INTEGRATED INFORMATION SYSTEMS DESIGN

An Approach Based on Object-Oriented Concepts and Petri Nets

PROEFSCHRIFT

ter verkrijging van de graad van doctor
aan de Technische Universiteit Eindhoven,
op geslag van de Rector Magnificus,
prof.dr. J.H. van Lint, voor een commissie
aangewezen door het College van Dekanen
in het openbaar te verdedigen op
vrijdag 17 december 1993 te 16.00 uur

door

PETER ANTONIUS CHRISTIAAN VERKOULEN

geboren te Rosmalen
Dit proefschrift is goedgekeurd door
de promotoren

prof.dr. K.M. van Hee
en
prof.dr. J.J.E.L. Paredaens
Preface

The development of accurate Information Systems that satisfy present-day requirements, is a complex process with a lot of pitfalls. Many approaches have been proposed to solve the problems around IS analysis and design. So-called integrated approaches are good candidates for controlling the development process. The research that is described in this monograph is such an integrated approach. Many people have contributed to this work. I want to thank them all. Below, some of them are mentioned explicitly, but it is obvious that there are many more.

First, I thank Kees van Hee, for giving me the opportunity to conduct my research in his group. His enthusiastic encouragements helped me in difficult times. The pleasant way in which we cooperated has contributed significantly to the research I have done.

Similarly, I am grateful to Jan Paredaens. His knowledge of the field of object-oriented databases has been of much assistance. Also, the presentation of my research has gained a lot from his remarks.

Peter Apers was one of the referees of this monograph. He provided many suggestions for improvements, for which I am grateful. Moreover, I am looking forward to our future cooperation.

I am also obliged to Kurt Jensen, for his careful reading of drafts of this monograph. He made many valuable comments, some very detailed and others at a global level. Also, he gave me the possibility to visit Århus University twice. I have made a lot of friends there, some of who also commented on drafts of this monograph. In particular, I would like to thank Søren Christensen and Niels Damgaard Hansen.

All members of the Information Systems group of Eindhoven University of Technology are gratefully acknowledged for providing a stimulating research environment. In particular, I have had several discussions with Geert-Jan Houben, who also co-authored some of my papers.

My research has been conducted within the NFI project ISDF. I thank both
the Netherlands Organisation for Scientific Research (NWO) and Eindhoven University of Technology for giving the financial support for my work in Eindhoven. I am indebted to the members of the ISDF project for their critical remarks on my work. Guus Ramackers, with whom I wrote a paper that forms the basis for part of Chapter 6, should be mentioned explicitly.

I am also indebted to Charles Lakos from the University of Tasmania, who also read some drafts of this monograph. His remarks resulted in several improvements.

During my research, I have participated in the AXIS team, consisting of some 15 Dutch junior researchers in the field of Information Systems development. Their remarks on earlier presentations of my research have been very useful. I hope that we will continue our cooperation in the years to come.

The aforementioned people have all contributed, to some extent, directly to my research. However, three people have been even more important. My parents gave me the opportunity to go to university. Besides taking care that I did not lack anything, they also gave me their love and support. Thank you for everything!

Last but not least, I want to thank my wife Rosita. She has supported me in all the ways she could. Moreover, she made me realise that there are more important things in life than writing a dissertation. To her, I dedicate this monograph.
# Contents

1 Introduction  
1.1 Problem Statement .................................................. 13  
1.2 Outline of the SimCon Approach  
1.2.1 The SimCon Object Model (SCOM)  
1.2.2 The SimCon Algebra (SCA)  
1.2.3 The SimCon Net Model (SCNET)  
1.2.4 Automatic Verification  
1.2.5 Tool Support .................................................. 20  
1.3 A Small Example .................................................... 21  
1.4 Position of this Research ......................................... 23  
1.5 Outline of this Monograph ......................................... 27  

2 The SimCon Object Model  
2.1 Introduction ......................................................... 29  
2.2 Informal Description ............................................... 31  
2.3 Formal Definition .................................................. 38  
2.4 Tool Support ......................................................... 43  
2.5 Concluding Remarks ............................................... 45  

3 The SimCon Algebra  
3.1 Introduction .......................................................... 47  
3.2 Definition of the SimCon Algebra  
3.2.1 Selectors .......................................................... 50  
3.2.2 Definition of the SCA Kernel ................................... 58  
3.3 A Graphical Version of SCA ......................................... 68  
3.4 Advanced Operations ............................................... 71  
3.4.1 Non-recursive and Fix-Point Repetition ....................... 72  
3.4.2 Selective Expressions .......................................... 73  
3.4.3 Picking an Element from a Set  
3.4.4 Negation of Selectors .......................................... 75  
3.4.5 Ordering an Unordered Relation  
3.4.6 Transitive Closure of a Relation  
3.4.7 The Train-Bag-Letter Example  .................................. 77
Chapter 1

Introduction

In this chapter, we first state the problems that are treated in this monograph. Afterwards, we sketch our solution. This solution is illustrated by means of a small example. This example does not cover all aspects of our approach, but it does give the reader some intuition about our solution. After a comparison with other work in the field, we outline the contents of the remaining chapters.

1.1 Problem Statement

The research that is described in this monograph is concerned with engineering: the development of artificial constructs by human beings. We concentrate on Information Systems (IS). An IS is a system that records information about another system, often called the target system. An IS serves two purposes. First, it stores information that can be retrieved whenever necessary. Also, an IS may provide facilities for computing derived information, e.g. an average number or a diagnosis. The second main purpose of an IS may be to control the target system. Based upon the collected information (that may be gathered actively by the IS, e.g. by using sensors), the IS may send control signals (decisions) to the target system. E.g., an IS in a chemical plant may send control signals determining which valves must be opened and closed, based upon information the IS collects by using sensors.

In order to comprehend an IS completely, it is often also necessary to describe the target system and other environmental aspects of an IS. Therefore, this monograph deals with IS development in the broadest sense of the word. Many researchers have proposed various ways for dividing the development process. We distinguish three main stages:

Analysis
First, all existing systems are evaluated. We study their functionality and try to determine what the target system and the rest of the environment of the IS expect from the IS. This means that we also have to describe the behaviour of this environment to some extent.
All this results in requirements for the IS to be built. Therefore, some people call this the requirements analysis phase.

Design
In the second phase, a specification (also called design) of the system is developed. This specification determines all relevant structural and behavioural properties. Moreover, the specification is validated and verified. Validation means that the designer checks (to some extent) whether the specification satisfies the requirements from the analysis phase. As these requirements are hardly ever complete and unambiguous, formal and complete verification of the analysis requirements is generally not possible. In our approach, a specification can be used as a prototype of the system under development. This prototype can be used for validation purposes. Verification constitutes a formal way of checking the design. In most cases, this boils down to a check whether the design is internally consistent. E.g., when we state that a specific IS will never hold two different records representing the same person, the design may not contain any update action that violates this requirement. Also a liveness check is a kind of verification.

Realisation
The third phase of the development process is the realisation phase. In this phase, the system is implemented, according to its specification. During the implementation process, additional decisions will have to be made, as the design will not yet hold all information that is necessary for implementing the system. Therefore, the implementation has to be validated and verified again. This is called testing. Finally, some maintenance is required, for when the system is being used, new or changed requirements may come to light, errors may be discovered, etc.

In this monograph, we deal primarily with the first two phases of the development process, viz. analysis and design.

During the past decade, (information) system designers have been confronted with an enormous increase of the complexity of the systems they had to develop. To start with, the systems have become larger. Novel applications have many purposes, contrary to systems that are relatively simple and have limited functionality. Moreover, as the systems to be developed increase, the projects of developing such systems also increase. This requires a good way of communication, in order to avoid misunderstandings between the designers. All this also implies an increase of routine work, which sometimes causes the designers to lose sight of the really important, creative work. When the developer is given the possibility of using a specification method that takes care of (part of) this routine work, he may be able to concentrate on the most important aspects. E.g., a specification method may support reuse of frequently needed components. Finally, it can be observed that in modern information systems, both static and dynamic properties are important and complex. Contemporary IS are neither systems dealing merely with large amounts
of information upon which simple operations have to be applied (e.g. a single-user database with primitive query- and update facilities), nor are they systems with elementary data and more complex dynamics (like a communication protocol).

New systems are developed by integrating data- and behaviour specifications or by applying integrated system concepts, i.e. concepts that intrinsically cover both static and dynamic system aspects. In each development phase, both perspectives (statics and dynamics) should be balanced. Such an integrated development method makes tuning of static and dynamic aspects of a system under design more feasible. Moreover, the modelling process becomes more flexible, as one does not have the rigid and sometimes artificial separation between data and process modelling.

![Diagram](image)

**Figure 1.1: Horizontal versus vertical integration**

In this monograph, we describe an integrated way of developing (information) systems. This integration has two dimensions. First and foremost, it covers what has been called *horizontal integration* [Ramakers & Verkoulen, 1992] (cf. Figure 1.1). This means that modelling static and dynamic aspects of a system has been integrated, at all levels of detail. Although this horizontal integration is the most important one in this monograph, we have also accomplished *vertical integration* to a certain extent: we have designed a formal framework for IS analysis and design, we have developed tool support for this framework and we have dealt with practical application of our approach, by giving some examples and by providing some guidelines stating how our framework could be used in practice.

Our framework has a formal semantics. In our opinion, this is necessary for a sound specification technique. The most important benefit from choosing a framework with a formal semantics is the possibility to obtain *executable specifications*. 
These can be used for simulation purposes, which gives the designer the opportunity to test his design. Of course, simulation does not provide a means for formal verification. However, in practice, simulation is very useful because it enables the designer to discover and repair errors in his design. Moreover, simulation can be used in order to communicate the design to non-experts. A formal specification technique also helps a designer to find omissions, contradictions and ambiguities, because he is forced to define exactly and in full detail what he means. The fact that formal specifications are not ambiguous also makes code generation feasible. A formal technique often leads to a hierarchical development method. Furthermore, a formal semantics makes it possible to develop techniques for formal verification. In fact, in Chapter 5 we present a technique for automatic verification of a design. A final benefit that we want to mention is the possibility to work in a modular way: it is possible to build a specification from smaller models that have already been proven to be correct. Such a correctness proof can only be accomplished when the modules have a formal semantics.

Of course, there are other specification methods that have an underlying formal semantics. However, many of them suffer from one or more problems [Lyytinen, 1987]. The most important problem is the fact that they are often suited for modelling either static or dynamic aspects, instead of both. And even when they are suited for both data and process modelling, they are often still focussed on one of these perspectives. Moreover, many of these formal modelling techniques do not provide the modeller with sufficient tool support or they are intrinsically difficult. On the other hand, modelling techniques that are comprehensible by the designer are often informal, causing the aforementioned problems. It is also important for an IS development technique to be able to describe the environment of the information system for requirements engineering. This is necessary to obtain a better insight in the function of the IS and the way the IS is or will be used. Finally, many existing techniques use private concepts that differ more or less from those used in general practice.

These observations have motivated the initiation of the research project ISDF (Information Systems Description Formalisms). The overall goal of this project was to develop an integrated IS development framework which solves (most of) the identified problems. In this research project, the Dutch universities of Eindhoven, Leiden, Maastricht and Twente and the Centre for Mathematics and Computer Science (Amsterdam) cooperate. The research that we have described here has been conducted as the contribution of Eindhoven University of Technology to this project.

Our work is part of the ExSpect\(^1\) project. This project started in the late 80's. It aims at providing a formal framework for (information) systems design.

\(^1\) ExSpect stands for Executable Specification Tool.
1.2 Outline of the SimCon Approach

ExSpect combines a functional language with a high-level Petri net model. Also, an advanced CASE tool is under development. This tool is commercially available since January 1993. We have used ExSpect successfully in a number of practical situations. Examples of such practical applications are cases in the context of the TASTE project [Van der Aalst, Voorhoeve & Waltmans, 1989], a study at Schiphol Airport and a project with Dutch Railways. In this monograph, we build upon the ExSpect approach.

More details about ExSpect can be found in [Van der Aalst, 1992; Van Hee, Somers & Voorhoeve, 1989; Van Hee & Verkoulen, 1991; Van Hee, Rambags & Verkoulen, 1993; Van Hee, 1993; Houben & Verkoulen, 1991a]. Of course, reading this monograph will also provide more insight.

1.2 Outline of the SimCon Approach

In this monograph, we deal with the SimCon\textsuperscript{2} approach, which we claim to be a framework that meets the aforementioned demands. In our approach, data and process modelling have been integrated in a consistent and comprehensible way. For describing the static aspects of a system, we have designed an object-oriented data model. This object model is used to model the passive entities in a system. For describing the properties of the active entities, a high-level Petri net model has been incorporated. Both models have been integrated at a high conceptual level.

Our approach forms the fundament of a specification tool, which will be explained also. Finally, we provide a means of automatic verification of constraints from the object model in the net model.

In this section, we introduce the components of the SimCon framework. In the following chapters, these components are dealt with in full detail.

1.2.1 The SimCon Object Model (SCOM)

We have developed an object model [Van Hee and Verkoulen, 1991, 1992; Houben and Verkoulen, 1991a, 1992b] that is close to existing and well-known models that are being used in practice, like the Entity-Relationship model [Chen, 1976]. The SimCon Object Model (SCOM) also incorporates the most important object-oriented concepts from [Atkinson, Bancilhon, DeWitt, Dittrich, Maier & Zdonik, 1989; Beeri, 1990], such as object identity and inheritance. SCOM has some resemblance with GOOD [Cyssens, Paredaens & Van Gucht, 1990]. We have kept in mind the purpose of this object model: we did not want to develop yet another object model, but the object model to be used in our framework must be suited for integration with a process model. Therefore, we have incorporated the most important concepts that have proved to be useful in practice, with some extensions

\textsuperscript{2}As can be seen in Chapter 2, the name SimCon contains the prefixes of two important concepts from our model, viz. \textit{simple} objects and \textit{container} objects. Also, it is an acronym of \textit{Simple Integrated Model for Complex Object Networks}. 
for making the integration possible. We have combined these aspects, together with some new features, in a new object model: this way, we have full control over the concepts in our model, which makes integration and verification more feasible. We have defined the formal semantics of SCOM in terms of set theory.

