Choice quantification in process algebra
Luttik, S.P.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: http://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
We shall conduct a systematic investigation of choice quantification in the context of process algebra. Choice quantification is used to describe the act of selecting an instantiation of a process with an arbitrary element from a data domain. This first chapter is meant to introduce the context in which the above mentioned subjects play a rôle. We first explain the basics of formal process specification, and why it is sometimes convenient to give a separate specification of some relevant data. Then, we shall describe a general method for assigning a mathematical meaning to formal specifications of processes, and we shall discuss the consequences for this method if some of the data is to be specified separately. Finally, we shall briefly mention the results about choice quantification that will be obtained in the remainder of this thesis.

1.1 Process specification

To start with, here is an informal description of a very simple process. At almost every street corner in downtown Amsterdam there is a car park ticket dispenser. This is a quite simple device that translates coins into parking time. When this thesis was written (on the eve of the introduction of the 'euro'), the machine accepted the following Dutch coins: 'kwartjes' (Dfl. 0.25), 'guldens' (Dfl. 1.00), 'rijksdaalders' (Dfl. 2.50) and 'vijfjes' (Dfl. 5.00).

A car owner wishing to avoid a wheel clamp will look for the nearest ticket dispenser —advertised by the capital P— immediately after parking his car. It presents him with the following options:

1. He can insert a coin.

2. He can press a green button to instruct the machine to produce a ticket.
   The dispenser requires the insertion of at least Dfl. 0.50 to print a ticket; otherwise, pressing the green button has no effect.

3. He can turn a red knob causing the machine to return all the coins that were inserted since the last time that either the button was pushed or the knob was turned.

In the city centre, the parking fee is approximately Dfl. 5.00 an hour. (To be entirely honest, the fee is Dfl. 5.75 per hour between 9am and 7pm, and Dfl. 3.25 between 7pm and 11pm, and outside these hours parking is for free. Further,
there is obviously a (physical) limit on the amount of money that the car owner can deposit in the dispenser. We ignore such details so as to guarantee that our example retains its promised simplicity. Also, we are not sure about the internal precision of our dispenser when it associates minutes with coins; somewhere in the process it presumably rounds off to the nearest minute. It is more convenient to work with a fee of Dfl. 5.00 per hour, so that the number of minutes associated with each coin is an integer; e.g., a 'kwartje' buys 3 minutes of parking time.)

1.1.1 Implicit data

Above, we have informally described the events that may occur in the process of obtaining a ticket from the ticket dispenser. We assign to each of these events a formal symbol from the following list

\[ \text{ink, ing, inr, inv, button, print, knob, return.} \]  \hspace{1cm} \text{(1.1)}

The symbols "ink", "ing", "inr" and "inv" respectively refer to the events of the car owner inserting a 'kwartje', a 'gulden', a 'rijksdaalder' or a 'vijfje' into the ticket dispenser. The symbol "button" refers to the event of him pressing the green button, and the symbol "knob" refers to the event of him turning the red knob. The symbol "print" refers to the event of the dispenser printing a ticket, and the symbol "return" refers to the event of it returning all the recently inserted coins.

Furthermore, let us attach the number of minutes for which the car owner has paid as a subscript to the name of the ticket dispenser. This gives us another list of formal symbols:

\[ P_0, P_3, \ldots, P_{3n}, \ldots \hspace{1cm} (n \geq 0). \]  \hspace{1cm} \text{(1.2)}

The number \( 3n \) \((n \geq 0)\) may be thought of as the state the ticket dispenser got into when the car owner inserted coins to the equivalent of \( 3n \) minutes. The symbol \( P_{3n} \) refers to the behaviour of the ticket dispenser when it is in state \( 3n \).

Inserting a coin, pressing the button and turning the knob will generally have the effect of changing the state of the ticket dispenser; e.g., inserting a coin will increase the subscript by the number of minutes associated with that coin, and turning the knob while the dispenser is in some state \( 3n \) \((n \geq 1)\) will make the dispenser return all the inserted coins and go back to state 0. We introduce the symbol "\( \cdot \)" to express that things happen consecutively. For instance, we write "\( \text{ink} \cdot P_3 \)" if we want to say that the insertion of a 'kwartje' makes the dispenser go into state 3, and we write "\( \text{knob} \cdot \text{return} \cdot P_0 \)" if we want to say that after turning the knob the dispenser returns the inserted coins and goes into state 0.

