A Study in Formal Specifications
With Application to Price Calculations in the Power Market

Maria Gramner
Spring 2015
Bachelor’s thesis, 15 Credits
Supervisor: Jerry Eriksson
External supervisors: Jonas Lundin and Petter Svärd
Examiner: Pedher Johansson
Bachelor’s program in Computing Science, 180 Credits
Abstract
The world today is getting more and more dependent on digital technique, and errors in software can be expensive or hard to fix. Because of this, it is more important than ever to have a reliable way of making sure that developed programs work correctly. This thesis will give an introduction to formal specifications and examine the possible usefulness of one for a price calculating program concerning electricity contracts. Some background information on formal specifications will be presented along with a description of some of the different available techniques for writing one. Furthermore, a model-based specification in the language Z will be developed, and the study will show that this approach to program correctness can be useful for, among other things, programs for the financial domain.
Acknowledgments

I would like to thank my supervisors from Hi5, Jonas Lundin and Petter Svärd, for having the patience to answer all my questions, both on the electricity contracts and how to present my results in this thesis in a proper way. I would also like to thank them for believing in me when I came up with the idea of writing a formal specification, even though that was not the original purpose of the assignment I was given. I would also like to extend a thank you to Magnus Kjellin for his enthusiasm on my work, which brought some extra joy to the writing process and inspired me to do my very best.

Furthermore, I would like to thank my supervisor Jerry Eriksson for his guidance during the work, and also Pedher Johansson for his help in figuring out what kind of project I should take on.

Last but not least, I am forever grateful for my family and their never ending love, support, and encouraging words, especially my wonderful boyfriend Marcus who never fails to brighten my days with laughter, warmth, or food, whenever my spirit is low.
1 Introduction
The world today is getting more and more dependent on digital technique, and the software solutions are getting more complex every day. Errors in software can have devastating consequences or be very costly to fix, and as such it gets very important to make sure that a developed software product works correctly. There are different approaches to solving this problem, and one approach is to write a formal specification before starting the implementation. This thesis will give an introduction to the world of formality, and present an example in the form of a specification for a price calculation program.

1.1 Purpose and Goals
The formation of this project started when the company 5 High Innovations AB (Hi5) wanted a system for performing price calculations related to something called Power DS Future contracts, which are contracts for purchasing electricity.

Since the problem domain is in finances, it is important that the calculations are performed correctly and that a verification of the procedure is easy to perform. Here, the formal specification is introduced. A formal specification is an abstract description of what a program should do, written in a mathematical language to make correctness proofs possible before the system is fully built. They are used as a way of verifying that a program works correctly.

The goal of this thesis is thus to provide answers to the following questions.

- Could a formal specification be useful as a foundation for a system that should perform price calculations related to electricity contracts?
- If so, what type of formal specification would be the best choice and why?

1.2 How to Present a Conclusion
A chapter with a brief introduction to formal specifications will be presented in the thesis, with information on the background and possible benefits or drawbacks of using one, so as to be able to make a conclusion on the first question.

Furthermore, in order to provide an answer to the second question, different techniques for writing a formal specification will be presented and matched against the criteria for the wanted price calculating system, provided through information on the electricity contracts in question and a description of what the intended system should do. The choice of specification technique will then result in a developed formal specification.

1.3 Related Work
There have been quite a lot of research done on the matter of formal specifications, with different approaches. To mention some, there is research on how to combine formal specification and testing [12], and research on combining formal specification and agile development [19].

For those who are interested in learning more about the foundations of the formal approach to computer program correctness there is a lot that can be read. To give some examples; Hoare’s “An Axiomatic Basis for Computer Programming” [13], and Dijkstra’s “Notes on Structured Programming” [5].
2 Formal Specifications

This chapter will give a description of what a formal specification is and what they are used for. It will also give a brief introduction to some of the different techniques and languages in which a specification can be made.

2.1 What is a Formal Specification?

A formal specification is a specification written in a mathematically based language with the purpose of aiding in the construction of software or hardware systems. It can be used to describe and analyze a system, and help find possible errors before the system is fully implemented. Furthermore, it can be used to help the development process by program derivation, and be of use in testing of the system [12, 9].