There are two levels in SCOM. The first one is the level of the so-called simple objects. This level is meant for global state space modelling, without paying attention to things like distribution, location and timing. A simple object has a unique identity, relationships with (other) simple objects and attributes. Simple objects are classified into simple types; when the state space of a system has to be described, a simple schema is introduced to define the existing simple types and their properties. Moreover, inheritance between simple object types is supported. The second level is that of the container objects. This level extends the simple level by describing location, distribution and timing aspects. Moreover, it serves as "glue" for making the integration possible. A container (object) has a unique identity, it contains a set of simple objects (not necessarily of the same type) and it has a location and a time-stamp. Again, container objects are classified into container types; the notion of a simple schema is extended to an object schema: besides defining the existing simple types, an object schema also defines which container types exist. On both levels, constraints can be expressed.

1.2.2 The SimCon Algebra (SCA)

It is obvious that it is not sufficient to develop an object model without some means to manipulate objects: we need some language for this purpose. It is preferable that the concepts in this manipulation language are on the same level of abstraction as the object model. The object model describes the structure of the objects, whereas the manipulation language serves for expressing creation, retrieval, update and deletion of objects. Both formalisms are equally important for modelling static and dynamic aspects of a system. It would also have been possible to define the manipulation language at the mathematical level that represents the formal semantics of the object model. However, in that case, it would only be understandable for experts with full knowledge of this formalisation. Although our approach is not intended to be used by laymen who have no background in (formal) computer science, this is highly undesirable: if someone who wants to apply our approach, would have to study and comprehend the complete formalisation in advance and he would have to express some system properties at this level of abstraction, it would not be very clear what is gained by applying the SimCon approach.

To meet these demands, we have introduced the SimCon Algebra (SCA) [Houben and Verkouwen, 1991a, 1992ab]. The SimCon Algebra consists of some basic operations resembling graph manipulations. As will be demonstrated, these operations give sufficient expressive power. In particular, we prove that SCA is at least as expressive as the Nested Relational Algebra (NRA) [Schek & Scholl, 1986; Roth, Korth & Silberschatz, 1988] (and thus also as the flat Relational Algebra) and
GOOD [Gysens, Paredaens & Van Gucht, 1990]. We even describe an operation that cannot be expressed in GOOD [Andries, 1990], but which is expressible in SCA. We explain where this difference in expressive power comes from.

In order to increase the expressive comfort of the algebra, we propose a graphical version, called SCAG. We also have defined some high-level operations in terms of the aforementioned basic operations. As these high-level constructs have been defined in terms of primitive ones, they do not have to be considered in our theoretical contemplations. These high-level operations are introduced in Section 3.4. We may develop libraries of dedicated high-level operations that can be used when modelling special kinds of systems. It has been recognised before that this may support the designer of a system well [Van der Aalst, 1992].

1.2.3 The SimCon Net Model (SCNET)

Actually, the SCNET model is the core of our integrated model. An SCNET is a special kind of Petri net. Below, we describe why we have chosen Petri nets as the basis for modelling dynamic aspects of a system. First, the most important concepts from Petri net theory are mentioned. We also give some references for further reading. This is elaborated in Chapter 4.

Petri [1962] is the founder of Petri net theory. Basically, a Petri net is a bipartite graph consisting of transitions and places. Transitions are active components, that communicate with each other by exchanging so-called tokens. The places connect the transitions. They contain the tokens. When a transition occurs or fires, it consumes tokens from each of its input places and produces tokens for its output places. In classical Petri net models [Reisig, 1985] these tokens do not have a value: the only information they represent is their presence. In high-level Petri net models like Coloured Petri Nets (CPN) [Jensen, 1991, 1992b] and ExSpect, these tokens have a value. In the tools that support these high-level net models, these values are specified using the type system of a functional language, while the functionality of the transitions is specified by functions from that language. However, when developing large and data-intensive applications, this way of defining the state space of the net is not always satisfactory. The main reason for this is that it may be difficult to acquire an overview of the data aspects of the specification. In the SCNET model, the state space of the net is defined in terms of the SimCon Object Model: tokens become SCOM (container) objects. To each place, an SCOM container type is assigned, defining which kind of objects can be contained by that place. Moreover, the functionality of a transition is defined by an SCA expression. As we will see in the chapters to come, this is a natural and dexterous way of working.

We use high-level Petri nets as a basis for describing the dynamic aspects of a system for a number of reasons. We want to mention the most important ones. High-level Petri nets are inherently simple, yet they give sufficient modelling power and comfort, especially because of their graphical nature. Moreover, they have a firm mathematical foundation which is necessary to obtain a formal semantics and
which is a prerequisite for the development of new theories for proving properties of such nets. At the moment, many theoretical results have been accomplished. Last but not least, advanced computer tools have been designed for high-level Petri nets. Two important ones are Design/CPN [Jensen, 1991] and the ExSpect tool [Van Hee, Somers & Voorhoeve, 1991; Van den Broek & Verkoulen, 1992].

1.2.4 Automatic Verification

In SCOM, it is possible to define constraints. When an SCOM schema is used to model the state space of an SCNET, we want to know whether this net satisfies the constraints that have been formulated upon the schema. Again, it would be possible to let the designer perform this verification process at the level of the formal semantics of our models. However, this is cumbersome. We want to support the designer in this verification process. We would prefer a means of automatic verification: in that case, the designer only has to formulate the constraints (which may be difficult enough). The automatic verifier then takes care of the verification.

In Chapter 5, such a method for automatic verification will be described for a certain class of constraints, the so-called compositional constraints. We mention the main idea here. In the verification process, the state space of the SCNET under consideration is divided into a finite number of equivalence classes: in order to check whether the net will violate a constraint, it is checked whether each of the transitions in the net leaves the constraint invariant. In this way, the number of states that have to be checked is reduced drastically because we do not consider all possible interleavings of transition firings. This is a method that has been recognised in literature before. E.g., Huber, Jensen, Jepsen & Jensen [1986] use this technique in the context of Petri nets. However, the number of states that have to be checked this way is still very large (often even infinite). Therefore, the number of relevant states is reduced further, by looking only at a set of characteristic states, called prime states. We prove that checking the prime states is necessary and sufficient to prove (or disprove) the invariance of a compositional constraint. This set of prime states is relatively small. In particular, it is small enough to do the checking of the constraints in a reasonable amount of time.

1.2.5 Tool Support

The tool that supports the SimCon approach is based upon the ExSpect tool. The ExSpect tool is extended by an editor for the SimCon Object Model [Van den Broek & Verkoulen, 1992]. Moreover, we indicate how automatic verification of constraints as described in Chapter 5 can be supported by this tool.

The tool supports hierarchical Petri nets as it is possible to define subnets. This subnet concept has also been introduced in the context of Coloured Petri Nets; there, a subnet has been called a page [Jensen, 1992a]. A subnet consists of

\footnote{In ExSpect terminology, a subnet is called a system.}
transitions, places and subnets. This construct allows for top-down development
and hiding of implementation details. Of course, bottom-up development of a spec-
ification is also possible. The SCNET model is not hierarchical; the semantics of
hierarchical nets can be expressed in terms of flat nets [Verkoulen, 1989]. Thus,
it is not necessary to make the formal SCNET model more complicated by incor-
porating this notion of hierarchy. Future research might indicate that it would be
better to cover the hierarchy concept also at the level of the SCNET model.

At the moment, the object model editor and the net editor are coupled rather
loosely. Eventually, it will be possible to define which objects can be contained by
a certain place by clicking on that place and on the desired container type in the
respective schema.

1.3 A Small Example

In this section, we present a small example which illustrates our approach. In
Chapter 4, the example is worked out in more detail (Example 4.2.1). Section 6.2
contains a larger example.

In Figure 1.2, a graphical representation of a SimCon model that partly specifies
a transport company, is given. We suppose that the company owns a number of
trucks and employs some drivers. It carries bulked goods. In order to keep the
example simple, we assume that trucks may be either filled completely or totally
empty. Upon receiving an order from a client, the company schedules a trip,
consisting of a truck and a driver that has a licence for that truck. The empty
truck is loaded at the storehouse and then the trip is executed. After a while, the
truck (that is now empty again) returns at the company: the empty truck is put
into the garage and the driver goes into the waiting-room.

Figure 1.2 gives a graphical representation of a SimCon model that describes this
situation. In the data part, we see that there are three simple types driver, truck
and order, respectively with attributes name, capacity and quantity. More-
over, a driver has a licence for a number of trucks and an order is carried
by a truck. Note that we do not make a difference here between the references
licence for and carried by. However, there is a difference: in general the licence
information of drivers is more or less static (persistent), whereas the data about
which order is carried by which truck is liable to continuous fluctuations. In a more
extensive example, this may be expressed by modelling the storage and possible
changes of the licence information explicitly, instead of simply assuming that each
driver knows for which kinds of trucks he has a licence.

In the net part, we see places and transitions. The places may hold container
objects, which are clusters of simple objects. Some of them are singletons, like
the trucks in the garage. Others contain more than one simple object, like the

\footnote{This assumption is not as artificial as it may look: for certain types of trucks, this constraint
really holds, as a half-full truck might topple.}
Figure 1.2: The Trip Example
trips in manned-truck: a trip consists of a driver and a truck. The behaviour of the transitions is defined in terms of the SimCon Algebra: the functionality of a transition is described by an SCA expression. Moreover, SCA is used for formulating preconditions; a transition may only fire when its precondition is satisfied. In the example, the transition TripPlanning may only fire when it can "consume" a truck from the garage with sufficient capacity for the order and a driver from the waiting-room that has a licence for that truck. This is formalised by the following precondition:

\[
\text{order[quantity]} \leq \text{truck[capacity]} \land \text{truck[}] \subseteq \text{truck[license_for driver []]}
\]

The expressions that define the transitions are trivial. More details can be found in the next chapters.

1.4 Position of this Research

In this section, we position our work in the field of related research. We first mention some models that are meant for modelling statics or dynamics. Afterwards, we deal with some approaches which try to integrate data and behaviour modelling.\(^5\)

During the past 20 years, many models that cover either the data or the process perspective have been developed. Well-known data models are the relational model [Codd, 1970] and the nested relational model [Schek & Scholl, 1986; Roth, Korth & Silberschatz, 1988]. Many theoretical results have been described for these models. Moreover, they have been used extensively in practice. One of the most famous query languages for the relational model is the Structured Query Language (SQL) [Date, 1988]. The relational approach has become widespread because the underlying idea of storing data in tables is intuitively clear. Moreover, the relational approach can be implemented efficiently. However, the relational model lacks graphical support. Moreover, a state consisting of many large tables is not always surveyable. Therefore, semantical data models like the Entity-Relationship model [Chen, 1976], the Functional model [Shipman, 1981; Aerts & Van Hee, 1988] and IFO [Abiteboul & Hull, 1987] have been developed. The introduction of these models did not solve the problem that one entity sometimes has to be modelled by a complex construct that cannot be identified as one object. To solve this problem, object-oriented models have been developed. Some examples are O2 [Deux, 1991], GOOD [Cysens, Paredaens & Van Gucht, 1990] and TM [Balsters, De By & Zicari, 1991]. These object-oriented data models also try to incorporate dynamic aspects, usually by introducing the method concept: objects do not only have attributes and/or relationships with other objects, but also contain methods which they can execute. Objects are triggered to execute a method when receiving a message from

\(^5\)When first reading this section, it may be difficult to judge whether the claims are justified, because the exact definition of the SimCon framework has yet to be read. It may be a good idea to reread this section after having read the remaining chapters of this monograph.
an object. Although such extensions form a firm step towards an integrated model in which static and dynamic aspects of a system can be modelled, these object-oriented models are often still better suited for modelling systems for which the data aspects are more important than the dynamic aspects, as they remain (extended) data models.

There are also many models that aim primarily at describing the dynamic aspects of a system. Dataflow Diagrams (DFD's) [Yourdon, 1989] are used often in practice, but they lack a formal semantics. On the other hand, algebraic approaches like CCS [Milner, 1980] and ACP [Baeten & Weijland, 1990] do have a formal basis, but they are not easy to use in practical situations, especially when the designer does not have a firm mathematical knowledge. Classical Petri nets [Reisig, 1985] are more easy to use, because they are intrinsically graphical. However, modelling a realistic application gives a net with an enormous number of transitions and places. In order to avoid this explosion, high-level Petri net models have been developed. We mentioned already CPN and ExSpect. These models tend to an integration with data concepts, as they provide a (functional) language to describe the state space of the net and the functionality of the transitions. However, where object-oriented data models with methods still remain data models, these high-level Petri net models still have some problems when describing systems with a large and complex state space, as it may be difficult to get an overview over a large set of complex type definitions. The same holds for declarative methods like Z [Hayes, 1987] and VDM [Jones, 1987]. Moreover, such declarative methods are not well-suited for describing concurrency and time aspects [Ostroff, 1991].

There are some other approaches that try to integrate data and behaviour modelling. We already mentioned high-level Petri nets. Another example has been described by Heuser & Meira Peres [1990]. They extend an Entity-Relationship model with facts (transitions that may never fire) in order to express dynamic constraints. This is not a real integration of data and process modelling: the facts only describe events that are forbidden, but the events which may occur are not specified.

Engels [1990] also proposes an Entity-Relationship approach. In his Extended Entity-Relationship model, he extends the ER-model with elementary actions that can be used to update an instance of an ER-schema. A database designer can use these elementary actions to compose complex actions. However, this approach is still a data-oriented one, which does not integrate statics and dynamics on an equal basis.

In [De By & Steenhagen, 1992], a first approximation for integrating TM and LOTOS is described. TM is a formal object-oriented data model, LOTOS is a process algebra formalism. De By & Steenhagen [1992] propose to model a database in terms of interacting processes. Here, the process structure is described using LOTOS, whereas the actions in the processes and the structural aspects are expressed in terms of TM. This approach is promising, but much work still remains
to be done, as this is just a first approximation.

