentation in the form of books, articles and conference material available for other modelling languages, there should definitely be made an effort to expand the documentation on SP.

There are also no software tools to support modelling with the set of rules in SP. Drawing the symbols is an easy task, but there’s a need for a tool that also enforces the rules and perhaps also offers assistance in the modelling process.

5.6 SP versus UML overall
Even though SP could use improvement in some areas, it’s pretty clear that it has a lot to offer in the development view and physical view of the 4+1 view model. As an extension of UML that replaces the deployment and component diagrams, SP performs much better.

The deciding factors for choosing SP over UML are:

- **Hierarchy and encapsulation**
  SP has an infinitely vertically extendable hierarchy that makes it possible to model a lot more than you can squeeze into the three basic levels in the
corresponding UML diagrams (node, package and component). SP also offers encapsulation to make tidier diagrams where other sub-diagrams are hidden behind levels of abstraction.

- **Set of rules**
  Even though SP’s rules carry a penalty in that they’re hard to grasp, they enforce the way you build a model in the language very well. Once you understand the rules, the models will be correct as well.

- **Expressing the domain**
  To a much larger degree, SP lets you express the properties of the domain. You can flesh the models out with a lot more detail and depth than in the corresponding UML models. Your main hindrance is not in the language, but rather in your own degree of understanding the domain (like your understanding of the rules, SP also requires understanding the domain to a high degree).

- **Visual simplicity**
  Although SP models aren’t pleasing to the eye (at least until you see the beauty of it), they are visually simple, tidy and clear. A modeller in SP really has to make an effort to make a diagram look cluttered and chaotic. In UML, the models have a tendency to quickly look a bit messy.

- **Handling complexity**
  SP handles complexity extremely well. The hierarchy, encapsulation, set of rules, expressiveness and visual simplicity all contribute to make it possible to make very complex models without being penalized with such problems as crossing edges, too much information presented at once, lack of readability, untidiness and so on.

✔ Goal 7: Indicate that SP is in fact better than UML for some parts of the architectural framework.

6 Further work

Even though SP is a potentially very useful tool and has a lot of interesting attributes, it needs to be worked on to make it easier to use for the average system developer. The language needs some work done on it and in this chapter I will propose several possible improvements that should be looked into.

6.1 Guidelines for using SP

SP is, as mentioned several times in this report, quite hard to grasp and use, especially when introducing the language in a new domain of which one has little knowledge. There is a need for establishing guidelines on how to develop SP diagrams as well as best practice further down the line.

Guidelines can be a lot of things. Basically, one needs to get into the SP way of thinking (which is rather unique). I will just suggest some possible avenues for further research in the following subchapters.
6.1.1 Step-by-step approach

Establishing a procedure like an algorithm for how to develop SP diagrams is one way of making a guideline. This would be the modelling equivalent of an instruction manual for putting together a shelf from IKEA.

One can make this algorithm along several axis of the diagram. The following approaches may be used:

1. **Top-down approach:**
   This is a very common way of thinking. You start at the top with the upper-level software and gradually work your way down through the layers until you reach the lower level hardware. This approach would need a lot of helpful description of how to go about adding layers downwards.

2. **Bottom-up approach:**
   Instead of starting at the top such as in the top-down approach, you start by deciding what underlying hardware you want in place and then build layers on top until you get to the top level software. This may be a viable approach for systems where the hardware is the main dependency (which would justify starting at the bottom).

3. **Granularity approach:**
   Instead of starting at the top or bottom, it may be interesting to start by making a model with a few general entities describing the main parts of the system. Then, as you learn more about the domain and understand more about the system, you add details by splitting the boxes into continually less general entities. This is a good approach when you don't yet know that much about the system (which is common when you start a system development project). A problem may be that you early on decide the basic structure of the system – if it proves to be wrong you get to do a lot of extra work.

4. **Sideways approach**
   In this approach you start out by adding nodes to the system as you decide on expansions. You can start out with a workstation node and fill in the needed components, then hook up a web server node and fill in components in it, then add a database server node and so on. This can be very problematic since these units usually depend on one another in some way. However, if you’re building additions to existing systems, you already know the existing specifications and can treat them as static structures that you add nodes to. Like if you have a database server up and running, you might want to add a new application through a web server and a client workstation.

Keep in mind that these ways of doing it are only suggestions. Researching these methods is outside the scope of this project.