In order to claim that a specification is formal, it needs to be written in a language that has syntax, semantics, and rules for inferring useful information [15]. This formality is often based on discrete mathematics such as logic, set theory and algebra [21].

2.2 Why are Formal Specifications used?

As stated by Lamsweerde [15]:

“Formality helps in obtaining higher-quality specifications”

Creating a formal specification can help raise questions and detect serious issues in informally given information on what a system should do. A simple example is given by Cohen [3] with the use of the Hoare Triple

\[
\{Q\} S \{x = y\}
\]

which states that if \(Q\) is satisfied, the program \(S\) will terminate with \(x = y\) satisfied. The problem here is that there is no indication on how \(x = y\) is to be achieved. Should it be done by setting \(x\) to the value of \(y\), setting \(y\) to the value of \(x\), or by giving both \(x\) and \(y\) the same arbitrary value? These are the type of questions that a formal specification can help answer.

Although a formal specification can be executable, it is not the main purpose of writing one. Instead, one of the main reasons for constructing a formal specification is the possibility to perform proofs. As described by Gaudel [9], these proofs can be used to validate a specification, verify that a design step is correct, or check that a program satisfies a specification. Proofs of this kind can be conducted at a low level, “by hand” in the style of classic reasoning in mathematics, at an intermediate level, using logic partially supported by tools, or at a high level where formal proofs are performed mostly with the help of a theorem prover or proof checker.

Formal specifications are currently more commonly used in the hardware industry than in software development, but if they are used it is often for systems such as hospital equipment, where it is crucial the system should work properly, or in the development of products that would bring high costs if they failed [12, 21].

---

1 A Hoare Triple is a feature in Hoare Logic which is a formal system used for reasoning about the correctness of computer programs [13].
2.3 Are There Limitations to Formal Specifications?
As stated in the previous section, formal specifications are not as common in the software business as it is in the hardware industry. Some of the reasons for why that may be are presented in the following list.

- The time needed to produce a system using formal specifications does not play well with the market of today where it is often more important to produce software quickly and get it sold. Some customers think the value of getting a product at a fast rate is higher than the value of it not having faults [21].
- They can be hard to develop completely correct without expertise in formal systems and mathematical logic. This is often due to subtle errors that are hard to discover, and the wide range of choices that can be made in the design. It has been noted though, that even slightly incorrect formal specifications can be helpful in making an informal specification less ambiguous and thus capture problems before they are implemented and possibly harder to correct [15, 21].
- Many software companies use agile development processes and the construction of a formal specification is considered to clash with this. There are, however, suggestions to how the two can be combined, like for instance with the use of executable specifications as described by Nummenmaa et al. [19].
- Formal specifications are best suited for smaller software systems. When a system increases in size, the time and effort needed to make a specification on all of it grows disproportionately [21].
- Due to the abstraction and notation used in the specifications, they may not be understood by everyone that has an interest in the project, such as clients, architects, domain experts, and developers [15, 19].

In spite of these limitations, it has been reported that companies that have invested in formal methods have had fewer errors in their developed products without noticing an increase in the development cost [21].

2.4 Different Types of Formal Specifications
There exists a wide variety of techniques to use when writing a formal specification, some are for a more general purpose and some fit better for specific types of applications. The following sections will give an introduction to some of the different techniques, explained through a notation language within that technique.

2.4.1 Model-Based Languages
One approach to formal specifications is to build a model of the intended system by describing the different states the system could be in and the operations that will change the state. The states are often described with sets, sequences, relations, and functions, and the operations with predicates in terms of pre- and post-conditions. Examples of model-based languages are Z, VDM (Vienna Development Method), and B [12, 21].

Model-based languages are good for describing a general system, and they could possibly have an infinite number of states. This, however, could make it less suitable for testing by automation without the use of powerful analysis tools [12]. On the other hand, since the basis is often more or less first order logic, like in Z, it should be possible to perform “low level proofs” on the model by mathematical reasoning [9].
An example of how a model-based specification can look is illustrated below with the use of the language Z, which makes use of Schemas with declarations, and predicates that constrain them. One advantage of using Z is that it was developed to be easy to read by using informal text combined with formal descriptions [21]. This particular example of a list of students is inspired by an example of a stack model by Hierons et al. [12].