The approach of Ramackers & Verrijn-Stuart [1991] describes a model that integrates what they have called the "data, process and behaviour perspectives". This is done in an object model extended with concepts like life cycles. However, this model lacks a formal semantics. In [Ramackers & Verkoulen, 1992], we have mapped this object model onto the SimCon framework. In this way, a formal semantics is obtained. From our perspective, this combination provides a way of working within the SimCon framework. More details about this joint research can be found in Section 6.3.

The work of Kappel & Schreif [1991] can be compared to our work that is described in Section 6.3 and in [Ramackers & Verkoulen, 1992]. Kappel & Schreif [1991] introduce so-called Object/Behaviour Diagrams. An object diagram describes which kinds of objects may exist in a certain system. A behaviour diagram describes the life cycle of the objects of exactly one object type. So-called activity specification diagrams describe the interactions between objects. The semantics of the model is described in terms of Predicate/Transition Nets [Genrich, 1987], which resemble Coloured Petri Nets. A system can only be specified in terms of objects and their life cycles. This way of working is not always the most preferable one, especially when the dynamic aspects of a system are the most important ones.

Becker & Moldt [1993] describe how object-oriented concepts can be expressed in terms of Coloured Petri Nets. So, they sketch a way of applying Coloured Petri Nets. Their approach resembles the work that we describe in Chapter 6. However, they do not cover all important object-oriented concepts: e.g., they lack an explicit construct for life cycle modelling.

An approach that is similar to the work of Becker & Moldt [1993] has been described in [Battiston & De Cindio, 1993]. This work clearly resembles our approach from Chapter 6. However, this model lacks sufficient support for specifying the static aspects of an object.

In [Sibertin-Blanc, 1985], another approach for integrating Petri nets and object-oriented concepts is described. This approach also integrates Petri nets and a kind of Entity-Relationship model. Although the work is interesting, it also lacks some features. The most important problem is that the integration is too vague and not on a conceptual level. For example, it lacks a data manipulation language like SGA: the transitions have to be defined in terms of mathematics. Issues like global state space modelling versus location, distribution and timing aspects are not being addressed. Finally, Sibertin-Blanc [1985] does not provide insight in how this approach could be used in practice.

Lakos & Keen [1993] describe the LOOPN framework, which stands for "Language for Object-Oriented Petri Nets". This approach is similar to the ExSpec approach. The most interesting extension is the feature of inheritance of subnets. The language has been implemented, but it is not supported (yet) by an advanced tool.

In the DeVise project [Christensen & Tokevig, 1993], one aims at integrating the object-oriented programming language BETA with the high-level Petri net model
CPN. For the moment, this integration is accomplished by making it possible to include BETA code within so-called code segments in the net. A code segment of a transition is executed each time the transition occurs. At least at the level of the corresponding tool, this gives a better combination than when they would have used BETA code as arc inscriptions. However, at a conceptual level, the latter may be preferable. As this approach is in a first research stadium, this may be changed in the future.

Richter & Dürchholz [1982] describe a model in which Predicate/Transition Nets and the so-called Information Management Language (IML) are combined. In this approach, IML is used to define the functionality of the transitions in the net. However, in our opinion, IML is a language that is too difficult to be used or even understood by a large group of users.

We conclude this section by giving some alternative ways in which data and process modelling could have been integrated in the context of ExSpect.

In [Verkoulen, 1990], a first approximation has been described. In that approach, a data schema is defined for each place in a Petri net. This schema is translated into an ExSpect type definition for that place. ExSpect functions have to be developed by the designer in order to manipulate tokens in that place. In fact, a graphical syntax for the ExSpect type system has been described. Although this solves the problem of the non-graphical nature of the ExSpect type mechanism, it does not solve the most important problems. E.g., it is still difficult to comprehend the relationship between the types of various places. A disadvantage that is even more severe is the fact that the designer has to develop the functions manipulating the objects all by himself. Although he can be given some support by generating some standard functions, this remains a difficult task.

In the ESPRIT-II project PROOF, it has been tried to integrate ExSpect with the information modelling method NIAM [Wintraecken, 1990]. This attempt is described in [Ter Hofstede, Van Oosterom & Renardel de Lavalette, 1991]. In this approach, the ExSpect stores (which are special kinds of places that always contain exactly one token) are typed by a NIAM subschema. In fact, this is the same solution as has been sketched above [Verkoulen, 1990]. This integration is much too loose to be the final solution. E.g., as a separate schema has to be defined for each store, it is difficult to get an overview of the relationships between these stores. Even worse is that the same NIAM type may have different meanings for different stores in the same system. Ongoing work within the PROOF project may solve these problems in the future.

In [Beelen, Stut jr. & Verkoulen, 1992], we have described the integration of the data model MOVIE [Stut jr., 1992] and ExSpect. MOVIE is a semantic data model. The most important concept is that of an object view: instead of developing one schema for an application, a subschema is developed for each object type. Such a schema represents the view that an object of that type has on the system. Moreover,

---

4PROOFs stands for “Promotion of Formal Methods in European Software Industry”.
active and passive objects and relationships are distinguished. In order to be able to model the dynamic aspects of a system, a transformation from a MOVIE model to an ExSpect specification has been described. This means that for each MOVIE model, a Petri net is defined. This net consists of a transition for each active object and it contains tokens representing the passive objects. The structure of these tokens and the functionality of the transitions are described in terms of the ExSpect functional language. However, as this approach is still focussed on the data perspective, it will probably not be generally applicable. Another reason for this is that the structure of the generated Petri nets is of a slightly artificial nature and does not always model the dynamic aspects of a system in an intuitive way.

1.5 Outline of this Monograph

In the previous section, we have sketched the SimCon approach. In the remainder of this monograph, we explain this approach in full detail. This monograph consists of two parts. The next three chapters introduce the components that constitute the SimCon model. To be more precise, we introduce the SimCon Object Model in Chapter 2. This object model has an associated manipulation language, the SimCon Algebra. This algebra is dealt with in Chapter 3. Finally, the SimCon Net Model is introduced in Chapter 4.

The second part of this monograph consists of Chapters 5 and 6. In Chapter 5, we deal with a method for automatic constraint verification within the SimCon approach. This is one of the benefits of providing an integrated model. Chapter 6 describes some guidelines which could be used when applying SimCon in practise and it contains some examples of how our approach can be applied.

Chapter 7 concludes this monograph. It summarises what we have accomplished and it gives suggestions for further research.
Chapter 2

The SimCon Object Model

The first important pillar of the SimCon approach is the SimCon Object Model (SCOM). The name SimCon contains the prefixes of two important concepts from the SimCon Object Model, viz. simple objects and container objects. In this chapter, we first describe related work and we motivate why a new object model has been developed, instead of taking an existing one. Afterwards, we describe the SimCon Object Model, both formally and intuitively. We also pay some attention to tool support. We illustrate the definitions by means of examples.

2.1 Introduction

Within the SimCon approach, SCOM has been defined for describing the static aspects of a system under consideration [Van Hee & Verkoulen, 1991, 1992; Houben & Verkoulen 1991a, 1992b]. Of course, we could have used an existing data model for this purpose. However, we have chosen to define a new model, for several reasons. We deal with each of them below. SCOM covers the most important concepts that have been recognised in database literature, as is shown in this chapter.

One of the fundamental data models in database literature is the relational model [Codd, 1970]. In the relational approach, information is organised in terms of mathematical relations. An instance of a relational schema can be seen as a table. In order to solve the problem that values in the relational approach are atomic (which means that the only operation that is defined upon them is the equation), the nested relational model has been defined [Schek & Scholl, 1986; Roth, Korth & Silberschatz, 1988]. Many theoretical results have been described for both the relational and the nested relational model. Moreover, they are being used extensively in practice. SQL is one of the most famous query languages for the relational model [Date, 1988]. The underlying idea of storing data in tables is intuitively clear. However, the relational model lacks graphical support. Moreover, a state consisting of many large tables is not always well surveyable. Finally, the relational model is value-oriented. This means that an entity in the system
under consideration can only be described by values. If two objects have the same values, they cannot be distinguished and hence they are the same. Even worse is the fact that two different values or value combinations cannot be thought of as representing two versions of one and the same entity. E.g., when a person moves to a new address, a completely new person “object” is created in a value-oriented approach. This new object has “by accident” the same name as the old one, but in a value-oriented approach, they cannot be considered as one and the same object at different moments in time.

These disadvantages do not matter for the original usage of the relational model, viz. for developing administrative applications with emphasis on the static aspects of a system. However, in contemporary systems development, the relational model will often no longer be sufficient.

In order to overcome (some of) these disadvantages, so-called semantic data models like the Entity-Relationship model [Chen, 1976] and IFO [Abiteboul & Hull, 1987] have been developed. The introduction of these models did not solve the problem of value-orientation. For this purpose, object-oriented models have been developed. It would be out of the scope of this monograph to mention them all here, but some examples are O2 [Deux, 1991], GOOD [Gyssens, Paredaens & Van Gucht, 1990], TM [Balsters, De By & Zicari, 1991], and the work of Witt [1988]. As can be seen in this chapter, the SimCon Object Model incorporates the most important object-oriented concepts.

Below, we describe the requirements for developing the SimCon Object Model. These requirements also motivate why we did not use an existing data model.

One of the main demands was that the model should be as simple as possible. This means that we do not want to incorporate concepts or conventions that are not strictly necessary. This simplicity has two main advantages. First, integration with other models (in our case, a process model) is more easy when the models to be integrated are as simple as possible. Moreover, in our opinion the “kernel” of a model should be as simple as possible in order to be understandable and usable in solving practical problems. Whenever necessary, high-level modelling constructs may be provided that can be expressed in terms of this kernel, e.g. by incorporating them in the tool supporting the approach. Of course, a model with few fundamental concepts can be formalised more easily than an overloaded model.

This brings us to another advantage of developing our own model. We strongly feel the need for having a formal model. For the only proper way that two or more models or modelling methods can be integrated is when there is an exact and unambiguous semantics. Otherwise, it is impossible to state exactly what a certain model means. Also, combination with other models is out of the question then. Other advantages of a formal approach have already been mentioned in Chapter 1.

Besides formality, it is important that a model is understandable and usable by its intended users. Note that this is not the same as simplicity, although those two notions are closely connected: a model is simple if it has few fundamental
concepts, but this is no guarantee for understandability. The same holds vice versa. If a modelling method cannot be comprehended by the user, it can only be useful for purely theoretical purposes. Therefore, we have provided a graphical representation for the SimCon Object Model. The SimCon CASE tool [Van den Broek & Verkoulen, 1992], which extends the ExSpect tool, supports this graphical representation. More details about facilities for object modelling provided by this tool can be found in Section 2.4.

An important advantage is the fact that it is convenient when we control both methods we want to integrate, i.e. we are free to adapt concepts whenever necessary. This is the only way to avoid strange conventions in an integrated model that arise from the fact that certain concepts were inherited from one of the component models and could not be changed. This is important, as integration is the most important contribution of our work.

2.2 Informal Description

The SimCon Object Model consists of two layers. The first one is a binary version of the Functional Model [Shipman, 1981; Aerts & Van Hee, 1988], extended with object-oriented concepts. This first level is meant for describing the global state space of a system. At this stage, distribution and timing aspects are not yet taken into account. The second layer covers these aspects. Moreover, it allows for aggregation of objects from the first level. The formulation of constraints is supported. We have not (yet) developed a private constraint language, but we use first order predicate logic. To some extent, the SimCon tool makes it possible to formulate constraints graphically. In the remainder of this section, we give an extensive explanation of the concepts of the SimCon Object Model. We illustrate these concepts by means of examples.

The first layer of SCOM is that of the so-called simple objects. This layer is meant for describing the global state space of a system: it can be used to model which kind of entities may occur in the system under development. For this purpose, each entity in the system is represented by a simple object. The properties of these entities are modelled by assigning attribute values and references to (other) simple objects to these simple objects. The references actually implement the concept of relationships between entities. We distinguish between attributes and references as an attribute value models some property of an object, whereas a reference models the knowledge an object has of other objects. Although this difference may be subtle, we think it is important enough to justify the introduction of an extra concept in our model. Moreover, each simple object has an object identity. This means that each simple object has a unique identification that is never changed, although the (other) properties of the object may change. This is very natural: when the properties of an entity change, we often still want to consider it as the same entity and not as a completely new one. E.g., when a person moves
to another country, he still remains the same person. He is not born again! So, there is a fundamental difference between an object identity and an attribute value, as duplicate values are allowed, whereas this is not the case for object identities. The data model of [Van Hee, 1993], which is a derivate of the SimCon Object Model, does not require object identities to be unique. So, his model allows for duplicates of objects. Only one of these duplicates may be changed, the other ones are "read-only". Although this way of working can be useful when modelling information objects (like cards in a card-tray), we want to adhere to the uniqueness of objects in order to preserve the valuable concept of unique object identities.

The attribute values of a simple object are values that have a clear meaning in the perception of the designer, like the string Peter, the number 2679 and the set \{VW, BMW\}. Attributes are modelled by means of the ExSpect type system that has been refined in [Verkoulen, 1989], which allows for set-, pair- and tuple-types, subtyping and polymorphism. Note that attribute values may belong to a complex type. The designer has to decide whether he wants to describe some construct by means of an attribute or by means of a simple type. Moreover, he has to decide about the granularity of the type definitions. In some applications, it may be sufficient to have one address type, whereas in others this may be split into street, house number and residence. In our opinion, it is desirable to be able to distinguish objects with possibly changing properties and values which do not represent any other information than their value (and which cannot change consequently). We think that this difference between evolving objects and never-changing values can be very important. Moreover, there is a more philosophical difference between entities like persons, cars and machines and properties like names, colours and capacities. Finally, as the concept of object identities may conflict with declarativeness and encapsulation [Ullman, 1988], we want to support the use of values (without an identity) in all cases where this is the most natural choice. Van Hee [1993] does not make this distinction between objects and values.