According to our informal descriptions, the car owner may activate a number of alternative events. To specify this, we introduce the symbol "\( + \)"; e.g., to express that the car owner may choose to insert a 'kwartje' to make the dispenser go into state 3, or to insert a 'gulden' to make the dispenser go into state 12, we write "\( \text{ink} \cdot P_3 + \text{ing} \cdot P_{12} \)". Incidentally, note that the car owner may choose to insert a coin, press the button or turn the knob irrespective of the actual state of the ticket dispenser.
1.1 Process specification

We can now define the behaviour of our ticket dispenser by simultaneously specifying the behaviours that it may exhibit in each of its states:

\[ P_0 = \text{in}_k \cdot P_3 + \text{in}_g \cdot P_{12} + \text{in}_r \cdot P_{30} + \text{in}_v \cdot P_{60} + \text{button} \cdot P_0 + \text{knob} \cdot P_0; \]
\[ P_3 = \text{in}_k \cdot P_6 + \text{in}_g \cdot P_{15} + \text{in}_r \cdot P_{33} + \text{in}_v \cdot P_{63} \]
\[ + \text{button} \cdot P_3 + \text{knob} \cdot \text{return} \cdot P_0; \]

and for all \( n \geq 2 \):

\[ P_{3n} = \text{in}_k \cdot P_{3n+3} + \text{in}_g \cdot P_{3n+12} + \text{in}_r \cdot P_{3n+30} + \text{in}_v \cdot P_{3n+60} \]
\[ + \text{button} \cdot \text{print} \cdot P_0 + \text{knob} \cdot \text{return} \cdot P_0. \]

The equations above may serve as a formal specification of the behaviour of any car park ticket dispenser in downtown Amsterdam. Now, suppose that our car owner did not find a place to park his car in the city centre, and that he was forced to put it somewhere just outside the city centre. He is still in a part of Amsterdam where he has to pay, but parking time is twice as cheap: the fee is Dfl. 2.50 an hour. The car park ticket dispensers in this part of Amsterdam look very similar to those in the city centre; they carry the same initial (P) and appear to behave in the same fashion too. The difference only becomes apparent when one compares the amount of money inserted and the number of minutes allotted in the two regions.

What should we do to adapt the specification given above in such a way that it describes the behaviour of a ticket dispenser in this part of Amsterdam? We should double every number that appears as a subscript, thus obtaining a specification that assigns a behaviour to the symbols

\[ P_0, P_6, \ldots, P_{6n}, \ldots \quad (n \geq 0). \]

The new specification accurately describes the behaviour of a ticket dispenser just outside the city centre. It is somewhat unfortunate, however, that the intuitively clear relationship between the new specification and the previous one is obscured by a computation. If we were to order ticket dispensers for all of Amsterdam, it would be more convenient if we could give the manufacturer just one specification of their behaviour, plus the going rates for the different regions.

1.1.2 Explicit data

We started out to say that a car park ticket dispenser is a machine that translates coins into minutes; coins and minutes are the types of data on which our ticket dispenser operates. That our specification involves data at all has thus far been implicit in our suggestive nomenclature for states and events. Let us now proceed and give explicit definitions of some of the data in our specifications. The accepted coins are the elements of a set \( \mathcal{C} \); in our example

\[ \mathcal{C} = \{k, g, r, v\}. \]
When a car owner inserts a coin $c \in C$ into the slot of a ticket dispenser, this has the effect of increasing the parking time bought by the number of minutes $T(c)$ associated with $c$; e.g., in the case of a ticket dispenser in the city centre

$$T(k) = 3, \quad T(g) = 12, \quad T(r) = 30, \quad \text{and} \quad T(v) = 60;$$

and in the case of a ticket dispenser just outside the city centre

$$T(k) = 6, \quad T(g) = 24, \quad T(r) = 60, \quad \text{and} \quad T(v) = 120.$$

We are going to consider the set of coins $C$ and the coins-to-minutes translation $T$ as parameters of a general specification of the behaviour of car park ticket dispensers.

To emphasize that the data have now come to the fore, we stop pushing them away in subscripts: we write $P(n)$ to refer to the ticket dispenser after the insertion of coins to the equivalent of $n$ minutes; and we write $\text{in}(k)$, $\text{in}(g)$, $\text{in}(r)$ and $\text{in}(v)$ to refer to the events of inserting the respective coins. The car owner inserting a coin, with the dispenser in state $n$, could then be denoted by

$$\text{in}(k) \cdot P(n + T(k)) + \text{in}(g) \cdot P(n + T(g))$$

$$+ \text{in}(r) \cdot P(n + T(r)) + \text{in}(v) \cdot P(n + T(v)).$$

(Caution: the symbol “+” occurs in two different capacities: referring to a choice between events, and referring to addition of natural numbers.)