Model of a system with a list of students:

First, if the list should contain elements of the type Students, the type needs to be defined. In this state it does not matter how the structure of Student would be implemented, it is simply defined in order to be able to use it in the rest of the specification. Also, a max size for the list can be defined.

\[
[\text{Student}]
\]
\[
\text{maxSize} = 45
\]

A schema can then be written as below, where registered is declared to be a sequence of Students, with a length that does not exceed the previously defined allowed size.

\[
\begin{align*}
\text{StudentList} & : \text{seq Student} \\
\#\text{registered} & \leq \text{maxSize}
\end{align*}
\]

An initialization schema can then be given, which shows that after the start, registered will be empty. This is shown with the prime symbol ‘, which denotes an after state of a component.

\[
\begin{align*}
\text{StudentList_Init} & : \\
\text{StudentList} & \\
\text{registered}' & = \langle \rangle
\end{align*}
\]

An operation to add a student to the list can be given as below, where the symbol Δ means that this operation will change the state of the list, the symbol ‘? denotes input, and again the prime symbol is used to show that after this operation, registered will contain the student given as input.

\[
\begin{align*}
\text{AddStudent} & : \\
\Delta \text{StudentList} & \\
\text{name'?} : \text{Student} & \\
\#\text{registered} & < \text{maxSize} \\
\text{registered}' & = \langle \text{name'?} \rangle \setminus \text{registered}
\end{align*}
\]

This is a very simple, and not full, specification of a list of students, but it illustrates the underlying principle of the technique.
2.4.2 Transition-Based Languages

Transition-based techniques are also somewhat state based, but the focus lies more on the transitions that change the states rather than the states themselves. Properties are defined with a set of transition functions, which for each input state and triggering event gives the corresponding output state [15]. Examples of transition-based languages are FSMs (Finite State Machines), Statecharts and X-machines [12].

These types of specifications are often shown in a graphic way with states and arrows denoting the transitions between those states. A simple FSM example of a traffic light might look something like in Figure 1, where the transitions represent the signals that change the state of the lights.

![Traffic Light FSM](image)

**Figure 1.** An FSM representation of a traffic light with the transitions that change which light is on.

Transition-based languages can be suitable for illustrating control systems such as traffic lights or vending machines, but the lack of expressiveness and the state explosion problem\(^2\) in for example FSMs might be a limitation to some system descriptions [20, 10].

2.4.3 Algebraic Languages

Algebraic languages are used to describe a system in terms of its algebraic properties. These languages are suited for describing abstract data types in a way that is not dependent on implementation [9, 21]. Examples of algebraic languages are OBJ, CASL (Common Algebraic Specification Language), and Larch [12, 21].

As described by Hierons *et al.* [12], in mathematical terms, algebra is said to have a set of symbols that denotes values of some type, and operations on that set. For algebraic specification languages the syntax for the rules of the operations is described with a signature for each operation that gives the domain of the input and output. The semantics are described in terms of axioms, or equations, which describe the desired properties. They may also be constricted by conditions.

Even though writing a specification is about defining a problem rather than to program the solution, Lamsweree [15] suggests that algebraic specifications tend to be harder to perform correctly without any experience in coding. On the other hand, one advantage of using an algebraic technique is the possibility to perform easy rewriting of terms, which can make testing easier [12].

An example of an algebraic specification is given on the next page in the language CASL, where a list of students is specified.

---

\(^2\) State explosion problem is the problem that occurs when the number of states a system can have increases disproportionately, which makes a representation of it difficult to define [14].
Example of a list of students:

```plaintext
from Basic/Numbers get Nat
spec StudentList =
then sort list
ops create: list
    add: list * Student → list
    remove: list * Student → list
    length: list → Nat
vars ls: list; st: Student
axioms
    length(create) = 0
    length(add(ls, st)) = length(ls) + 1
    length(remove(ls, st)) = length(ls) - 1
end
```

Assuming that `Student` is previously specified, a specification of `StudentList` might look something like the above example. `ops` defines the operations that can be performed, where `create` returns an empty list, `add` and `remove` takes a list and a student as arguments and returns a list with the student added or removed, respectively, and `length` takes a list as argument and returns the length as a natural number. The `axioms` here define what the length of the list would be when combined with the different operations that change the list.