6.1.2 Process oriented development with SP

A big advantage of UML and its popularity is that it’s part of several processes for developing software systems. It’s also process oriented internally as it’s normal to start out with a diagram like the use-case diagram at the beginning of the project and
then continually develop the other diagrams as you add more aspects to the system. SP could benefit a lot from adding process both externally and internally.

Adding process externally would involve making SP a natural part of a more general software development process. This would involve having SP models being used to accomplish some goal in the process. The advantage for SP would be to have a generally accepted purpose and prerequisites that have to be in place before one can make a model in the language. More research has to be done on what purpose SP should fill and what goals it achieves and prerequisites it needs in its new context explored in this report (there’s been plenty of research on this in its original context of Performance Evaluation).

Adding process internally would mean making SP itself process oriented. In this report, I have concentrated on showing that SP is better than some parts of UML. The issue is extending UML with SP. A big advantage would be to enable SP diagrams to be built partly using other diagrams, so that it naturally evolved through the modelling process. So by extending UML with SP, you can get benefits for SP as well. More research has to be done on how to actually make it all work together, though. Developing methods for making SP models from other diagrams is a demanding task in itself.

6.1.3 Building SP models from examples

A way to make a quick and useful method of building SP diagrams would be to create a library of examples for various common problems and add guidelines for how to expand and modify these examples to fit whatever specific problem the developer wants to solve. A lot of systems in the field of software development are similar, so it should be possible to make generic examples for the various categories of systems.

This won’t help those trying to make unique or very complex systems to the same degree, but will give useful hints and help in the learning process of understanding SP. The cookbook way of making programs (just putting examples together without having to do too much thinking and without having to have full understanding of the tool you’re using) is often used as well and can be of help to make SP more usable.

6.2 Concept of nodes

A way of making SP diagrams easier to read and also possibly partly solving the problems of adding vague knowledge to the diagrams presented in chapter 5.2 could be adding nodes. The use of nodes is seen in the UML deployment diagram and something similar can be introduced in SP.

A node in the UML diagrams is usually a hardware unit of some size, like a server or a workstation. An SP model usually spans over several such nodes implicitly, but the nodes themselves aren’t drawn. Explicitly showing these nodes will partition the diagram visually, so that it might be easier to read. An example is shown in illustration 6.1 (this is just a simplified variation with nodes of the diagram in illustration 2.25).
Vague knowledge is usually something a modeller uses at higher levels of abstraction. It’s less common for a modeller of a system to introduce vague knowledge to a box called “CPU”. So a possible way of introducing vague knowledge could be to add this to the nodes. And example is shown in illustration 6.2.

If the need arises, one might want to also add something similar to the individual entities and their connectors. Keep in mind that these are just thought experiments to suggest some possibilities of further development of SP.
6.3 Design improvements for more intuitive use

Some of the more ascetic modellers might claim that what a diagram looks like aesthetically and how neat it is, is rather unimportant. However, superficial aspects of a modelling language are very important. Mechanisms for emphasizing parts of the model, the ADL’s general design in symbols and similar properties can be crucial in some ways. Without good reason, few people would like to use a modelling language that looks very difficult to learn and has a horrific design. The language might, quite literally, scare the potential modeller away.

SP is not really that bad when it comes to how it looks and feels, but it’s not that great either. Although the boxes and their connectors are an exercise in visual simplicity, one might benefit from introducing a little more in the way of aesthetics as well as some mechanisms to bring out important parts of the model.

To introduce more different symbols, one ultimately has to decide what the differing factor between symbols should be. Is it interesting to differ between hardware and software, between different types of software and hardware, between local and remote entities or maybe the differing factor should be whose responsibility that part of the system is? Deciding what to go with will have to be a future task, but an example is shown in illustration 6.3 where I’ve chosen to go with differing between various types of hardware. I will use the basic example from chapter 6.2 as a basis for this example.

Illustration 6.3 – An example of design expansions for SP

How to do the design expansions will, of course, be something that has to be worked on and maybe one has to confer with people who know proper design. One might argue that if people want to make different symbols for SP like in illustration 6.3 and introduce mechanisms to emphasize parts of models, they can come up with whatever suits them. The problem with this is that the further you travel away from the accepted standard, the harder it is to exchange models between developers and other actors. Difficulties will arise when symbols appear that some modellers haven’t seen before. Or maybe some system developers will use bold red font to signal a possible problem in the system while others use the same font to signal a part that is not to be tampered with (i.e. a “frozen” part of the system during development).
6.4 Flexibility in rules
As mentioned a lot of times in this report, the SP rules are both hard to fully grasp and very strict. The modeller isn’t allowed a lot of freedom to experiment or model domain properties not specifically allowed by the SP set of rules.