This notation abstracts from the particular association between coins and minutes, and hence is suitable for the specification of ticket dispensers inside and outside the city center. But it is quite long, and it contains some redundant information. That is, which coins the ticket dispensers accept is already clear upon presenting the parameter $C$; we can abstract from this information in the specification of their behaviours. We want to say that the car owner may insert any member $c$ of the set of coins $C$, upon which the dispenser updates its state with the appropriate number of minutes $T(c)$. We give the symbol “$c$” the status of a variable that ranges over $C$, introduce a new formal symbol “$\sum_c$”, and specify the event by

$$\sum_c \text{in}(c) \cdot P(n + T(c)).$$

In contrast to the previous notation, the new notation has the additional advantage of being ‘euro proof’: to make the transition from the present Dutch coins to European currency, the only thing that has to be done is to adapt the parameter $C$, replacing the Dutch coins by euros, and to adapt the mapping $T$ accordingly. Incidentally, the new notation also reflects in a more natural way the physical appearance of ticket dispensers in Amsterdam: they have only one slot, which takes all types of coins.

The effect of pressing the green button depends on the state $n$ of the dispenser: if the car owner has paid for at least 2 times the amount of minutes associated with a ‘kwartje’, i.e., $n \geq 2 \times T(k)$, then the dispenser produces a ticket; otherwise nothing
1.1 Process specification

happens. To specify this in a concise way, we use the notation “< n ≥ 2 × T(k) ▷” (in general, read ‘x < b ▷ y’ as ‘then x if b else y’); e.g., we write

button · print · P(0) < n ≥ 2 × T(k) ▷ button · P(n)

to abbreviate ‘if n ≥ 2 × T(k), then button · print · P(0) happens, and otherwise button · P(n) happens’.

We can now specify the behaviour of ticket dispensers in Amsterdam concisely by means of the following equation:

\[
P(3n) = \sum_c \text{in}(c) \cdot P(3n + T(c)) \\
+ \text{button} \cdot \text{print} \cdot P(0) < 3n ≥ 2 \times T(k) \triangleright \text{button} \cdot P(3n) \\
+ \text{knob} \cdot \text{return} \cdot P(0) < 3n ≥ 3 \triangleright \text{knob} \cdot P(0).
\]  

(1.3)

Note that, to make this behaviour specification completely euro proof, we should eliminate the explicit mention of k in 2 × T(k), e.g., by introducing a constant that represents the minimum number of minutes that can be bought.

Most of the symbols used in (1.3) explicitly refer to specific aspects of ticket dispensers, denoting specific events associated with such machines or naming specific behaviour that they may exhibit in a certain state. In contrast, the symbols “+”, “,”, “∑” and “< ▷” refer to mechanisms that are not specific to ticket dispensers. They have the kind of generality that one expects of the primitives of a general purpose specification formalism.

1.1.3 The process specification language μCRL

The previous two subsections serve to illustrate that there is at least a conceptual advantage in defining some relevant data separately when specifying a process. Many process specification languages, i.e., specification languages whose principal purpose is to specify the behaviour of systems, nowadays are accommodated with facilities to define data separately and with mechanisms to incorporate these in the actual behaviour specification. Examples of such process specification languages are, e.g., LOTOS (Bolognesi and Brinksma, 1987), PSF (Mauw and Veltink, 1990) and μCRL (Groote and Ponse, 1995). In this thesis, we shall elaborate on the theoretical foundations of the process specification language μCRL (micro Common Representation Language; for a survey, see Groote and Reniers (2001)).

In a μCRL specification, abstract data types are defined by means of a many-sorted algebraic specification (see, e.g., Bergstra et al., 1989; Loeckx et al., 1996). There is a facility to declare basic events that may take the specified data as parameters, and to aid the description of processes, μCRL includes the mechanisms symbolised by “+”, “,”, “∑” and “< ▷”. To give some idea of what a μCRL specification looks like, we present in Table 1.1 a complete formal specification of a ticket dispenser in the centre of Amsterdam, in μCRL syntax. At this point, two further remarks about μCRL are in order.