### 2.4.4 Other Languages

There are, of course, specification languages that are not mentioned or described in detail in this paper. The reason for that is either that it was deemed too big of a task to go in to detail on all of them, or that some techniques are more suitable for another type of system than the one described in Chapter 4. Below is a short description of concurrent based techniques and hybrid languages, which fall in the latter category.

Concurrent languages are useful for describing systems with a number of concurrent processes [9, 15, 12]. Some transition based languages can also be used for multiprocess systems, but, in general, process specific languages have a richer theory where one advantage is that it is easier to compare and see equivalence between two specifications on the same system [12]. Examples of concurrent based languages are CSP (Communicating Sequential Processes), CCS (Calculus of Communicating Systems), and LOTOS (Language Of Temporal Ordering Specification) [12, 9].

Hybrid languages are mostly suitable for hybrid systems which consists of both digital and analog components that interact with each other, where the analog component is in most cases different types of mathematics. Robot systems, aircrafts or temperature controllers are examples of systems that can be of hybrid design, and one example of a hybrid specification language is CHARON [12, 1, 16].
3 Power DS Future Contracts

When a company plans to purchase electricity, they can make use of something called Power DS Future Contracts. These contracts are good for budget planning since they give the company a chance of knowing the price they will pay for the electricity in the future. This chapter will give a brief description of the contracts and the calculations that are needed to present the final electricity cost.

3.1 What are Future Contracts?

According to Investopedia [7], a futures contract is defined as

A contractual agreement, generally made on the trading floor of a futures exchange, to buy or sell a particular commodity or financial instrument at a pre-determined price in the future.

They are primarily used for hedging risks in price changes or for taking advantage of the movement in price in order to make a beneficial transaction [8].

Explained in simpler words, a futures contract is a contract written beforehand where a seller and a buyer agree on a fixed price for an amount of some product, for a period of time in the future, regardless of what the product might actually be worth during that period. If the actual market value is higher during the delivery period, the seller made a loss on the contract, and if the value is lower than the contractual price, the buyer made a loss.

3.2 Nordic Electricity DS Futures

The electricity contracts subject to the calculation system in Chapter 4 are Nordic Power DS Future contracts from Nasdaq. There are month-, quarter-, and year-contracts for both base- and peak loads, and the contract bases are Nordic System Price, or Peak Nordic System Price, as according to the “Elspot System Price” published by Nord Pool AS [17, 18, 4]. The system described in Chapter 4 will only deal with base load contracts, and full contract specifications can be found at [4].

The different month-, quarter-, and year-contracts are often combined in something called a Strategy, where the planned electricity purchase is distributed over several different contracts.

3.3 Required Calculations

During the delivery period for the contracts, the calculated payment for each day is based on the market value (the Spot price), during that day. Since a buyer should ultimately pay the contractual price and not the actual current value, a Spot Reference Settlement is made. This means that if the market value is higher than the contractual price, the difference is subtracted from the payment, and if the market value is lower than the agreed price, the difference is added to the payment [4].

---

3 Hedging is defined as making an investment to reduce the risk of losing money due to market fluctuations [11].
4 Base load contracts are delivered Mon-Sun 00.00-24.00 as opposed to peak load contracts that are delivered Mon-Fri 08.00-20.00, when the electricity consumption tends to be higher [6].
A simplified example:

Jodi has a futures contract which says she can buy 2MW for 10 EUR/MW.

When the delivery day arrives, the market value is 15 EUR/MW.

This means that for those 2MW that she bought, Jodi would originally be billed 30 EUR, but since she had a futures contract, a Spot Reference Settlement is made. Since her contract says that she is only required to pay 20 EUR, 10 EUR is subtracted from the bill.

If on the other hand, the market value had been 8 EUR/MW, the original bill would be for 16 EUR. Since Jodi is obliged to pay the contract value, 4 EUR is added to the bill.

If a buyer purchases more electricity than defined in the contracts he owns, the Spot Reference Settlement is only made on the contractual amount. The excess electricity is purchased for the current market value.