![Figure 2.1: Two memory managers with some memory blocks](image)

Figure 2.1: Two memory managers with some memory blocks
The references of a simple object to other simple objects are used to model the relationships of simple objects with each other. A reference consists of a reference name and a set of object identities of simple objects that are being referred to. For example, a person may have a reference owns to each car he possesses. In a global database model, there is no fundamental difference between references that are realised by identifications or by incorporating the referenced objects: all information is always available. However, in a distributed environment, there is a difference between knowing and being able to access objects. This difference is illustrated in Figure 2.1. The figure shows two container objects¹, each containing a memory manager (MM) and some memory blocks (MB). There is a reference r between the memory managers and a reference c between a memory manager and each memory block that is controlled by that manager. We suppose that these blocks have some properties, like a size, but we have not depicted them in the figure. From each memory manager, the information of all blocks that are in the same container, can be accessed directly. However, if a manager needs some information about the memory blocks that are controlled by one of its “colleagues”, this can only be accomplished by “asking” it to this colleague, which is able to access the properties of the memory blocks under its control.

More details are described below. The model of Van Hee [1993] does not make this difference between knowing objects and being able to view their properties.

A so-called simple schema is used for defining which kinds of simple objects exist in a system, together with their possible attributes and references. All objects with similar properties are classified in one simple type. The attributes and references are not compulsory, i.e. for a simple object of a certain type, not all allowed attributes and references have to be known at each moment in time. As well as using a simple schema to define which simple types exist, it can also be used for describing inheritance. This means that a simple type may be a specialisation of one or more other simple types, i.e. it inherits implicitly the attributes and references of its generalisations. Moreover, additional attributes and references may be defined. Overriding is possible when the specialisation has an attribute or reference with the same name as that of a generalisation; in that case, the definition that is inherited from the generalisation is overridden by the new definition. The designer has to avoid name clashes (the inheritance of different definitions of a reference or attribute from different generalisations). Moreover, a simple type may never inherit from itself, neither directly nor indirectly. In other words, the inheritance relationships have to be acyclic. We apply inheritance as a short-hand construct which makes it possible to reuse definitions.

In the examples, we use the following graphical conventions: simple types are depicted as rectangles with rounded corners, attributes as circles; relationships be-

¹Container objects are introduced at page 35. For the moment, think of a container as a cluster of simple objects.
tween simple types are denoted by labelled directed double-headed arcs; attributes are connected to the simple type(s) they belong to using undirected unlabelled edges; if the attribute types are indicated in the pictures, they are written near the attributes (in the SimCon tool, these types can be viewed and changed when required by clicking on the corresponding attribute\(^2\)), when a simple type A inherits from some simple type B, this is depicted by a boldface arrow from A to B. Some cardinality constraints can be denoted by putting special symbols at the head or the root of the reference arrows. Others have to be defined textually. These graphical conventions have also been used in the SimCon CASE tool that has been described in [Van den Brock & Verkouwen, 1992] and in this monograph.

**Example 2.2.1 (simple schema)** We consider a schema (cf. Figure 2.2) with three simple types, viz. person, student and car. A person may have three attributes, viz. name, street and city. A car may have a type and a speed. The type of speed is num; all other attributes have type str. Furthermore, each person owns a set of cars. Finally, the simple type student inherits all attributes and references of person (a student is a person). So we may refer to the city where a student lives and to the cars he owns. Furthermore, a student may have an additional attribute year (of type num), representing the year in which he entered university.

![Figure 2.2: An example simple schema](image)

Given a simple schema \( \Sigma \), the simple object universe \( S_\Sigma \) is the set of all simple objects according to \( \Sigma \) that only have references and attributes according to \( \Sigma \).

We allow the formulation of constraints. Constraints allow the modeller to express additional information about the properties of the objects in the system.

\(^2\)When using the tool, one would introduce a new type name for each attribute, denoting some (complex) type definition.
that cannot be described solely by introducing types, attributes and relationships: constraints are used to express additional requirements about the attributes and relationships of certain simple objects. Some constraints can be defined graphically, the other ones have to be defined textually. We distinguish local constraints that have to hold separately for all objects of a certain type and global constraints ranging over the complete state of the system. A local constraint in the context of the schema of Example 2.2.1 is the following: the speed of a car may not exceed 300. An example of a global constraint is the requirement that no two different persons may own the same car.

Most constraints have to be expressed by means of a formula. The following constraints can be formulated graphically:
- the functionality constraint: a reference \( r \) from \( A \) to \( B \) is functional if each \( A \) has at most one \( r \)-reference (graphical convention: a single-headed arrow);
- the injectivity constraint: a reference \( r \) from \( A \) to \( B \) is injective if the sets of \( B \)-objects to which two different \( A \)'s have an \( r \)-reference, are disjoint (graphical convention: a vertical bar through the arrow);
- the surjectivity constraint: a reference \( r \) from \( A \) to \( B \) is surjective if each \( A \) has at least one \( r \)-reference to some \( B \) (graphical convention: a bullet at the foot of the arrow);
- the bijectivity constraint: this is a combination of the injectivity and the surjectivity constraint;
- the totalness constraint: a reference \( r \) from \( A \) to \( B \) is total if each \( B \) has at least one incoming \( r \)-reference from some \( A \) (graphical convention: a bullet at the head of the arrow).

In Chapter 5, we describe a method for automatic verification of a certain class of constraints. It is possible to implement this method for automatic constraint verification in the SimCon tool. There are also constraints that do not belong to this class. These must be verified or at least validated by the designer. This can be done either by giving a formal proof (verification) or by using simulation (validation), using the tool. In the latter case, we provide assistance in that the tool may give a warning when a constraint is violated (when the tool is in constraint-validation mode). Some remarks about these facilities can be found in Section 5.5.

The SCOM features that have been introduced so far allow for global state space modelling. However, situations where a cluster of entities exists during a period of time cannot be modelled (in a natural way). An example of such situation is a trip made by a person and a car. Also, modelling the location of objects is not being supported at the simple level, so this would have to be described explicitly in each application where it is relevant. The same holds for timing aspects.

In order to cover these aspects, a second layer has been added to the SimCon Object Model, viz. that of the container objects. Basically, a container (object) is
an aggregate of simple objects: it contains one or more simple objects, possibly of different types. Moreover, it has a unique identity, a location and a time-stamp. The location defines where in the target system the container resides. E.g., a car object may be located in a garage. The time-stamp models when the container will be "born". When the time in the system is earlier than this time-stamp, the container does not really exist yet. This can be compared with an unborn child in his mother's womb. In Chapter 4, it will become clear how to use this concept.

In the integrated framework, the location and the time-stamp of an object will be modelled implicitly. However, we need to make them explicit here in order to allow for this integration.

At the container level, the difference between knowing the identity of an object and knowing all properties of an object can be made. To our knowledge, there are not many data models that have made this difference between knowing and accessing before. If an object s has a reference to an object s' in the same container, all information of s' is "visible" from s. However, if they are in different containers, s has only the information that it knows some object of the type of s'. In Chapter 4, we will see that Petri net transitions are used to transform the objects. Only if the containers that hold s and s' are processed together by one transition, the properties of s' become visible for s again. When modelling systems where concepts like location, information hiding, etc. are important, this difference is a useful and important concept. There are many situations where it is useful, for specifying that some information is hidden for some entities (but not for others) during a certain period of time.

In order to be able to define which kinds of container objects may exist in a system, the notion of a simple schema is extended to that of an object schema. An object schema contains all information of a simple schema. Moreover, it defines all container types by assigning names to clusters of simple types. When the direction of the references is left aside, such a container type must be connected: we do not allow a container type to contain two simple types that have no (indirect) relationship whatsoever.

In Example 2.2.2, the simple schema from Example 2.2.1 is extended to an object schema. In this monograph, we use the following graphical conventions for defining object schemas. The conventions regarding simple types have already been mentioned. A container type is defined by drawing a closed line around the contained simple types and writing the name of the container type near this line. In case of a singleton container type (a container type consisting of one simple type), the line may be omitted and the name may be written near the box of the contained simple type. These graphical conventions slightly differ from the conventions that have been used in the SimCon editor [Van den Broek & Verkouwen, 1992].

Given an object schema Σ which is an extension of a simple schema S, the container object universe C_Σ is the set of all container objects according to Σ that
contain only simple objects from $S_S$. Moreover, the types of all contained simple objects in a container $c$ have to be an element of the contained simple types of the container type of $c$. The state space $SS_S$ of $S$ consists of certain sets of container objects from $C_S$. In a state, the container objects give a partitioning of the simple objects. In other words, each simple object is part of exactly one container object; container objects may not overlap. Note that this does not hold for simple types: they may overlap. Also, it is possible that a simple object moves from one container to another, possible of another type.

**Example 2.2.2 (object schema)** Consider the schema from Figure 2.3. The definition of the simple types has been copied from Example 2.2.1. Moreover, it contains the container type trip (consisting of person and car) and the singleton container type class, containing student. As a student is a person, a student is allowed to participate in a trip too.

![Diagram of an example object schema](image)

Figure 2.3: An example object schema

In Section 6.2, we present an example of a complete SimCon model. Part of this is a SimCon Object schema. So, if you are interested in reading another example, please refer to Section 6.2.

Analogously to the possibility of defining constraints on a simple schema, it is also possible to define constraints on an object schema. Besides the constraints that have been mentioned already, one can also define constraints on container types. An important one is the so-called free- or root-constraint. This constraint
may be imposed upon a container type that is a tree: one simple type is the root of
the tree and all other contained simple types are reachable via directed references
from this root. When we impose the tree constraint upon such a container type,
it means that all containers of that type should be a tree too: they should contain
one simple object of the root type and all other contained simple objects should
be reachable from this root.

Note that a container object is built from simple objects and not (recursively)
from containers again. The main reason for this is that we want a clear distinction
between defining the properties of the objects at a global level and the descrip-
tion of distribution and timing properties. However, this does not mean that
(dis)aggregation is not possible: using the structure of a container (as defined in
the object schema) the situation where a container consists of containers can be
modelled. Especially when developing large applications, this way of defining con-
tainers recursively may be useful. An example of a situation where this can be
useful is the following: suppose we have a container type bag consisting of the
simple type letter. When a container of type train has to be loaded with bags,
we want to include container objects of type bag into a container object of type
train. We show how this can be expressed in our model in Section 3.4 and describe
high-level operations providing this functionality. This simulation is such that the
train can be defreighted, giving the same bags it was loaded with: no information
about which bag holds which letters will be lost.

2.3 Formal Definition

This section contains the formalisation of the SimCon Object Model. If the reader
is not (yet) interested in this formalisation, he may skip this section (for the mo-
ment) and turn to the next one.

The first SCOM concept we formalise is that of a simple schema. A formal rep-
resentation of a simple schema should define which simple types exist and which
attributes and references that simple objects of such a type may have. Moreover,
the inheritance relationships between these simple types should be described ex-
licitly. Finally, each schema is defined in the context of some set ID of object
identifications.

Definition 2.3.1 (simple schema) A simple schema consists of a tuple of the form
(SY, AT, RF, IS) where:

- ST is the set denoting the simple types.
- AT = (AN, AS, DD): the attribute structure.

Here AN is the set of attribute names and AS ∈ ST → (AN → P(DD)) is a
function\(^3\) assigning to each simple type and each suitable attribute name the

\(^3\) The "\(\rightarrow\)" symbol denotes a partial function.
set of all possible attribute values, which is a subset of the attribute domain set $DD$. For $s \in ST$ and $a \in \text{dom}(AS.s)^5$, we call $AS.s.a$ the attribute domain of attribute $a$ of type $s$.

- $RF = \langle RN, RS \rangle$: the reference structure.
  Here $RN$ is the set of reference names and $RS \in ST \rightarrow (RN \rightarrow ST)$ assigns to each simple type $t$ a simple type $s$ of each suitable reference name $a$ of simple type $t$.

- $IS \in ST \rightarrow P(ST)$ defines the inheritance structure.
  For each simple type $t \in ST$ the set $IS.t$ denotes the simple types of which $t$ is a specialisation. Note that this allows for multiple inheritance.

A simple type is not allowed to inherit directly or indirectly from itself (acyclicity constraint):

$$(\forall t \in ST : t \not\in (\bigcup_{n=1}^{\infty} IS^n.t))^6$$

Furthermore, name clashes have to be avoided:

$$(\forall t \in ST : -(\exists n \in \mathbb{N} : (\exists t_1, t_2 \in IS^n.t : t_1 \neq t_2 \land (\text{dom}(AS.t_1) \cap \text{dom}(AS.t_2) \neq \emptyset \lor \text{dom}(RS.t_1) \cap \text{dom}(RS.t_2) \neq \emptyset)))).$$ 

This means that we allow overriding of attributes and references, but we prohibit situations in which no deterministic choice can be made.

Example 2.3.2 (simple schema formalisation) The simple schema from Example 2.2.1 is represented formally by the tuple $\langle ST, AT, RF, IS \rangle$ where

- $ST = \{\text{car, person, student}\}$.
- $AT = \langle AN, AS, DD \rangle$ where
  - $AN = \{\text{type, speed, city, street, name, year}\}$;
  - $AS$ consists of exactly the following elements:
    - $AS.\text{car.speed} = \mathbb{N}^+$
    - $AS.\text{car.type} = \text{str}$
    - $AS.\text{person.city} = \text{str}$
    - $AS.\text{person.street} = \text{str}$
    - $AS.\text{person.name} = \text{str}$
    - $AS.\text{student.year} = \{1900, \ldots, 2100\}$

- $RF = \langle RN, RS \rangle$ where
  - $RN = \{\text{owns}\}$
  - $RS$ consists of exactly the following element:
    - $RS.\text{person.owns} = \text{car}$

---

4 In the SimCon tool, these sets of attribute values are modelled by means of a type system. However, at the level of this formal model, we do not want to make this explicit, as other languages for describing attribute values would be applicable too.
5 Function application will be denoted by $f.x$ instead of $f(x)$.
6 Here, $IS$ is generalised for sets: $IS.\{x_1, \ldots, x_m\} = \{IS.x_1, \ldots, IS.x_m\}$. 
• IS contains exactly the following elements:

\[
\begin{align*}
\text{IS.car} & = \emptyset \\
\text{IS.person} & = \emptyset \\
\text{IS.student} & = \{ \text{person} \}
\end{align*}
\]

In the next definition, we introduce the subtype ordering \( \leq \) upon simple types that is induced by the is-a relationships in a schema. E.g., in the schema of Example 2.3.1, it holds that student \( \leq \) person.