Firstly, one should take our ‘μCRL syntax’ with a pinch of salt. The official syntax of μCRL was designed to be read by computers; to enhance readability for humans we deviate slightly from it. We use mathematical symbols (e.g., ≤, +)
sort B
func T, ⊥ :→ B
   ≥: M × M → B
var x, y : M
rew (x ≥ 0) = T
       (0 ≥ 3 + x) = ⊥
       (3 + x ≥ 3 + y) = (x ≥ y)

sort M
func 0.3 :→ M
   + : M × M → M
var x, y, z : M
rew (x + y) + z = x + (y + z)
       x + y = y + x
       x + 0 = x

sort C
func k, g, r, v :→ C
   T : C → M
rew T(k) = 3
       T(g) = 4 × 3
       T(r) = 10 × 3
       T(v) = 20 × 3

act button, print, knob, return
   in : C
   in (c)
   P(0)
   P(m)
   P(m + T(c))
   button · print · P(0) < m ≥ 2 × T(k) >
   button · P(m)
   knob · return · P(0) < m ≥ 3 >
   knob · P(0)

proc P(m : M) = \sum_{c : C} \text{in}(c) \cdot P(m + T(c))
   + button · print · P(0) < m ≥ 2 × T(k) > button · P(m)
   + knob · return · P(0) < m ≥ 3 > knob · P(0)

Table 1.1: A μCRL specification of a car park ticket dispenser. We use an abbreviation that ought to be spelled out: if t is a term of sort M and n is a natural number, then we write $n \times t$ to denote the term $(\cdots (t + t) + \cdots + t) + t$ with $n$ occurrences of $t$.

as names of functions declared in func sections and write them infix, whereas the official μCRL syntax only allows strings of letters from the Latin alphabet as names for such functions and prescribes that they be written prefix. Also, we write $\sum_{c : C} -$ instead of $\text{sum}(c : C, -)$ and $- < \ldots > -$ instead of $< \ldots >$.

Our specification in Table 1.1 is, of course, a rather simplistic example, and that it describes what we intended to describe is a fact that hardly needs additional justification. But for larger and more complex specifications, it is vital to have computer support, e.g., to simulate specifications and to verify that they have certain properties. For μCRL, such computer support is available (see Blom et al. (2001), or consult http://www.cwi.nl/~mcrl).

Secondly, μCRL has additional mechanisms to facilitate the specification of processes; e.g., it includes mechanisms to specify that a process consists of several components running in parallel, to specify that certain parallel components must
synchronise, and to specify that certain events should be considered unobservable. In this thesis these mechanisms will not play a rôle. Interestingly, one of the computerised tools for $\mu$CRL, the so-called lineariser (see Groote et al., 2001), is able to translate many $\mu$CRL specifications to $\mu$CRL specifications without these additional mechanisms. The other tools operate on the output of this lineariser.

1.2 Process theory

We have introduced a collection of formal symbols to write down $\mu$CRL specifications. Our explanations of the meanings of these symbols are still informal, saying something to the effect that "+" indicates a choice between alternatives, that "·" indicates that events occur consecutively, that "<·>" indicates a choice that depends on a condition, and that "$\Sigma$·" indicates a choice that depends on input. So far, we got away with such informalities, because we have been specifying a ticket dispenser and most people already have a pretty good idea of how such slot machines tend to behave. But it is, of course, an undesirable situation that the behaviour of the ticket dispenser explains the meaning of its $\mu$CRL specification. It should be the other way around: our $\mu$CRL specifications should explain the behaviours of the systems they are meant to describe.

In other words, we want to give $\mu$CRL specifications a meaning that is independent of the systems that they intend to describe, preferably as a mathematical abstraction of the concept of a process. By interpreting $\mu$CRL specifications as mathematical objects, $\mu$CRL becomes a mathematical language. A distinct advantage of this is that we can then prove by mathematical means that a system behaves (or does not behave) the way it should. Before explaining the approach taken in this thesis to turn $\mu$CRL into a mathematical language, we first discuss one of our methodological considerations in a more general context.

1.2.1 Process calculi

A mathematical theory about objects that are thought of as mathematical abstractions of processes, is frequently called a model of concurrency, since, intuitively, a process consists of a number of activities running in parallel. Such a model of concurrency together with a formal language to reason about its elements, is what we call a process calculus. The pioneers of the design of process calculi are Hoare (1985) and Milner (1980).

Hoare introduced CSP (Communicating Sequential Processes) to reason about a mathematical model in which a process is viewed as a set of failures. A failure consists of a sequence of events in which the process may engage, together with a set of events that it subsequently refuses to engage in. Typically, the formal symbols of CSP are interpreted as operations on the failures model. These operations are shown to satisfy a set of basic mathematical laws in the form of equations, which support the mathematical reasoning about them (see Brookes et al. (1984)).

Milner introduced CCS (Calculus of Communicating Systems; see also Milner (1989, 1999)) to reason about a mathematical model in which a process is viewed as a labeled transition system modulo observation equivalence. A labeled transition
system consists of *states*, and *transitions* between states labeled with names of events; a transition marks the occurrence of the associated event. With each such labeled transition system one can, intuitively, associate a notion of observable behaviour. To consider a labeled transition system modulo observation equivalence means to consider a set consisting of all labeled transition systems that represent the same observable behaviour.