There are also other calculations that can be needed in order to get a total cost of a strategy, but those include different types of contracts and currency conversions. The system described in Chapter 4 will only deal with Spot Reference Settlements of system price contracts in Euro.
4 A Formal Specification for Calculation Software

This chapter will present the formal specification that was developed in reference to the informally given criteria on what the software should do. It will also mention some criteria that needed to be more specified in order to perform the formal specification accordingly.

4.1 The Informal Specification

The company Hi5 wanted a software system for calculating electricity cost in reference to the Power DS Future contracts mentioned in Chapter 3. The most important criteria was that the system should be simple and easy to understand, and that it should be possible to verify that the computed results are correct.

Thus, the initial list of criteria for the system consisted of the following elements.

- Should be simple.
- Should be transparent.
- Should be verifiable.
- Should calculate the cost for physical power consumption.
- Should calculate Spot Reference Settlements (in the specification called hedge settlements).
- Should only handle contracts in Euro.
- Should only handle base load contracts.

During the development of the formal specification, questions arose regarding input, output, and time detail for the calculations, among other things. The list of criteria was thus expanded with the following list.

- A user should be able to give input on what time period and what strategy the system should calculate on.
- The output should be in the form of tables: one for physical consumption, one for Power DS Future contracts (in the specification called hedge contracts), and one for the total cost of each date.
- The calculations should be performed for every hour in the wanted time period. Thus, anywhere in the specification where a date is referenced, it should contain both the date and hour.

4.2 The Formal Specification

The chosen specification technique and language for the calculation system is a model-based specification in the language Z. It was chosen primarily for its expressiveness, and combination of informal language and formal descriptions, and also due to the fact that the other techniques described in Chapter 2 are best suited for other types of programs.

Instructions on how to develop a Z specification can be found in “Mathematical Logic for Computer Science” by Ben-Ari [2], and the Z reference manual by Spivey [22].

The first things that are defined in the specification are the types that will be used in the schemas. The four types in brackets are types whose structures are unimportant for the specification. This means that, for example, the exact form of DATE is not important at the moment. The following eight type definitions are defined to take on values of the type $\mathbb{Q}$. This means the value is a rational number. Lastly, the three types PHYSSETTLROW, HEDGESETTLROW and TOTALCOSTROW are types whose values are made up of ordered
pairs of other types. In simpler words, a TOTALCOSTROW, for instance, will contain an INDID, a DATE, and a TOTCOST in that exact order.

\[
\begin{align*}
[\text{INDID}, \text{DATE}, \text{AREAID}, \text{PRODID}] \\
\text{PHYSVOL} & := Q \\
\text{AREAPRICE} & := Q \\
\text{PHYSMKT} & := Q \\
\text{HEDGVol} & := Q \\
\text{HEDGEPRICE} & := Q \\
\text{SYSPRICE} & := Q \\
\text{HSTTLMNT} & := Q \\
\text{TOTCOST} & := Q \\
\text{PHYSSETTLROW} & := \text{null} | n \langle \text{INDID} \times \text{DATE} \times \text{PHYSVOL} \times \text{AREAID} \\
& \times \text{AREAPRICE} \times \text{PHYSMKT} \rangle \\
\text{HEDGESETTLROW} & := \text{null} | n \langle \text{INDID} \times \text{DATE} \times \text{PRODID} \times \text{HEDGVol} \\
& \times \text{HEDGEPRICE} \times \text{SYSPRICE} \times \text{HSTTLMNT} \rangle \\
\text{TOTALCOSTROW} & := \text{null} | n \langle \text{INDID} \times \text{DATE} \times \text{TOTCOST} \rangle
\end{align*}
\]

The schema CostCalculator is the state space of the system. It contains three variables named physTable, hedgeTable and totalTable, which can all contain sequences of previously defined types. One can say that each table will have “rows” of their corresponding ordered-pairs type. The variables strategy, startDate, and endDate will be used for defining what strategy and time period the system should calculate on. The remaining variables are only used by the program to keep track of the progress.

```
CostCalculator
| physTable: seq PHYSSETTLROW |
| hedgeTable: seq HEDGESETTLROW |
| totalTable: seq TOTALCOSTROW |
| strategy: INDID |
| startDate: DATE |
| endDate: DATE |
| prevPhysDate: DATE |
| prevHedgeDate: DATE |
| physTableCounter: \mathbb{N} |
| hedgeTableCounter: \mathbb{N} |
| totalTableCounter: \mathbb{N} |
| physPrintCounter: \mathbb{N} |
| hedgePrintCounter: \mathbb{N} |
| totalPrintCounter: \mathbb{N} |
```