**Definition 2.3.3 (subtype ordering (\( \leq \) on simple types)** Suppose that \( \Sigma \) is an object schema as in Definition 2.3.6. Then, it holds for \( T_1, T_2 \in \text{ST} \) that \( T_1 \) is a subtype of \( T_2 \) (\( T_1 \leq T_2 \)) if

\[
T_1 = T_2 \lor (\exists T \in \text{IS.T} : T \leq T_2)
\]

In order to be able to determine the inherited attributes and references of a simple type, the next definition is necessary. The reader may verify that the set of inherited attributes of student in Example 2.3.1 is \{name, street, city\}.

**Definition 2.3.4 (transitive closure of AS and RS)** Suppose that \( \Sigma \) is a simple schema \( \langle \text{ST, AT, RF, IS} \rangle \). Then we define the transitive attributes and references of a simple type \( t \in \text{ST} \) as follows:

- \( \text{AS.t} = (\bigcup_{n=0}^{\infty} (\langle a, D \rangle \in \text{AS.t}' \mid t' \in \text{IS}^{n}.t \land \\
\neg(\exists m \in \mathbb{N} : m \leq n : (\exists t'' \in \text{IS}^{m}.t : a \in \text{dom}(\text{AS.t}'') \land t'' \neq t'))) \)
  
  So, \( \text{AS.t} \) associates with all direct and inherited attributes of \( t \) an attribute domain. Moreover, overriding is supported as for each attribute, the definition which is the "closest" to \( t \) is chosen. In case of name clashes (\( t \) inherits an attribute \( a \) from two (different) types at the same "distance" of \( t \), \( \text{AS.t}.a \) is undefined.

- \( \text{RS.t} = (\bigcup_{n=0}^{\infty} (\langle r, T \rangle \in \text{RS.t}' \mid t' \in \text{IS}^{n}.t \land \\
\neg(\exists m \in \mathbb{N} : m \leq n : (\exists t'' \in \text{IS}^{m}.t : r \in \text{dom}(\text{RS.t}'') \land t'' \neq t'))) \)

  So, \( \text{RS.t} \) associates with all direct and inherited references of \( t \) a simple type. Moreover, overriding is supported as for each reference, the definition which is the "closest" to \( t \) is chosen. In case of name clashes (\( t \) inherits a reference \( r \) from two (different) types at the same "distance" of \( t \), \( \text{RS.t}.r \) is undefined.

In the context of Example 2.3, it holds that \( \text{RS.student} = \langle \langle \text{owns}, \text{car} \rangle \rangle \).

**Definition 2.3.5 (simple object universe)** Let \( \Sigma = \langle \text{ST, AT, RF, IS} \rangle \) be a simple schema according to Definition 2.3.1. Then, the **simple object universe** \( S_\Sigma \) is a set of simple objects according to \( \Sigma \). Such a simple object \( s \) is represented by a tuple \( \langle \text{id}, st, rf, uv \rangle \), where
2.3 Formal Definition

- $id \in ID$, the identity of $s$;
- $st \in ST$, the simple type to which $s$ belongs;
- $av \in (AN \rightarrow DD)$, a function assigning to each applicable attribute name an attribute value;
- $rf \in (RN \rightarrow P(ID))$, a function assigning to each applicable reference $r$ a set of identities of simple objects which are related to $s$ in the context of $r$.

The simple objects in $S_D$ must satisfy two additional requirements. The first one states that each simple object should have attribute values according to $AT_D$:

$$(\forall s \in S_D : (\forall a \in \text{dom}(av.s) : a \in \text{dom}(\overline{AS}.(st.s)) \land a.v.s.a \in \overline{AS}.(st.s).a)).$$

The second requirement expresses that all simple objects must have references according to $RF_D$:

$$(\forall s \in S_D : (\forall r \in \text{dom}(rf.s) : r \in \text{dom}(\overline{RS}.(st.s)))).$$  

In a simple schema, we define for each simple type the possible reference names and for each combination of a type and a reference, the type of objects that may be referenced. The latter requirement cannot be checked at the stage of a simple object universe, as we may only impose requirements that can be verified by "looking" at one simple object at a time. Therefore, this constraint will be formulated in Definition 2.3.9, where the state space concept is defined.

The following definition extends the definition of a simple schema to that of an object schema. For this purpose, the information represented by a simple schema has to be extended by a set of container types, a function representing the structure of each container type, a set of location names and a set of time-stamps.

Definition 2.3.6 (Object schema) Let $SS$ be a simple schema according to Definition 2.3.1. Then, an object schema is a tuple $(SS, CT, CS, LN, TP)$, where

- $CT$ is the set of container types;
- $CS$ defines the container type structure: $CS \in CT \rightarrow P(ST)$ assigns to each container type a set of contained simple types, such that
  $$(\forall c \in CT : (\forall o_0, o_1 \in CS.c : (\exists n \in \mathbb{N} : (\exists s_0, \ldots, s_n \in CS.c : s_0 = o_0 \land s_n = o_1 \land$$
  $$(\forall 0 \leq i < n : s_{i+1} \in \text{rng}(RS.s_i) \cup s_i \in \text{rng}(RS.s_{i+1}))))).$$

This means that for simple types $o_0$ and $o_1$ that are part of a container type $c$, there must be a path of references within $c$ from $o_0$ to $o_1$ when the directions of the references are not taken into account. Furthermore, it must hold that $(\bigcup_{c \in CT} CS.c) = ST$ (no "dangling" simple types).

- $LN$ is the set of location names where (container) objects may reside;
- $TP$ is the set of time points: a totally ordered set with an addition operator +.
Example 2.3.7 (object schema formalisation) The object schema from Example 2.2.2 is represented by the tuple \((S_S, CT, CS, LN, TP)\), where\(^7\)
- \(SS\) is the simple schema from Example 2.3.2.
- \(CT = \{\text{class, trip}\}\).
- \(CS\) contains exactly the following elements:
  \[
  \begin{align*}
  \text{CS.class} &= \{\text{student}\} \\
  \text{CS.trip} &= \{\text{person, car}\}
  \end{align*}
  \]
- \(LN = \{\text{garage, room, road}\}\).
- \(TP = \mathbb{R}\) with the addition operator \(+\).
\[
\]
□

Definition 2.3.8 (container object universe) Let \(\Sigma\) be an object schema. Let \(SD\) be the simple object universe corresponding to \(\Sigma\). The container object universe \(CD\) is a set of container objects according to \(\Sigma\). Such a container object \(c\) is represented by a tuple \((id, ct, cs, lc, at)\), where
- \(id \in ID\): the object identity of \(c\);
- \(ct \in CT\): the container type to which \(c\) belongs;
- \(cs \in \mathcal{P}(SD)\): the set of contained simple objects of \(c\);
- \(lc \in LN\): the location of \(c\);
- \(at \in TP\): the time-stamp of \(c\).

All container objects in \(CD\) must satisfy the following condition:
\[
(\forall c \in CD : (\forall x \in cs.c : st.x \in CS.(ct.c)))).
\]

This means that the types of all contained simple objects in a container \(c\) have to be an element of the contained simple types of the container type of \(c\). □

Definition 2.3.9 (state space) Let \(\Sigma\) be an object schema, \(SD\) the corresponding simple object universe and \(CD\) the corresponding container object universe. We then define the state space \(SSD\) to be a set, which consists of sets of container objects which satisfy the following additional constraints. All containers and simple objects must have a unique identity. Furthermore, the simple objects within these containers have to satisfy the reference structure of \(\Sigma\)\(^8\):
\[
SSD = \{s \in \mathcal{P}(CD) | (\forall c_1, c_2 \in s : c_1 \neq c_2 \Rightarrow id.c_1 \neq id.c_2 \land \\
(\forall s_1 \in cs.c_1, s_2 \in cs.c_2 : id.s_1 = id.s_2 \Rightarrow (c_1 = c_2 \land s_1 = s_2)) \land \\
(\forall s \in cs.c_1 : (\forall r \in \text{dom}(rf.s) : (\forall x \in cs.c_2 : \\
id.x \in rf.s.r \Rightarrow st.x \leq Rs.(st.s).r))))
\]

\(^7\)Example 2.2.2 does not specify locations and time-stamps for the container objects. We complete the schema in this example in a logical way.

\(^8\)The symbol \(\in\) is introduced in Chapter 4. For the moment, it can be read as a simple \(\in\).
2.4 Tool Support

In this section, we describe the features of the SimCon tool that support the use of the SimCon Object Model. Before dealing with the functionality of this tool, we first make some general remarks.

Although tool facilities for the SimCon Object Model are important, we cannot deal with a SCOM tool in isolation. The main purpose of developing SCOM was the integration with a process model, in particular a high-level Petri net model. We could have developed a CASE tool from scratch. However, we had an existing Petri net tool at our disposal, viz. the ExSpect tool. This tool has been under development for several years now, and we have used it successfully in a number of practical situations. Examples of such practical applications are cases in the context of the TASTE project [Van der Aalst, Voorhoeve & Waltmans, 1989], a study at Schiphol Airport and a project with Dutch Railways. The ExSpect tool supports the definition and analysis of a kind of Coloured Timed Hierarchical Petri Nets. More details can be found in Chapter 4 or in the ExSpect literature. It will not be surprising that we have used ExSpect as basis for our CASE tool. In this section, we describe part of the layer that we have built “on top” of ExSpect in order to provide tool support for SCOM. In the next two chapters, other relevant aspects of the integrated tool are presented.

In ExSpect, the state space of a net and the functionality of the transitions are described by means of a typed functional language. As has been recognised in [Verkoulen, 1990; Van Hae & Verkoulen, 1991; Houben & Verkoulen, 1991a], this is not satisfactory for modelling data-intensive applications. This provides motivation for developing a new, integrated model. However, we use this functional language for implementing some of the facilities that are necessary for developing and using SCOM schemas.

There were two fundamental possibilities for extending the ExSpect tool in order to support SCOM. First, we could have implemented these facilities directly, parallel to the ExSpect type system. However, this choice has several disadvantages. We want to mention the major ones. The most important disadvantage is that we would be obliged to model all static information in an application by means of SCOM. Although this seems natural and recommendable, there are situations where this gives unnatural and contaminated models, as this means that the schema has to be extended with simple and container types representing control flow. E.g., a trigger without a value, for synchronising two machines would result in a simple type and a singleton container type in the schema. This is undesirable: for this purpose, we still want to be able to use the ExSpect type system. Other disadvantages are the disturbance of the upward compatibility of ExSpect and the fact that two separated (but similar) tools would be formed. Of course, the implementation “on top of” ExSpect takes less time than an implementation from scratch.

It would be wiser to reuse the ExSpect tool as much as possible. This is exactly
what we have done. We have developed some graphical facilities for developing schemas. For representing instances and operations upon them, we have used the ExSpect type system [Hermans, 1992]. So, we have implemented SCOM "on top of" ExSpect.

For applying the SimCon Object Model, the most important facility of the tool is the schema editor. This editor can be used to design schemas as described in the previous sections of this chapter. Besides a representation of schemas, we also need a representation of instances. This representation has to support the manipulations as described in the next chapter.

The graphical editor that has been developed for designing SCOM schemas provides facilities for defining, modifying and deleting simple and container types. Moreover, attributes and references can be added graphically. The same holds, to some extent, for formulating constraints: some important constraints can be formulated graphically. Others can be entered textually. Container types can be defined by clicking on simple types in the simple schema. For each container type, a separate window can be opened that contains the simple types that form the container type. It is also possible to view the "contents" of a container type (the simple types constituting a container type) directly in the simple schema, by emphasizing the constituting simple types. In this way, the relationships of the container type with other simple and container types can be viewed. The editor checks whether the container types are (indirectly) connected, as demanded by Definition 2.3.6. Moreover, the tool computes all knowledge and overlap relationships between container types. Two container types overlap if they contain at least one common simple type. There is a knowledge relationship between container types T and T' if a simple type in T refers to a simple type contained by T'.

When developing a schema, it is useful to be able to state that there is a relationship between two object types, without stating yet the name of the relationship or the direction of the reference that represents the relationship. Therefore, the tool allows for undirected, unlabelled references, that can be changed later into directed labelled ones.

The symbols that are used for representing simple types, attributes and references are customisable.
2.5 Concluding Remarks

In this chapter, we have introduced the SimCon Object Model. We have provided both an informal introduction and a formal definition. Also, we have described the most important features of the tool that supports the SimCon Object Model. However, as it was our goal to develop a framework for describing static and dynamic system aspects, we are not ready yet. One of the things that must be developed is a language to manipulate objects. This language is introduced in the next chapter.

In the field of database research, some people have argued that such a manipulation language is part of the data model in the context of which it will be used. Other researchers take the view that model and manipulation language should be clearly distinguished. Without opting for either one of these views, we present the object model and the associated manipulation language in separate chapters for convenience.

Some approaches that are more or less similar to our approach do not provide such a manipulation language. Examples are [Van Hee, 1993; Sibertin-Blanc, 1985; Kappel & Schrefl, 1991]. In these approaches, manipulations have to be expressed at a lower level of abstraction, viz. at a level giving the formal semantics of the object model. The approach in [Van Hee, 1993] has been derived from [Van Hee & Verkouwen, 1991, 1992]. However, Van Hee [1993] does not provide a manipulation language that is at the same level of abstraction as the object model. Instead of this, he translates a schema from the object model into set of type definitions. Manipulations have to be expressed in terms of a sugared typed λ-calculus, using these type definitions. Kappel & Schrefl [1991] and [Sibertin-Blanc, 1985] offer a similar kind of object manipulation as Van Hee [1993]. In our opinion, it is preferable to have a manipulation language that is at the same level of abstraction as the object model.
Chapter 3

The SimCon Algebra

3.1 Introduction

In the previous chapter, we have defined the SimCon Object Model. SCOM can be used for describing the static aspects of a system. This is done by developing schemas, defining which kinds of objects may occur in a system. However, we also need a mechanism that can be used to manipulate these objects. We prefer a manipulation mechanism that is on the same level of abstraction as the object model. This is motivated in the next paragraph.