Again, the formal symbols of CCS are interpreted as operations on the mathematical model for which CCS was introduced, i.e., on sets of labeled transition systems modulo observation equivalence. And again, to support the mathematical reasoning, these operations are shown to satisfy a set of basic mathematical laws in the form of equations. We quote Milner (1983):

> "These four operators [of CCS] obey (as we show) several algebraic identities. It is not too much to hope that a class of these identities may be isolated as axioms of an algebraic "concurrency" theory, analogous (say) to rings or vector spaces. For the present, however, we concentrate on an interpretation of the calculus derived from an operational or dynamic understanding of each operator, whereupon the algebraic identities arise as theorems."

By using the terms 'rings' and 'vector spaces', Milner makes a connection with an established area of mathematics, that of abstract algebra (see, e.g., Hungerford, 1974). It comprehends the study of algebras, structures that consist of a set (universe) with a sequence of operations defined on it. The desideratum is to abstract from the nature of the elements of the universe, and to study the fundamental properties of the operations, conventionally expressed in the form of equations. Typically, one studies all algebras that satisfy a particular collection of equational axioms.

### 1.2.2 Process algebra

In the literature, there is not (yet) an established consensus about what is the appropriate mathematical abstraction of the notion of process, judging by the many different models of concurrency that are currently in use. However, the languages associated with these models (if any) often include mechanisms to express

1. that a process consists of a choice between a number of alternative behaviours (*alternative composition*);

2. that a process consists of a number of behaviours that are performed consecutively (*sequential composition*); and

3. that a process consists of a number of behaviours that are executed in parallel (*parallel composition*).

Bergstra and Klop (1984) propose to study these and other process theoretic mechanisms through the axiomatic method, instead of via a presupposed model of concurrency. They coined the term *process algebra* for their approach.
Wholly in the style of the contemporary textbooks on abstract algebra, Bergstra and Klop present their algebraic theory of processes in a modularised fashion. They begin with formulating the algebraic theory BPA (Basic Process Algebra) of alternative and sequential composition, both represented as binary operations. Then, they consider the algebras that satisfy the axioms of BPA and in which there is a neutral element for alternative composition that acts as a left zero for sequential composition. That element stands for deadlock, the process without any behaviour. The algebraic theory of alternative composition, sequential composition and deadlock is called \( \text{BPA}_\delta \) (Basic Process Algebra with deadlock). Subsequently, they discuss extensions of the theories BPA and \( \text{BPA}_\delta \) with a binary operation for parallel composition. First, they consider parallel composition without communication, obtaining the theories PA (Process Algebra) and \( \text{PA}_\delta \) (Process Algebra with deadlock). Thereafter, they also consider a form of parallel composition in combination with mechanisms to express and to require synchronisation between parallel components; the resulting theory is called ACP (Algebra of Communicating Processes).

What makes the process theories of Bergstra and Klop truly algebraic is that they are axiomatic, and, moreover, not prescriptive with respect to the objects that are taken to represent processes (in the same way as group theory does not prescribe what the elements of a group should be). This has a didactical advantage; for instance, to understand what alternative composition is, one does not need to first digest, e.g., the mathematically quite involved definition of labeled transition system modulo observation equivalence. Furthermore, placing process theoretic mechanisms in a general algebraic context has the methodological advantage that they easily make contact with mechanisms studied elsewhere. For instance, it is at once clear that an algebra satisfying the axioms of BPA is a semilattice with respect to alternative composition, and that it therefore has a natural partial order associated with it, defined in terms of alternative composition. This partial order turns out to be a convenient tool in process algebra.

The algebraic theory BPA of alternative and sequential composition may be used to give a precise mathematical interpretation of our first formal specification of the ticket dispenser (the one with implicit data). The most important stipulation is that the symbols "+" and "·" denote the binary operations of alternative and sequential composition from the theory BPA. To assign a mathematical object to our specification, we need to select

1. a particular algebra, say \( P \), that satisfies the axioms of BPA, and
2. interpretations of the symbols listed in (1.1) on p. 2 as elements of \( P \).

Henceforth we shall refer to the combination of 1 and 2 as a model of BPA with actions. Then, a solution of our specification in \( P \) is an assignment of elements of \( P \) to the symbols listed in (1.2) such that all the defining equations are true in \( P \).