The schema CostCalcInit on the next page shows what the values of every variable will be after the start, or initialization, of the program. The sequences will be empty and the strategy and time period will not yet be defined. The progress variables will be in start positions.
UserInput defines an operator that will change the state of the program CostCalculator. This operation can only be made if the variables strategy, startDate, and endDate are empty. The schema takes input on which strategy to calculate on, and what the start- and end-date are for the time period of interest. The input is stored in the corresponding variables in the program.

The operation PhysCalcInput on the following page is used for storing, and calculating on, information on the physical energy consumption of a specific date. It explains that a variable called VP1 is to be created, and that it will be of the type PHYSCOST and be given the value of the formula (volume * price). A variable named PR1 of type PHYSSETTLROW will also be created and then contain the indId, date, volume, areaId and price given as input. It will also contain VP1. PR1 will then be stored in physTable. The operation is repeated for every new row one wish to store in the system.
The schema also contains conditions used for making sure that the given input is valid in accordance to the wanted strategy and time period, and for setting physTable in chronological order.

```
PhysCalcInput

\(\text{ΔCostCalculator} \)

\(\text{indId}\): \text{INDID}

\(\text{date}\): \text{DATE}

\(\text{volume}\): \text{PHYSVOL}

\(\text{areaId}\): \text{AREAID}

\(\text{price}\): \text{AREAPRICE}

\(\text{startDate} \leq \text{date} \text{?} \leq \text{endDate} \land (\text{prevPhysDate} = \text{null} \lor \text{prevPhysDate} \leq \text{date}?) \land \text{indId} = \text{strategy}

\exists \text{VP1} : \text{PHYSCOST} \bullet \text{VP1} = (\text{volume}? \ast \text{price}?) \land \exists \text{PR1} : \text{PHYSSETTLROW} \bullet \text{PR1} = n(\text{indId}?, \text{date}?, \text{volume}?, \text{areaId}?, \text{price}?, \text{VP1}) \land \text{physTable}' = \text{physTable} \sim (\text{PR1})

\text{prevPhysDate}' = \text{date}?
```

The schema \textit{HedgeCalcInput} on the following page describes how to store, and calculate on, information on a hedge contract for a specific date. It describes that a variable called HVOL1 of type HEDGEVOL is to be created. HVOL1 will receive the value for the contract volume per hour, instead of the volume for the entire contract period. Furthermore, a variable called STTL1 of type HSTTLMNT is also created and is given the value of the hedge settlement that is calculated by the formula \((\text{HVOL1} \ast \text{hedgePrice} - \text{HVOL1} \ast \text{spotPrice})\).

Lastly, the variable H1 is created. This variable is of the type HEDGESETTLROW and will contain the indId, date, contractName, hedgePrice and spotPrice given as input, along with the created variables HVOL1 and STTL1. H1 is then stored in hedgeTable.

The schema contains the same conditions as \textit{PhysCalcInput}, and can also be repeated for every new hedge contract row one wish to store.
HedgeCalcInput

ΔCostCalculator

indId?: INDID

date?: DATE

contractName?: PRODID

contractVol?: Q

nrOfHoursInContract?: N

hedgePrice?: HEDGEPRICE

spotPrice?: SYSPRICE

startDate ≤ date? ≤ endDate ∧ (prevHedgeDate = null ∨ prevHedgeDate ≤ date?) ∧ indId? = strategy

∃HVOL1 : HEDGEVOL • HVOL1 = (contractVol? ÷ nrOfHoursInContract?) ∧

∃STTL1 : HSTTLMNT • STTL1 = (HVOL1 * hedgePrice? – HVOL1 * spotPrice?) ∧

∃H1 : HEDGESETTLROW • H1 = n(indId?, date?, contractName?, HVOL1, hedgePrice?, spotPrice?, STTL1) ∧

hedgeTable' = hedgeTable ~ (H1)

prevHedgeDate' = date?