In Chapter 4, we introduce the SimCon Net Model, which is obtained by integrating the SimCon Object Model within the Petri net approach of ExSpect. In this way, data and process modelling have been integrated by defining passive objects as SimCon objects and active objects as transitions in the SimCon net. In the integrated model, the tokens in the net are represented by SimCon container objects. To each place, a container type is assigned, denoting which kind of containers may occur in that place. This integration is more effective when the transitions in the Petri net can be defined directly in terms of this object model. We could have given a mathematical formalisation of the object model (e.g. in terms of λ-calculus) and let the modeller use this formalisation in order to describe the state space and the transitions of his net model. The latter has been done by Van Hee [1993] among others. [Verkoulen, 1990] also describes such an approach. In both cases, an object model is used to represent type definitions in a graphical way. However, data manipulation still has to be done by defining functions in a (sugared) λ-calculus. For modelling dynamic systems where static properties are also important and complex, a closer integration of formalisms for data and process modelling is desirable.

Figure 3.1 illustrates the difference between our integration at the conceptual level and the approaches that integrate at a lower level. Many approaches integrate static and dynamic modelling according to the three continuous arcs in Figure 3.1. They formalise a data model and a process model and combine them by using the
formalisation of the data model to describe the state space of a process specification, or by using the process model to add dynamics to a schema from the data model. In the conclusion of the previous chapter, we have given some examples of such approaches. However, we have integrated our models both at this "fundamental" level and at the conceptual level that is depicted by the dashed arrow. Already at a high level of abstraction, data and process modelling are combined. Moreover, in our integrated approach neither the data model is more important than the process model, nor the other way around.

![Diagram of integration]

Figure 3.1: Levels of integration

For making this integration at a higher level of abstraction possible, we introduce the SimCon Algebra (SCA) [Houben & Verkoulen, 1991ab, 1992ab]. This algebra is defined for manipulating SimCon objects. In the context of the integrated SimCon net model, it is meant for describing the functionality of the Petri net transitions in terms of the SimCon Object Model; the latter will be used for defining the token structure of the net.

We have developed the SimCon Algebra according to a number of guidelines, which are dealt with in this paragraph. The operations of SCA should be sufficiently powerful. This makes SCA interesting in isolation. In this chapter, we deal with some interesting properties of SCA, like its expressive power in relation with other object manipulation mechanisms. We have kept the concepts of the SimCon Algebra as simple as possible by defining a kernel\(^1\) of primitive operations. This kernel has been kept small, which makes it more easy to obtain theoretical results for the algebra. Yet, it provides sufficient expressive power (cf. Section 3.5). In

\(^1\) Such a kernel is often called the generating part of the language.
order to provide satisfactory expressive comfort to the user of the algebra, we have
defined a number of high-level operations in terms of this kernel. In this way,
it is not necessary to include these operations in the investigation of theoretical
properties of SCA, yet giving operations that are practically applicable. Of course,
the set of high-level operations can always be extended. E.g., we may develop
libraries with special-purpose operations for special classes of applications, as it is
possible to define operations in terms of other ones, using a kind of non-recursive
\( \lambda \)-abstraction. The expressive comfort of the algebra can also be improved by pro-
viding a graphical syntax for SCA. This is discussed in Section 3.3. A final reason
for making SCA sufficiently powerful is motivated by its role within the integrated
model. As can be seen in Chapter 4, Petri net transitions can be used to model
active objects, like machines in a factory. The functionality of the transitions will
be described by an SCA expression. We do not want to be forced to contaminate
the net structure because some manipulation cannot be expressed by one SCA
expression in one transition, even though the concept of hierarchical nets gives
possibilities for hiding this at the highest level of abstraction.

Reconsidering the SimCon Object Model, we observe that both schemas and
instances have been defined in terms of graphs. Not only a SimCon schema can
be thought of as a graph, but also an instance can be represented as a graph. As
we will see, a state of a SimCon net is a set of container objects. This means
that such a state can be represented as a directed, coloured, partitioned graph.
The nodes of this graph are the existing simple objects. As these objects may
have attributes, we say the nodes are coloured. Moreover, simple objects may have
(directed) references to one another, which are the edges of the graph. Finally,
each simple object is contained by exactly one container object, so the graph is
partitioned by the present containers.

With this observation in mind, it is obvious that the primitive SCA operations
are graph-based. The SCA kernel consists of nine operations. Applying these
operations, simple objects can be created and deleted, references between simple
objects can be created and deleted, attributes values can be assigned to and re-
moved from simple objects, container objects can be created and simple objects
can be moved between containers. Besides these graph-manipulation operations,
there is one primitive operation which can be used to apply a sequence of SCA
operations recursively.

In most operations, the set of simple objects to which the operation is applied,
must be denoted. For this purpose, we introduce the concept of simple object selec-
tors (sos). This mechanism can be used to select simple objects from an instance,
based on their type, their attribute values, their incoming and outgoing references
and/or their location. A similar mechanism is that of the value selectors (vs).
Given a simple object, this mechanism can be used to access all attribute values
of a certain type that are reachable via certain references from that object. E.g.,
this mechanism can be used for computing the names of all direct subordinates of
a manager.
As has been mentioned in Section 2.4, we have decided to implement SimCon and its associated manipulation language SCA on top of the ExSpect tool. This means that a SimCon schema is described in terms of ExSpect types. The SCA operations are realised by ExSpect functions. Details about this way of building SimCon/SCA upon the ExSpect functional language can be found in [Hermans, 1992]. In a future research effort, the first approximation by Hermans [1992] should be improved.

The next section gives a formal definition of the SimCon Algebra. We introduce a syntax and a semantics for SCA. These formal definitions are illustrated by means of examples. The formal definitions may look complex, because they cover some pages. However, the SCA kernel is actually quite simple, as it consists of some basic and intuitive graph manipulations. Section 3.4 introduces some high-level operations. These operations can be expressed by means of primitive SCA operations. However, they are useful because they add expressive comfort to the algebra. Afterwards, we deal with the expressive power2 of SCA. We show that our mechanism is at least as powerful as two well-known data models, viz. GOOD and the Nested Relational Algebra. We also show that SCA is more expressive than GOOD. This difference in expressive power is caused by a non-deterministic property of one of the SCA operations. According to some people, this non-determinism is undesirable. Therefore, in Section 3.4, we provide two operations that can be used to replace this non-deterministic operation without losing more expressive power than necessary. We close this chapter by making some conclusive remarks, evaluating what has been accomplished in this chapter.

3.2 Definition of the SimCon Algebra

An SCA operation is applied to a so-called SimCon-instance. As can be seen in Definition 3.2.1, such an instance consists of a set of container objects, a set of identities and a set of container labels. These identities are used when new objects are created. The container labels are used to refer to (newly created) container objects in an SCA expression. The container identity cannot be used for this purpose, as we want to hide the identities of objects. Moreover, we want our operations to be generic: when an SCA expression $\mathcal{E}$ is applied upon any two instances $I$ and $I'$ which are equal modulo identities, the results $\mathcal{E}(I)$ and $\mathcal{E}(I')$ should also be equal modulo identities. In order to have generic operations, container labels can be used as a kind of "intermediate references". In the following chapter, this will become more clear.

---

2The operations from Section 3.4 can be used when describing the expressive power of the algebra. That is why the section about expressive comfort precedes the one dealing with expressive power.
Definition 3.2.1 (SimCon-instance) A SimCon-instance (in the context of a schema \( \Sigma \)) is a triple \((C, AI, CL)\), where:
- \( C \) is a set of pairs \((\ell, c)\); here, \( \ell \) is a container label and \( c \in C_\ell \) is a container object in the container object universe of \( \Sigma \);
- \( AI \) is the set of available identities (a subset from the identification set in the context of \( \Sigma \));
- \( CL \) is a set of container labels.

The set of all SimCon-instances in the context of a schema \( \Sigma \) will be denoted by \( I_\Sigma \).

For a SimCon-instance \( J = (C, AI, CL) \), we will denote the number of simple objects in \( C \) by \( \#J \). We call this number the cardinality of \( J \).

For a SimCon-instance \( J = (C, AI, CL) \), we will denote the set of simple objects in \( C \) by \( \mathcal{N}(J) \). The set of container objects in \( C \) (without labels) will sometimes be denoted by \( C(J) \).

When the identities and the container labels are not relevant, we will not mention them explicitly. So, we will sometimes refer to a set of container objects as a SimCon-instance.

Roughly speaking, two instances are equal if only the identities of the objects and the labels of the containers in those instances differ. The sets of available identities and available labels are not taken into account. This is formalised by the following definition.

Definition 3.2.2 (equal instances) Two SimCon-instances \( J \) and \( J' \) are equal if and only if:
1. The number of simple objects in \( J \) equals the number of simple objects in \( J' \).
   This also must hold for the container objects.
2. For each simple object in \( J \), there is a simple object in \( J' \) of the same type, with the same attributes and references to and from objects that may only differ regarding their identities, and vice versa.
3. For each container in \( J \), there is a container in \( J' \) with equal contents (modulo identities) as that in \( J \), and vice versa.

3.2.1 Selectors

In the introduction to this chapter, we have encountered simple object selectors and value selectors. In this section, these notions are formalised. We start by giving the sos- and vs-syntax. Afterwards, we define the semantics of object and value selectors. An example is used to clarify the definitions.
Definition 3.2.3 (selector syntax) A simple object selector sos obeys the syntax rules as described below. Here, sos stands for a simple object selector, v for a value, vs for a value selector and TD for a type descriptor (a simple type name and (optionally) a container label).

\[
\begin{align*}
sos & := TD[condition^+] \\
condition & := v \equiv v \mid \leftarrow sos \mid \rightarrow sos \\
TD & := \langle simple-type-name \rangle \mid \langle simple-type-name \rangle, \langle container-label \rangle \\
v & := vs \mid \langle attribute-name \rangle \mid \langle constant \rangle \\
vs & := TD[\langle attribute-name \rangle] \mid TD[\leftarrow vs] \mid TD[\rightarrow vs, condition^+] \\
r & := \langle reference-name \rangle
\end{align*}
\]

The aforementioned syntax is a textual one. Section 3.3 suggests a graphical syntax for selectors and then also for the complete SimCon Algebra. In the remainder of this monograph, we mainly use the textual version of SCA. However, a graphical version of SCA may improve the expressive comfort of SCA, especially for laymen.

Before giving the semantics of object and value selectors, we illustrate their syntax and semantics by giving a schema and some selectors based on this schema.

Example 3.2.4 (selectors) Consider the schema from Figure 3.2 (container types have been omitted).

![Figure 3.2: An example schema](image)

The following simple object selector denotes all simple objects of type A which have a w-attribute which is equal to 6:

\[\text{Words between (...) denote terminals. The vertical bar (|) denotes a choice between alternatives. An expression } e^0 \text{ means zero or more occurrences of that expression } e, \text{ separated by commas. An expression } e^+ \text{ means one or more occurrences of that expression } e, \text{ separated by commas. All other symbols are to be taken literally.}\]
\[ \lambda[w = 6] \]

The following simple object selector denotes all simple objects of type \( A \) within a container with label \( \ell \) which have a \( w \)-attribute which is equal to 6:
\[ \lambda, \ell[w = 6] \]

The following simple object selector denotes all simple objects of type \( A \) which have a \( w \)-attribute which is equal to 3 and a \( p \)-reference to a simple object of type \( B \) which has an \( r \)-reference to a simple object of type \( C \) which has a \( y \)-attribute which is equal to 6:
\[ \lambda[w = 3, \stackrel{p}{\rightarrow} B \stackrel{r}{\rightarrow} C[y = 6]] \]

The following simple object selector denotes all simple objects of type \( A \) which have a \( p \)-reference to a simple object of type \( B \) which has an \( r \)-reference to a simple object of type \( C \) which has a \( y \)-attribute which is equal to the \( w \)-attribute of that \( A \)-object:
\[ \lambda[\stackrel{p}{\rightarrow} B \stackrel{r}{\rightarrow} C[y]] = w \]

The following simple object selector denotes all simple objects of type \( A \) which have a \( p \)-reference to a simple object of type \( B \) which has an \( r \)-reference to a simple object of type \( C \) which has a \( y \)-attribute which is equal to the \( x \)-attribute of some \( B \)-object to which that \( A \)-object has a \( p \)-reference:
\[ \lambda[\stackrel{p}{\rightarrow} B \stackrel{r}{\rightarrow} C[y]] = \stackrel{p}{\rightarrow} B[x] \]

The following simple object selector denotes all simple objects of type \( B \) to which some simple object of type \( A \) with a \( w \)-attribute which is equal to 6 has a \( p \)-reference:
\[ b[\stackrel{p}{\Lambda}[w = 6]] \]

\[ \Box \]

In the definitions below, we give the semantics of simple object selectors and value selectors in the context of an object schema \( \Sigma \) and an instance \( J \) according to \( \Sigma \). Note that we define three sorts, viz. \( O \) (simple objects), \( V \) (values) and \( I \) (identities of simple objects):

(i) \( o[\tau[C]] \); gives all simple objects in \( J \) of type \( \tau \) which satisfy \( C \).

(ii) \( v[\tau[v, C]] \); gives a set of pairs. The first component of such a pair is a simple object \( x \) in \( J \) of type \( \tau \) satisfying \( C \) and the second one is the set of attribute values of attributes that can be reached by following the references contained by \( v \), starting from \( x \). The constraint \( C \) is optional.

(iii) \( t[\tau[C]] \); gives a set of simple object identities, viz. the identities of all simple objects of type \( \tau \) in \( J \) satisfying \( C \) and the identities of all objects of type \( \tau \) which satisfy \( C \) and to which an object in \( J \) refers.