To proceed a little more generally we define grammatical categories of

process expressions: (i) each of the symbols in the lists (1.1) and (1.2) are process expressions; (ii) if \( p \) and \( q \) are both process expressions, then so are \( p + q \...
and \( p \cdot q \); and (iii) every process expression can be obtained by finitely many applications of (i) and (ii):

**process equations:** if \( P \) is a symbol from the list (1.2), and \( p \) is a process expression, then \( P = p \) is a process equation that defines \( P \); and

**process specifications:** a set of process equations such that each symbol from the list (1.2) is defined exactly once.

We see that the symbols in the list (1.1) and those in the list (1.2) have different grammatical functions in a process specification. To distinguish them, we agree to call the symbols in the first list *action symbols*, and those in the second list *process variables*.

We can now speak of the solution of an arbitrary process specification in a model \( \mathbf{P} \) of BPA with actions. The point of the algebraic approach, however, is that we do not need to commit ourselves to a particular model of BPA with actions, before we can start doing calculations. For instance, we can already prove the equivalence of process specifications by applying the axioms of BPA (the first five axioms in Table 2.1 on p. 17 below) to the right-hand sides of their process equations, or by applying other rules that preserve the solutions of recursive specifications in *any* model of BPA with actions (see, e.g., Ponsé and Usenko, 2001).

Now, let us consider our \( \mu \text{CRL} \) specification of the ticket dispenser to see whether it can also be given a precise mathematical interpretation by means of the theory BPA. At first sight, it does not fit our definition of a process specification, because of the occurrences of the symbols "\( \Sigma \)" and "\( \langle \_ \rangle \)". However, recall that our initial motivation for introducing these extra symbols was to be able say things more succinctly, and not to be able to say new things. Our \( \mu \text{CRL} \) specification of the ticket dispenser was intended to specify in a better way what had already been specified by our first specification. The latter is, according to our definition above, a genuine process specification. We could perhaps give a mathematical interpretation to \( \mu \text{CRL} \) specifications by first translating them to process specifications.

In our example, the translation may be carried out in three straightforward steps:

1. Replace the expression \( \sum_c \text{in}(c) \cdot P(n + \text{T}(c)) \) by the sum of all the instances of the expression \( \text{in}(c) \cdot P(n + \text{T}(c)) \) with a coin for the variable \( c \).

2. Collect in a set all the instantiations of the equation defining \( P(m : M) \) with a natural number of the form \( 3n \) for the variable \( m \).

3. Eliminate all occurrences of "\( \langle \_ \rangle \)" from the equations in the set obtained in the second step. This can be done by replacing the occurrences of expressions "\( \text{button-print} \cdot P(0) \langle 3n \geq 2 \times \text{T}(k) \rangle \text{button} \cdot P(3n) \)" by "\( \text{button-print} \cdot P(0) \)" if \( 3n \geq 2 \times \text{T}(k) \) evaluates to true and by "\( \text{button} \cdot P(3n) \)" if it evaluates to false, and by treating the occurrences of "\( \text{knob-return} \cdot P(0) \langle 3n \geq 0 \rangle \text{knob} \cdot P(0) \)" in a similar fashion.

Note that the first two steps of the translation eliminate all occurrences of variables, and that this guarantees that the evaluation of the middle component of an
occurrence of "< _ >" in the third step can always be done. Further note that the set of equations generated in the second step is infinite; but this is not a problem since process specifications consisting of an infinite set of process equations are allowed according to our definition.

Nevertheless, the recipe does not work in general, and the culprit is in the first step. The variable associated with an occurrence of the symbol "\( \sum \)" may range over any of the specified data sorts, and consequently it may range over an infinite set (e.g., it could range over the sort \( M \) in our example specification). So the syntactic sum that should be associated with a \( \mu \text{CRL} \) expression starting with an occurrence of the symbol "\( \sum \)" may, in general, consist of infinitely many components. However, such an infinite sum is not a process expression according to our inductive definition. And with reason: the intended infinite alternative composition may fail to exist for some models of BPA with actions. That is, whether a model of BPA with actions is suitable for the interpretation of a certain \( \mu \text{CRL} \) specification, depends on whether it has the right sums.

### 1.2.3 Infinite sums

Let \( P \) be an algebra that satisfies the axioms of \( \text{BPA}_\delta \) (in what follows it will be convenient to assume the presence of the neutral element \( \delta \) for alternative composition). We have already indicated that \( P \), being a semilattice with respect to alternative composition, has a natural partial order associated with it. Conversely, the alternative composition of two elements of \( P \) may be defined as their least upper bound with respect to that partial order. Generalising this definition, the alternative composition of the empty set is the minimal element \( \delta \), and the alternative composition of a nonempty finite subset \( P' \) of the universe of \( P \) is its least upper bound, say \( \sum P' \). If \( P' \) is an infinite set, then it may not have a least upper bound \( \sum P' \) in the universe of \( P \).