TotalCalc is used for starting the total cost calculation of a specific date for which there are data in the tables for physical consumption and hedge contracts. If TotalCalc finds a date in physTable that has not been added, and thus calculated, to the table for total cost, it creates a variable named C1 of the type TOTCOST with the initial value of 0. It then creates the variable TCR1 of type TOTALCOSTROW and gives it the previously defined IndId, the Date that is to be calculated on, and the variable C1. TCR1 is then added to totalTable.

The schema contains conditions for making sure a previously started calculation for a date is completed before adding a new date to calculate on.

TotalCalc

ΔCostCalculator

physTableCounter ≤ #physTable ∧

(totalTable = ⟨⟩ ∨

(physTable(physTableCounter).Date ≠ totalTable(totalTableCounter).Date ∧

hedgeTable(hedgeTableCounter).Date ≠ totalTable(totalTableCounter).Date))

∃C1 : TOTCOST • C1 = 0 ∧

∃TCR1 : TOTALCOSTROW • TCR1 = n(physTable(physTableCounter).IndId,

physTable(physTableCounter).Date, C1) ∧

totalTable' = totalTable ~ (TCR1)

totalTableCounter' = totalTableCounter + 1
AddPhysCost is then used for adding every previously calculated PhysCost for the date currently being processed to the total cost for that date. It takes the value PhysCost in a row in physTable with the correct date and adds it to the value TotCost in the row for that date in totalTable. AddPhysCost is repeated for every row in physTable with the correct date.

\[
\text{AddPhysCost} \\
\Delta \text{CostCalculator}
\]

\[
\text{AddPhysCost} \\
\Delta \text{CostCalculator} \\
\text{physTableCounter} \leq \#\text{physTable} \land \text{totalTableCounter} > 0 \\
\land \text{physTable}(\text{physTableCounter}).\text{Date} = \text{totalTable}(\text{totalTableCounter}).\text{Date} \\
\text{totalTable}(\text{totalTableCounter}).\text{TotCost}' = \text{totalTable}(\text{totalTableCounter}).\text{TotCost} \\
\quad + \text{physTable}(\text{physTableCounter}).\text{PhysCost} \\
\text{physTableCounter}' = \text{physTableCounter} + 1
\]

The operator AddHSettlement is used for adding every previously calculated hedge settlements for a certain date to the total cost of that date. It does the same thing as AddPhysCost, but with data from hedgeTable instead. This means that it takes the value of Hsttlmnt in a row with the correct date from hedgeTable and adds it to the value of TotCost in the corresponding row in totalTable. AddHSettlement is repeated for every row in hedgeTable with the correct date.

\[
\text{AddHSettlement} \\
\Delta \text{CostCalculator}
\]

\[
\text{AddHSettlement} \\
\Delta \text{CostCalculator} \\
\text{hedgeTableCounter} \leq \#\text{hedgeTable} \land \text{totalTableCounter} > 0 \\
\land \text{hedgeTable}(\text{hedgeTableCounter}).\text{Date} = \text{totalTable}(\text{totalTableCounter}).\text{Date} \\
\text{totalTable}(\text{totalTableCounter}).\text{TotCost}' = \text{totalTable}(\text{totalTableCounter}).\text{TotCost} \\
\quad + \text{hedgeTable}(\text{hedgeTableCounter}).\text{Hsttlmnt} \\
\text{hedgeTableCounter}' = \text{hedgeTableCounter} + 1
\]

The following three schemas PrintPhysTable, PrintHedgeTable, and PrintTotalTable are used for showing the values of the three corresponding tables to the user. They can be repeated for every row in each table.

\[
\text{PrintPhysTable} \\
\Xi \text{CostCalculator}
\]

\[
\Xi \text{CostCalculator} \\
\Xi \text{CostCalculator} \\
\text{physPrintCounter} \leq \#\text{physTable} \\
\text{physResult}' = \text{physTable}(\text{physPrintCounter}) \\
\text{physPrintCounter}' = \text{physPrintCounter} + 1
\]
PrintHedgeTable

\[\exists\text{CostCalculator} \quad \text{hedgeResult}!: \text{HEDGESETTLROW}\]

\[
\text{hedgePrintCounter} \leq \#\text{hedgeTable} \\
\text{hedgeResult}! = \text{hedgeTable}(\text{hedgePrintCounter}) \\
\text{hedgePrintCounter}' = \text{hedgePrintCounter} + 1
\]