In the definitions below, we use an auxiliary function \( id \) that can be applied to a set of simple objects and returns the set of all identities of these objects.
Definition 3.2.5 (O-semantics). Let $\Sigma$ be an object schema and consider a SimCon-instance $J = \langle C, A1, A2 \rangle$ according to $\Sigma$. The O-semantics of object and value selectors is given by the following equations.

$$o[I[I]]_J = \{x \in (\bigcup_{(\ell, y) \in cs.y}) | st.x \leq T\}$$

This denotes all simple objects in any container of $C$ which are of (a subtype of) type $T$.

$$o[I[a = v]]_J = \{x \in (\bigcup_{(\ell, y) \in cs.y}) | st.x \leq T \land a \in \text{dom}(av.x) \land (av.x).a = v\}$$

This denotes all simple objects in any container of $C$ which are of (a subtype of) type $T$ which have an attribute $a$ which is equal to $v$.

$$o[I[v = a]]_J = o[I[a = v]]_J$$

$$o[I[\exists sos]]_J = \{x \in (\bigcup_{(\ell, y) \in cs.y}) | st.x \leq T \land \tau \in \text{dom}(rf.x) \land (\exists \tau \in o[sos]_J : \text{id.}\tau \in (rf.x).\tau)\}$$

This denotes all simple objects in any container of $C$ which are of (a subtype of) type $T$ which have an $\tau$-reference to some simple object selected by sos.

$$o[I[X_1, X_2]]_J = o[I[X_1]]_J \cap o[I[X_2]]_J$$

$$o[I[\ell[X]]_J = o[I[X]]_J \cap cs.y, \quad \text{if} \quad (\ell, y) \in C$$

$$\emptyset, \quad \text{otherwise}$$

This denotes all simple objects in the container in $C$ with label $\ell$, if it exists, which are of (a subtype of) type $T$ and satisfy $X$.

$$o[I[v_{S1} = v_{S2}]]_J = \{x \in (\bigcup_{(\ell, y) \in cs.y}) | st.x \leq T \land (\exists y : (x, y) \in v[I[v_{S1}]]_J \land (\exists z : (x, z) \in v[I[v_{S2}]]_J \land y \cap z \neq \emptyset))\}$$

We see here that first the V-semantics of $v_{S1}$ and $v_{S2}$ are determined\(^5\). Then all the

---

\(^4\)The predicate $\exists$ is introduced in Section 4.3. For the moment, the reader may ignore the dots above mathematical symbols like $\exists$ and $\cup$.

\(^5\)One of the value selectors may be a constant. The V-semantics of a constant simply returns this constant.
3.2 Definition of the SimCon Algebra

simple objects \( x \) in the intersection of the first components of both semantics are taken which have at least one common attribute value in their second component.

The last clause of the previous definition states that all objects are selected for which evaluation of vs1 and vs2 gives at least one common value. This is analogous to the situation with simple object selectors, where all objects are selected that have an \( r \)-reference to at least one simple object satisfying some condition. Within the integrated framework, this is the most natural choice, because of the locality principle of Petri nets.

**Definition 3.2.6 (V-semantics)** Let \( \Sigma \) be an object schema and consider a SimCon-instance \( f = (\mathcal{C}, \mathcal{A}_1, \mathcal{C}_1) \) according to \( \Sigma \). The V-semantics of value selectors is given by the following equations.

\[
\nu[\tau[\alpha]]_1 = \{ (x, (\nu[\alpha \vee \alpha], \alpha)) \mid x \in \nu[\tau[\alpha]]_1 \land \alpha \in \text{dom}(\alpha[\nu]) \}
\]

This gives all objects of (a subtype of) type \( \tau[\alpha] \) for which the attribute \( \alpha \) is defined, together with the singleton containing their value on \( \alpha \).

\[
\nu[\tau[\alpha]]_1 = \{(x, y) \mid \text{st.} x \leq \tau \land y = \bigcup \{ \beta \mid (\alpha, \beta) \in \nu[\tau[\alpha]]_1 \land \text{id.} \alpha \in (\text{rf.} x, \tau) \}
\]

This gives all objects of (a subtype of) type \( \tau \) together with the generalised union of all second components in the V-semantics of \( \tau[\alpha] \) for which \( s \) has an \( r \)-reference to the corresponding first component.

\[
\nu[\tau[\alpha]]_1 = \{(x, y) \mid \text{st.} x \leq \tau \land y = \bigcup \{ \beta \mid (\alpha, \beta) \in \nu[\tau[\alpha]]_1 \land \text{id.} x \in (\text{rf.} x, \tau) \}
\]

This gives all objects of (a subtype of) type \( \tau \) together with the generalised union of all second components in the V-semantics of \( \tau[\alpha] \) for which there is an \( r \)-reference from the corresponding first component to \( s \).

\[
\nu[\tau[\alpha]]_1 = \{(x, y) \in \nu[\tau[\alpha]]_1 \mid x \in \nu[\tau[\alpha]]_1 \}
\]

This gives all simple objects satisfying \( \mathcal{C} \) together with their sets of attribute values according to \( \nu \).

The informal meaning of the clauses defining the semantics of a value selector is illustrated by the following example.

**Example 3.2.7 (semantics value selector)** Consider Figure 3.3. Suppose we want to determine which objects are selected by

\[
\tau[\tau[\tau[\alpha, \tau[\tau[\alpha, \tau[\tau[\alpha]\alpha]]]]]]] = \{ x_1, x_2, \ldots, x_n \} [\tau[\tau[\tau[\alpha, \tau[\tau[\alpha, \tau[\tau[\alpha]\alpha]]]]]]]
\]
Figure 3.3: Two attribute paths
3.2 Definition of the SimCon Algebra

To this end, we first determine which objects are selected by the left-hand side. This gives all objects of (a subtype of) type $T$ together with all the $a$-attribute values they can "reach" via $r_1, r_2$ and $r_3$; in case of this example ($x, \{2,3\}$) and ($x', \{4\}$). We do the same for the second value selector, yielding ($x, \{2\}$) and ($x', \{5,6\}$). In the final result, only $x$ occurs, because in the evaluation of both value selectors $x$ occurs as a first component and the corresponding second components have a non-empty intersection. It is obvious that the object $x'$ is not selected. □

Definition 3.2.8 (t-semantics) Let $\Sigma$ be an object schema and consider a SimCon-instance $I = (C, A, CL)$ according to $\Sigma$. The identity-semantics of object selectors is given by the following equations.

\[
i[T][a = v]] = i[T][a = v]]_1
\]

\[
i[T][v = a]] = i[T][v = a]]_1
\]

\[
i[T][\text{dom}]] = \id(C_{[T][\text{dom}]}_1) \cup \\ \{ t \mid (3x \in (\bigcup_{(t,u) \in C_{T,x} : T = T \land i \in (r(T),T)}) \land \begin{cases} x \leq T \land r \in \text{dom}(r(T)) \land (3i' \in I_{[T][\text{dom}]}_1 : t' \in (r(T),T)) \end{cases} \}
\]

This denotes all identities of all simple objects in any container of $C$ which are of (a subtype of) type $T$ for which the reference $r$ is defined and which have an $r$-reference to some simple object selected by $sos$.

\[
i[T][\text{ran}]] = \id(C_{[T][\text{ran}]}_1) \cup \\ \{ t \in ID : (3x \in C_{[T][\text{ran}]}_1 : (3r \in RN : (x_{[T]ran},r = T \land i \in (r(T),T))) \}
\]

This denotes the identities of all simple objects in any container of $C$ which are of (a subtype of) type $T$ from which some simple object in $sos$ has an $r$-reference.

\[
i[T][X_1, X_2]] = i[T][X_1]]_1 \cap i[T][X_2]]_1
\]

\[
i[T][t(X)] = i[T][t(X)]_1
\]

\[
i[T][v_{s1} = v_{s2}]] = i[T][v_{s1} = v_{s2}]]_1
\]
3.2.2 Definition of the SCA Kernel

This section presents the kernel of the SimCon Algebra. The operations in the kernel give SCA its expressive power. For practical use, these operations will not always be satisfactory. That is why we present some operations that are more advanced in Section 3.4. These high-level operations can be expressed in terms of the SCA kernel, which is defined in this section.

We illustrate each definition by means of a simple example. All example expressions will work upon the instance \( I \) from Figure 3.4. This instance consists of three container objects\(^6\) with labels \( l_1, l_2 \) and \( l_3 \). The underlying schema is that of Figure 3.2, extended with one container type \( y \), consisting of the simple types \( A, B \) and \( C \).

![Figure 3.4: An example instance](image)

In the following definitions, we assume that some object schema \( \Sigma \) is given. We use the abbreviation \( I \) for some SimCon-instance \( (C, A_1, C_1) \in I_\Sigma \). Furthermore, \( \sigma \) denotes the semantics of some operation \( o \). Finally, \( sos, sos_1 \) and \( sos_2 \) denote a simple object selector, \( r \) a reference name, \( a \) an attribute name, \( \ell \) a container label, \( st \) a simple type, \( ct \) a container type and \( (id, ct, cs, lc, ct) \) a container object.

Before giving the formal definition of the SCA kernel, one remark has to be made. Each SCA expression is applied in the context of a SimCon schema. This means that the expressions should "fit" into this schema, i.e. proper attribute and reference names, simple types and container types should be used. However, it is also allowed to use auxiliary ("fresh") attributes, references and types. We will encounter several examples of such SCA expressions. This auxiliary information is

---

\(^6\)Please remind that it is not necessary that all attributes and/or references of an object are present. As it may occur in reality that we do not know all properties of some entity, we want to be able to model this in our framework.
not persistent, i.e. "outside" the expression, this information does not exist.

The first operation we define is the Simple object Creation (SC). There are three versions of the SC-operation. They create new simple objects, one for each simple object satisfying the sos-parameter. However, the second version also creates one reference from each newly created object to exactly one unique object selected by sos. Finally, it is also possible to create exactly one simple object of a certain type, without stating additional conditions upon the context in which the operation is executed.

Of course, it is also possible to create a new object for each occurrence of a pair of objects. The definition of simple object selectors does not make this possible by only one SC-expression. In Example 3.4.1, an expression is given that creates a simple object of type marriage for each couple of persons of which one is the husband of the other.

The identities of the newly created simple objects are taken from the set of available identities $AI$ of $J$. It is not relevant which identities are chosen; it only has to be guaranteed that they are unique. In this chapter, we simple postulate that this is the case. In the next chapter, where SCA will be used for defining Petri net transitions, we prove that our way of working satisfies this postulate.

**Definition 3.2.9 (Simple object Creation)**  
$SC(J, st, sos, \ell)$ adds for each simple object selected by sos a new simple object of type $st$ to the container with label $\ell$. For $J = (C, AI \cup X, CL)$ with $|o[sos]| = |X| = n$, it holds that:

$$SC(J, st, sos, \ell) =$$

$$(C \setminus \{(\ell, (id, ct, cs, lc, at)\}))$$

$$\cup \{(\ell, (id, ct, cs \cup \{(i, st, \emptyset, \emptyset) \mid i \in X \}, lc, at)\)},$$

$AI,$

$CL$.

If $C$ does not contain a container with label $\ell$, applying this operation gives $J$ as result.

$SC(J, st, sos, \ell, r)$ adds for each simple object selected by sos a new simple object of type $st$ to the container with label $\ell$. Furthermore, each of the newly created simple objects is connected to exactly one simple object selected by sos via an $r$-reference (and no two newly created simple objects are connected to the same simple object). For $J = (C, AI \cup X, CL)$ with $|o[sos]| = |X|$ and a bijective function $i$ from the identities of the objects selected by sos to $X$, it holds that:

$$SC(J, st, sos, \ell, r) =$$

$$(C \setminus \{(\ell, (id, ct, cs, lc, at)\}))$$

$$\cup \{(\ell, (id, ct, cs \cup \{(i, st, \{r, \{i\}\}, \emptyset) \mid i \in id(o[sos])\}, lc, at)\)},$$

$AI,$

$CL$.

If $C$ does not contain a container with label $\ell$, application of this operation yields $J$. 

SC(\(J, st, true, f\)) adds exactly one new simple object of type \(st\) to the container with label \(f\). For \(J = (C, AI \cup \{1\}, CL)\), it holds that:
\[
SC(J, st, true, f) = 
\{(c - \{(f, \{id, ct, cs, lc, at\})\}) 
\cup \{(f, \{id, ct, cs \cup \{(i, st, \emptyset)\}, lc, at\})\}, 
\{AI, 
\{CL\}\}
\]
If \(C\) does not contain a container with label \(f\), applying this operation gives \(J\) as result.

Example 3.2.10 (Application of SC) Consider the instance \(J\) that has been depicted in Figure 3.4. In Figure 3.5, the instance is given that results from applying \(SC(J, A, 2 \{x = 2\}, f_3, p)\).

\[\text{Figure 3.5: Result of applying SC on the example instance J}\]

The "inverse" operation of \(SC\) is the Simple object Deletion (SD). This operation removes all objects satisfying sos from the instance to which it is applied.

Definition 3.2.11 (Simple object Deletion) SD(J, sos) removes all the simple objects selected by sos from any container in C (recall that it can be specified within the sos-parameter that only objects in a certain container have to be deleted):
\[
SD(J, sos) = 
\{(\{f, \{id, ct, cs \sim o[sos]\}, lc, at\}) \mid \{f, \{id, ct, cs, lc, at\}\} \in C\}, 
\{AI, 
\{CL\}\}
\]
3.2 Definition of the SimCon Algebra

The identities of the deleted objects can not be reused in A1. This is caused by the way of assigning unique identities to newly created objects, which is demonstrated in Section 4.3.

Example 3.2.12 (Application of SD) Consider the instance J from Figure 3.4. Figure 3.6 depicts the instance SD(J,A [\{b\}]).

![Diagram of example](image)

Figure 3.6: Result of applying SD on the example instance J

The following two definitions give operations for adding and removing references.