In Chapter 2 of this thesis we shall develop a general theory about infinite sums in algebras that satisfy the axioms of \( \text{BPA}_\delta \). If \( P \) is the universe of such an algebra, then we define on it an operation

\[
\sum : D \to P, \text{ with } D \subseteq \{P' \mid P' \subseteq P\}.
\]

In the terminology of Rasiowa and Sikorski (1963), the operation \( \sum \) is a "generalised operation". Assigning an element of \( P \) to some (but generally not to all) subsets of the universe \( P \), it may be thought of as a partial operation with variable (and possibly infinite) arity. The element \( \sum P' \) assigned to a subset \( P' \) of \( P \) must satisfy a few requirements. To tie in with process algebraic traditions, we shall define those requirements by means of axioms in the form of equations. The theory that is obtained by adding these axioms to the axioms of \( \text{BPA}_\delta \) we shall refer to as \( \text{GBPA}_\delta \) (Generalised Basic Process Algebra with deadlock); the mechanism embodied by \( \sum \) we call generalised summation.

To substantiate our definitions, we shall study certain natural extensions with infinite sums of the algebra of finitely branching transition trees of finite depth (for which the axioms of \( \text{BPA}_\delta \) are known to be sound and complete). In the case of transition trees, the sum \( \sum T \) of a set \( T \) of transition trees is the transition tree
that we get by identifying the roots of the trees in $T$. Clearly, $\sum T$ is a countably branching tree provided that the set $T$ is countable and its elements are countably branching trees. Therefore, on the algebra of countably branching transition trees the operation $\sum$ assigns a sum to every countable set of transition trees. In the same way we can define for every infinite cardinal $\kappa$ an algebra of transition trees with branching degree $\leq \kappa$ that is closed under sums of cardinality $< \kappa$. We shall prove that the axioms of $\text{GBP}_\delta$ are sound and complete for each of these algebras (see Theorem 2.11).

### 1.3 Choice quantification

The symbol "$\sum_x$" of $\mu\text{CRL}$ refers to a rather special kind of generalised summation. Let $x$ be a variable that ranges over an arbitrary set $D$ of data values (e.g., associated with a sort in the data part of a $\mu\text{CRL}$ specification). If $p$ is an expression that denotes a process after instantiating its data variables, then

$$\sum_x p = \sum\{p[x := d] \mid d \in D\}.$$  

In words, $\sum_x p$ is the generalised sum of all the instantiations of $p$ with an element of $D$ for the variable $x$. Thus, $\sum_x$ refers to taking the generalised sum of a set that is obtained by quantification over $D$; henceforth, we speak of choice quantification, and call $\sum_x$ a choice quantifier.

From Chapter 3 onwards, we shall study choice quantification in the context of $\text{pCRL}$ (pico Common Representation Language), which is $\mu\text{CRL}$ without parallelism. With respect to its common definition, we shall make two further simplifications:

1. We restrict our attention to processes that are specified from an alphabet of actions parametrised with data by means of alternative and sequential composition, conditional composition and choice quantification; in particular, we do not consider recursion. Thus, we only consider processes whose behaviour is finite with respect to the number of actions that they can perform consecutively. This is certainly not a minor restriction. Clearly, $\mu\text{CRL}$ without recursion would not have many applications as a process specification language. For all that, we do believe that it is sensible to try and understand the case without recursion first, since it usually greatly helps the understanding of the case with recursion as well.

2. We shall only consider choice quantification over a single data domain, which may be fitted with functions and relations. This is for the sake of clarity of presentation only; all of our results generalise in the obvious way to choice quantification over multiple domains and thus fit in with $\mu\text{CRL}$.

In Chapter 3 we give a semantics to the language $\text{pCRL}$ by explaining how its expressions denote elements of suitable models of $\text{GBP}_\delta$, where suitability depends on the availability of generalised sums. For every particular choice of a data domain $D$ and an alphabet of parametrised actions $A$ we shall define a suitable algebra of transition trees that is initial in the class of all suitable models.
of GBPA (Theorem 3.15). If the cardinality of D is infinite, then there exist pCRL expressions whose interpretation in this initial algebra is an infinitely branching transition tree.