PrintTotalTable

\[\exists\text{CostCalculator} \quad \text{totalResult}!: \text{TOTALCOSTROW}\]

\[
\text{totalPrintCounter} \leq \#\text{totalTable} \\
\text{totalResult}! = \text{totalTable}(\text{totalPrintCounter}) \\
\text{totalPrintCounter}' = \text{totalPrintCounter} + 1
\]

Full specification is achieved by the formula:

\[\text{FullCostCalculator} \equiv \text{CostCalculator} \land \text{CostCalcInit} \land \text{UserInput} \land \text{PhysCalcInput} \land \text{HedgeCalcInput} \land (\text{TotalCalc} \lor \text{AddPhysCost} \lor \text{AddHSettlement}) \land \text{PrintPhysTable} \land \text{PrintHedgeTable} \land \text{PrintTotalTable}\]
5 Conclusions
This thesis has examined formal specifications and their possible usefulness regarding a price calculating program for electricity contracts. In this chapter, a conclusion of the findings will be presented.

5.1 Formal Specifications Useful for a Price Calculation System?
Since the main idea of a formal specification is to reduce the possible problems and faults in a developed program, and it is actually currently mostly being used for, among other things, systems with a risk of high costs should they fail, the conclusion here is that a formal specification can in fact be useful for a smaller price calculation system such as the one described in this thesis. The emphasis lies, though, in the word smaller, since one of the limitations of formal specifications is that they can grow disproportionally for larger systems.

5.2 Choice of Formal Specification Type?
Out of the different specification techniques examined in the thesis, the most suited technique for a price calculation system such as the one described in Chapter 4, would seem to be a model-based specification, particularly written in the language Z. This choice is mainly due to the fact that most of the other techniques, such as transition-based specifications and algebraic specifications, are better suited for a different kind of program and they are not as expressive as the model-based one, and the criteria for the system made it important to be able to express the different calculation formulas as simple and easy to read as possible.

5.3 Future Work
If one were to continue the research work started in this paper, the next logical step would be to perform a proof check on the developed specification and possibly make changes accordingly if it would be discovered that the specification is not fully complete.

Furthermore, if the program would be implemented and tested, it could give a better insight in the possible benefits of having written a formal specification first.
6 Discussion
This chapter will present a brief discussion on some of the advantages and disadvantages of formal specifications.

6.1 Advantages of Using a Formal Specification
One of the advantages of writing a formal specification is that it can help find errors and possible issues in a system before it is fully implemented and possibly harder, or more expensive, to correct. Even slightly incorrect formal specifications can help reduce the possible errors that can occur in a system, and companies that have invested in the use of these specifications have reported that, as a result, they had less faults in their products without noticing an increase in the development cost.

Another advantage is that it can help the development, and design, process by raising questions not answered by the informal specification. This can help make a clearer description of what the software should do, and thus help building “the right” product from the beginning. This advantage was also noticed during the development of the price calculation program specification.

Furthermore, a formal specification can be used to formally prove that a system works correctly, and it can give the testers of an implemented system something concrete to test against when verifying the correctness of the program.

6.2 Disadvantages of Using a Formal Specification
One drawback of using a formal specification is that it can take a longer time to produce a system than it otherwise would. Today, many customers want their products at a faster rate.

Another disadvantage is that a formal specification can be impractical for a large system since both the time and effort needed to write it, and the ultimate size of the specification, grows disproportionally with the size of the system. A hint of this can also be seen in the developed specification, where a seemingly small calculation program still generated a six pages long specification.

Furthermore, using a formal specification does not necessarily work well with agile development, which many software companies use. There is, however, research that aims to suggest how the two can be combined.

Lastly, the specifications can be hard to perform correctly without expertise in formal systems and mathematical logic, although, as mentioned above, slightly incorrect specifications can still be of use. This drawback also shows itself when it comes to understanding an already written specification. Not everyone that has an interest in a project, such as customers, architects and developers, might understand it due to their different knowledge domains.
7 References