Having guidelines to allow modellers to make their own expansions and take a few liberties can be a good thing if done right. How this is done has to be carefully considered since allowing too much without some control will render the modelling language pretty meaningless.

What one must consider is what kind of flexibility is relevant and interesting. Deciding on this will probably depend a lot on the purpose of allowing such flexibility. Plausible purposes may include:

- **Make up for uncertainty**
  In a software development process, the developers learn about the system they’re making as it’s being developed. So at first they may not know enough to make a detailed SP model. Maybe they’re unsure of how the system will be partitioned in the model or what connectors will go where. Introducing symbols that signify temporary entities and connectors might be interesting here.

- **Expand for unforeseen needs**
  It’s hard to know exactly how modellers work and what they want to use a given modelling language for. A need for unforeseen expansions might arise to accommodate a modeller’s method of design or modelling various parts of the domain SP wasn’t originally meant for.

- **Lighten the SP modelling process**
  Since SP is a difficult language to model in, it might be interesting to help the process of modelling by starting out with a less strict set of rules and then add rules as you go along to make it incrementally more correct. If this just makes the modelling process more confusing or helps remains to be seen.

I’m sure there are many more viable purposes for adding flexibility to the SP rules, but the listed examples give an idea of what may warrant it. I think the important part here is not to try to take into account every purpose possible, but instead make some guidelines for how flexibility should be handled. This sort of thing has been done in other languages in varying degrees of formality and should be just as possible to do for SP.

6.5 Tools and documentation
A possible improvement that hasn’t really got anything to do with the language and its development, is tool support and documentation. As mentioned in chapter 3.5.1, tools for SP are limited and in 3.5.3 I talk about its lack in documentation.

When it comes to tool support, the easiest way to try to improve the situation may be to add SP rules to an existing modelling tool like MS Visio. MS Visio is structured around stencils that contain the actual visual entities of the language along with the rules governing them. It should be a relatively simple feat to add an SP stencil and maybe later on build a stand-alone tool if the stencil proves usable and interesting.
As for documentation, there simply is very little of it. Creating more articles on the various topics surrounding the SP language is a step in the right direction. Especially articles on the challenges mentioned in this report would be a great addition to the so far small library of available documents.

✔ Goal 8: Suggest modifications and further work for SP.
Appendix A: References/bibliography

This appendix shows all the background material I have used for this project. I have divided my sources into books, articles and websites for convenience.

A.1 Books

[1B] Information Systems Engineering – Conceptual modelling in a quality perspective
   by John Krogstie and Arne Sølvberg

   by Craig Larman

A.2 Articles

[1A] “SP Principles”
   by Peter Hughes
   STC Technology Ltd. 1988

[2A] “Quality of Interactive Models”
   by John Krogstie and Håvard D. Jørgensen
   IWCQM 2002, Tampere, Finland
   Norwegian University of Science and Technology,
   Institute of Computer and Information Sciences and
   SINTEF Telecom and Informatics, Norway.

[3A] “The 4+1 View Model of Architecture”
   by Philippe Kruchten
   Rational Software Corp.

[4A] “Project based Software Architecture Teaching”
   by L. Jaccheri and M. Torchiano
   IDI, NTNU 2002
   Technical report

[5A] SP (Rules)
   by Gunnar Brataas, Post Doc.
   IDI, NTNU 8th of April, 2003
   (Lecture slides)

[6A] “ACME: An Architecture Description Interchange Language”
   by David Garlan, Robert Monroe and David Wile
   January 14, 1997

[7A] “SADL: Simulation Architecture Description Language”
   by Kenneth G. Ricks (National Aeronautics and Space Administration, Marshall Space Flight Center),
   John M. Weir (National Aeronautics and Space Administration, Marshall Space Flight Center) and B. Earl Wells (Department
A.3 Websites

Checked: August 2003
A tutorial on UML

Checked: August 2003
A tutorial on Rational Rose Object Oriented Design Tool
by Bill Brooks

Checked: August 2003
The MSDN Library – Chapter 3

Checked: October 2003
The ACME Homepage

Checked: October 2003
My Little UML Page

Checked: October 2003
UML Home Page

Checked: August 2003
A list of Architecture Description Languages
provided by Carnegie Mellon Software Engineering Institute

Checked: October 2003
Software Architecture People