For use in later chapters, we shall establish in Chapter 3 two results regarding particular syntactic forms of pCRL expressions. The first result states that every pCRL expression is semantically equivalent to a tree form, a pCRL expression whose syntactic structure, intuitively, reflects the structure of the transition tree that it denotes (Lemma 3.22). The second result makes use of a straightforward translation of the finite, sequential fragment of value-passing CCS (as discussed, e.g., by Hennessy and Lin (1996)) into pCRL expressions. Our interest is in the input prefix mechanism, the translation of which involves choice quantification: if the translation assigns the pCRL expression $p'$ to an expression $p$ of value-passing CCS (let us concisely denote this by $p \rightarrow p'$), then

$$c?x.p \rightarrow \sum_x c(x) \cdot p'.$$

Note that the variable $x$ of the choice quantifier $\sum_x$ occurs as a parameter of the action $c$ that immediately follows it. If all choice quantifiers in an expression have this property, then we shall say that the expression has explicit instantiation. It turns out that the tree forms associated with (translations of) expressions of value-passing CCS have explicit instantiation (Lemma 3.26).

### 1.3.1 Expressiveness

In Chapter 4 we shall investigate the expressiveness of the mechanisms of pCRL by considering the complexity of pCRL equations. Not surprisingly, the complexity of a pCRL equation depends on the data that occur in it. The results that we shall obtain are therefore relative to the complexity of the incorporated data. Precisely, we prove that any pCRL equation can be effectively transformed into an equivalent first-order assertion about the data (Theorem 4.10), and that, conversely, any first-order assertion about the data gives rise to an equivalent pCRL equation (Theorem 4.17). Hence, pCRL is as expressive as first-order logic, with respect to the incorporated data.

In particular, we shall see that pCRL owes to a large extent its expressiveness to choice quantification. It accounts for the simulation of the universal as well as the existential quantifiers of first-order logic. It turns out that an equation of pCRL expressions with explicit instantiation has the content of a universal first-order assertion about the data that occurs in it (Corollary 4.23). Hence, the finite, sequential fragment of value-passing CCS is as expressive as the universal fragment of first-order logic, with respect to the incorporated data.

### 1.3.2 Deductive system

In Chapter 5 we shall present a deductive system for pCRL equations, so that when doing calculations with pCRL expressions we may proceed entirely syntactically. The desiderata for the design of our deductive system are
1. to separate reasoning about the data inside \( \text{pCRL} \) expressions from reasoning about behavioural aspects; and

2. to fit in as much as possible with standard equational reasoning.

A natural question to ask about a deductive system is whether it is complete, i.e., whether it allows a deduction for every \( \text{pCRL} \) equation that holds in every suitable model of \( \text{GBPA}_\delta \). The expressiveness results of Chapter 4 indicate that such a completeness result cannot be obtained, unless drastic restrictions on the incorporated data are imposed. We prove that our deductive system is complete provided that it may ask an oracle to provide deductions of valid first-order assertions about the incorporated data (Theorem 5.20).

### 1.3.3 Algebraic semantics

The framework developed in Chapters 2, 3 and 5 has a syntactical side and a semantical side (see Figure 1.1). On the syntactical side we find the formal system \( \text{pCRL} \): it extends the formal system associated with \( \text{BPA}_\delta \) with choice quantifiers and conditionals. On the semantical side we find the algebraic theory \( \text{GBPA}_\delta \): it extends the algebraic theory \( \text{BPA}_\delta \) with generalised summation. The reason for extending \( \text{BPA}_\delta \) differently on both sides is as follows. On the one hand, generalised summation is an infinitary operation, i.e., it may take infinitely many arguments. Since is a desirable property of a formal system that its expressions are finite, an infinitary operation is not a convenient construction to have in such a system. On the other hand, the choice quantifiers of \( \text{pCRL} \) are binders, relying on the syntactic nature of their arguments, while the desideratum of an algebraic theory is to abstract from the nature of the objects under consideration.

A \( \text{pCRL} \) expression \( p \) together with a valuation that assigns data values to data variables describes a process, an element of a generalised basic process algebra with deadlock. The \( \text{pCRL} \) expression \( p \) itself may thus be thought of as the description of function from the set data values into a universe of processes, i.e., as the description of a parametrised process. In Chapter 6 we shall propose an algebraic theory of parametrised processes. It unites the syntactical and the semantical sides of \( \text{pCRL} \) in a single purely algebraic theory of basic process modules (BPM). It is obtained from \( \text{pCRL} \) by abstracting from the syntactic aspects of choice quantification, and it is obtained from \( \text{GBPA}_\delta \) by adding a notion of dimension. We shall prove that the (ground) equational theory of basic process modules is equivalent to that of \( \text{pCRL} \) (Theorem 6.37).
1.3 Choice quantification

Figure 1.1: The process theories in this thesis.