Definition 3.2.13 (Reference Creation) \(RC(J, sos_1, sos_2, r)\) adds to each simple object selected by sos_1 an r-reference to all simple objects selected by sos_2:

\[
RC(J, sos_1, sos_2, r) = \\
\{ \langle (i, (id, ct, rc(cs, r, _o sos_1), sos_2)), ic, at \rangle \mid \langle (i, (id, ct, cs, ic, at) \rangle \in C, \\
A_1, \\
CL \}
\]

Here, the function rc is defined as follows. If S and s_1 are sets of simple objects, s_2 is a set of identities of simple objects and r is a reference name, then \(rc(S, r, s_1, s_2)\) adds to all objects in \(S \cap s_1\) an r-reference to all simple objects with an identity in s_2. Formally:

\[
rc(S, r, s_1, s_2) = \\
(S \setminus s_1) \cup \\
\{(id, st, (rf - \{(r, X)\}) \cup \{(r, X \cup s_2)\}, av) \mid \\
(id, st, rf, av) \in S \cap s_1 \}
\]

\(\square\)
In this RC-operation, there is no means of relating the two sos-expressions. However, using the FA-operation, such relationships can be expressed (cf. Example 3.4.1, where an expression is given that creates a reference wife from each marriage object \( m \) to each person object \( p \) for which it holds that there is a husband reference from some other person \( p' \) to \( p \) and a husband reference from \( m \) to \( p' \)).

**Example 3.2.14 (Application of RC)** Consider the instance \( I \) from Figure 3.4. Figure 3.7 depicts the instance \( RC(I, A \{ w = 0, b \{ x = 2 \}, p \} \).

![Diagram](image)

**Figure 3.7: Result of applying RC on the example instance \( I \)**

**Definition 3.2.15 (Reference Deletion)** \( RD(I, sos_1, sos_2, r) \) deletes from the simple objects selected by \( sos_1 \) all \( r \)-references to any simple object selected by \( sos_2 \):

\[
RD(I, sos_1, sos_2, r) =
\{(l, (id, ct, rd(cs, r, o[sos_1], [sos_2]), lc, at)) \mid (l, (id, ct, cs, lc, at)) \in C, A_l, C_l\}
\]

The function \( rd \) is defined as follows. If \( S \) and \( s_1 \) are sets of simple objects, \( s_2 \) is a set of identities of simple objects and \( r \) is a reference, then \( rd(S, r, s_1, s_2) \) removes from all objects in \( S \cap s_1 \) the \( r \)-references to all simple objects with an identity in \( s_2 \). Formally:

\[
rd(S, r, s_1, s_2) =
(S \setminus s_1) \cup
\{(id, st, rf - (\{r, X\}) \cup \{\{r, X \in s_2\}), av) \mid (id, st, rf, av) \in S \cap s_1\}
\]

**Example 3.2.16 (Application of RD)** Consider the instance \( I \) from Figure 3.4. Figure 3.8 depicts the instance \( RD(I, A \{ w = 0, b \{ x = 0 \}, p \} \).
3.2 Definition of the SimCon Algebra

Figure 3.8: Result of applying RD on the example instance J

The following two definitions introduce operations for adding attribute values to and removing attribute values from simple objects. Note that attribute modification can be implemented by first deleting the old values and then adding the new ones.

Definition 3.2.17 (Value Addition) \(VA(J, sos, a, v)\) maps for each simple object \(s\) selected by \(sos\) the attribute name \(a\) onto \(v\):

\[
VA(J, sos, a, v) = \\
\{(\langle \ell, (id, ct, va(cs, a, v, o[sos]1), ic, at) \rangle, (\ell, (id, ct, cs, ic, at)) \in C) \} \\
\Lambda 1, \\
CL)
\]

Here, \(v\) can be a constant, an appropriate attribute name or an appropriate value selector. The function \(va(S, a, v, s)\) extends, if necessary, the attribute domain \(av.x\) of each simple object \(x \in s \cap S\) with attribute name \(a\) and maps \(a\) onto the semantics of \(v\).

- If \(v\) is a constant, it holds that

\[
va(S, a, v, s) = (S \setminus s) \cup \\
\{(id, st, rf, av \cup \{(a, v)\}) | (id, st, rf, av) \in s \cap S \land a \notin dom(va) \} \cup \\
\{(id, st, rf, \{a, av.a\}) \cup \{(a, v)\}) | (id, st, rf, av) \in s \cap S \land a \in dom(va)\}
\]

- If \(v\) is a value selector, it holds that

\[
va(S, a, v, s) = \bot \text{ if } (\exists x \in s : (\exists (x, \beta) \in v[v] : |\beta| \neq 1))
\]
Otherwise, it holds that
\[
\text{vd}(S, a, v, s) = (S \setminus s) \cup \\
\{(id, st, rf, av \cup \{(a, v)\}) | x = (id, st, rf, av) \in s \cap S \land a \notin \text{dom}(av) \land \langle x, \{v\} \rangle \in v[v]\} \cup \\
\{(id, st, rf, (av \setminus \{(a, av.a)\}) \cup \{(a, v)\}) | x = (id, st, rf, av) \in s \cap S \land a \in \text{dom}(av) \land \langle x, \{v\} \rangle \in v[v]\}
\]

\* If v is an attribute name and the type of the objects in s is T, the previous clause can be used by substituting the value selector t[v].

\*\* Example 3.2.18 (Application of VA) \* Consider the instance I from Figure 3.4. Figure 3.9 depicts the instance VA(I, λ[ ], w, 1).

\*\* Definition 3.2.19 (Value Deletion) \* VD(I, sos, a) deletes the a-attribute from all simple objects selected by sos:
\[
\text{VD}(I, sos, a) = \\
\{(\ell, (id, ct, vd(cs, a, o[sos][])), lc, at)) | (\ell, (id, ct, cs, lc, at)) \in C), \\
A1, \\
CL
\]

The function vd is defined as follows. If S and s are sets of simple objects and a is an attribute name, then vd(S, a, s) removes the a-attributes from all objects in S \cap s. Formally:
\[
\text{vd}(S, a, s) = (S \setminus s) \cup \\
\{(id, st, rf, av \setminus \{(a, av.a)\}) | (id, st, rf, av) \in S \cap s)\}
\]
Example 3.2.20 (Application of VD) Consider the instance J from Figure 3.4. Figure 3.10 depicts the instance VD(J, 8 [x = 0], x).

Figure 3.10: Result of applying VD on the example instance J

Now that we have defined all basic operations for manipulating the properties of the nodes and the edges in the graph representing a SimCon-instance, we define two operations for changing the partitioning of this graph. The first operation serves for creating new container objects, whereas the second one can be used for moving simple objects to a specific container. As can be seen in the next chapter, we do not need an explicit container deletion operation. However, it is obvious that this operation could have been defined in a straightforward way.

Definition 3.2.21 (Container Creation) \( \text{CC}_{\tau, l}(J, ct, \{\ell_1, \ldots, \ell_n\}) \) adds to \( C \) the set of empty containers of type \( ct \) with labels \( \ell_1, \ldots, \ell_n \). The identifications and the container labels of these new containers are removed from \( AI \) and \( CL \) respectively:

\[
\begin{align*}
\text{CC}_{\tau, l}(J, ct, \{\ell_1, \ldots, \ell_n\}) &= \{c \cup \{\ell_1, (\ell_1, ct, \emptyset, \tau, L)), \ldots, \{\ell_n, (\ell_n, ct, \emptyset, \tau, L))\}, \\
AI &= \{v_1, \ldots, v_n\}, \\
CL &= \{v_1, \ldots, v_n\}
\end{align*}
\]

Here \( \tau \) is a time-stamp, \( L \) is a location, \( |\{\ell_1, \ldots, \ell_n\}| = n = |\{v_1, \ldots, v_n\}|. \)

If the time-stamp and the location in the previous definition are not relevant, we simply omit them. In Chapter 4, we show how these properties can be defined implicitly.
Example 3.2.22 (Application of CC) Consider the instance $J$ from Figure 3.4. Figure 3.11 depicts the instance $CC(J, y, \{\ell_4, \ell_5\})$.

![Diagram](image)

Figure 3.11: Result of applying CC on the example instance $J$

Definition 3.2.23 (Container contents Modification) $CM(J, sos, \ell)$ moves the simple objects selected by sos to the container in $C$ with label $\ell$:

$$CM(J, sos, \ell) = ((\{y, (id, ct, cs - o[sos], lc, at)\} | (y, (id, ct, cs, lc, at)) \in C) - \{(\ell, (id, ct, cs, lc, at))\}) \cup \{(\ell, (id, ct, cs, lc, at))\},$$

$AI$, $CL$.

If $C$ does not contain a container with label $\ell$, this operation yields $J$.

Example 3.2.24 (Application of CM) Consider the instance $J$ from Figure 3.4. Figure 3.12 depicts the instance $CM(J, \lambda[w = 2], \ell_2)$.

We still have to introduce one operator from the kernel of SCA. This operator has been called RA, for Recursive Application. The idea is the following: RA executes an SCA expression for each simple object selected by its sos parameter. It works recursively, meaning that it applies its argument expression for one selected simple object, yielding a new SimCon-instance $J'$, then evaluates the sos again, applies the expression for an arbitrary next element in the context of $J'$. This process is repeated until the SCA expression does not cause changes anymore for any of the selected simple objects.
3.2 Definition of the SimCon Algebra

Figure 3.12: Result of applying CM on the example instance $I$

As can be seen in the next section, RA can be used to simulate a non-recursive repetition, too.

Definition 3.2.25 (Recursive Application) $RA_{\alpha,\lambda}(I, sos, e)$ applies for all simple objects $\alpha$ in sos the expression $e$ recursively ($\alpha$ is a variable to which some element in sos is assigned; $\alpha$ may occur in $e$) in some order; $I'$ holds the "intermediate" SimCon-instance "during" execution of the expression ($I'$ may also occur in $e$). First, we introduce the following abbreviation: we define $\delta(I, sos, e, \alpha, I')$ as the set of simple objects in sos for which execution of $e$ in the context of $I$ yields an instance which differs from $I$. Formally$^7$:

$$\delta(I, sos, e, \alpha, I') = \{ x \in o[sos]_i \mid \llbracket e \rrbracket_{\{x\}}^{\tau} \neq I' \}$$

The semantics of RA is given by the following two clauses:

- $RA_{\alpha,\lambda}(I, sos, e) = I$ if $\delta(I, sos, e, \alpha, I') = \emptyset$;
- $RA_{\alpha,\lambda}(I, sos, e) = RA_{\alpha,\lambda}(\llbracket e \rrbracket_{\{x\}}^{\tau}, sos, e)$, otherwise$^8$, where $x$ is an arbitrary element of $\delta(I, sos, e, \alpha, I')$.

The following example demonstrates how the RA-operation is used. In the remainder of this chapter, a number of other examples will be given that demonstrate how to use RA.

Example 3.2.26 (Recursive Application) Consider a schema with a simple type $A$ with attributes $x$ and $y$, both of type integer and a reference $r$ from $A$ to $A$.

$^7$In this definition, we denote the semantics of an expression $e$ in the context of $I$ by $\llbracket e \rrbracket_I$.

$^8$The notation $\llbracket e \rrbracket_{\{x\}}^{\tau}$ is not completely correct, as $\{x\}$ is not a simple object selector. However, it is obvious that it is possible to "repair" this notational slovenliness. We have not done so in order to prevent the definitions from becoming (even) more complex.
Suppose we want to compute all A-objects that are reachable from some A with \( x = 0 \). We do this by assigning the value 0 to the y-attribute of each reachable A-object. This manipulation is expressed by the following SCA expression:

\[
I' = RA_{\alpha}(I_A \mid \rightarrow_A \{ y = 0 \}), VA(I, \alpha, y, 0)
\]

Note that the modeller who uses this operation has to persuade himself of the proper termination of it. Moreover, he will have to prove that the non-deterministic choice that is inherent to this operation does not influence the final result.

The research community is divided into two camps about the question whether a decent language may contain such a non-deterministic operation. Some people strongly oppose the idea of non-determinism in a language, as it may cause ambiguous results. Especially in the field of database manipulation languages, this feeling has many adherents. However, some efforts have been made for allowing a limited form of non-determinism, e.g. the notion of semi-determinism introduced by Van den Bussche & Van Gucht [1992]. Semi-determinism is a form of non-determinism where all possible results are isomorphic. Another example is the B-approach\(^1\) [Abrial, 1992], that offers the choice-operation which allows for selecting an arbitrary element from a set.

Our RA-operation offers maximal non-determinism. If one wants to "protect" oneself by not using this full non-determinism, one may decide to use only the repetition mechanisms from Section 3.4.1.

### 3.3 A Graphical Version of SCA

In this section, we introduce SCA\(_G\), which is a graphical version of SCA. We do this by sketching how SCA\(_G\) could look like. Using SCA\(_G\) in practical situations may provide suggestions for changes.

As GOOD is inherently graphical and much research concerning GOOD has been conducted, we have tried to define SCA\(_G\) as much as possible in the spirit of GOOD. However, because the difference between the GOOD operations that have always exactly one "pattern-parameter" and the SCA-operations that sometimes have more than one sos-parameter, some differences were necessary. The examples below illustrate this. All examples are given in the context of the schema in Figure 2.3.

**Example 3.3.1 (Value Addition)** The following operation adds to all persons who own a car with a speed of 100 the attribute city with value Boston. This can be depicted by the operation in the following figure.

---

\(^{1}\)Suppose that in the original instance \( I \) it holds for each A-object that its x-value equals zero iff its y-value equals zero.

\(^{10}\)B is a successor of Z [Spivey, 1988].
Example 3.3.2 (Value Deletion) The following operation deletes the name attribute from all persons who live in Boston. This can be depicted by:

\[
\text{VD}(\text{J}_{\text{person}}[\text{city} = \text{Boston}], \text{name})
\]

Example 3.3.3 (Simple Object Creation) The following operation creates a simple object of type BCPerson in the container with label \(f\) for each person living in Boston who owns a car, and an is-reference from the created BCPerson-object to that person. This is not part of the schema, but can be used as an intermediate result for computing a more comprehensive manipulation. When the SC-operation without reference parameter is applied, we have to mark the "starting point" of the simple object selector (in this example this would be person.) This can be depicted by the operation in the following figure.
Example 3.3.4 (Simple Object Deletion) The following operation deletes each person living in the same city as the person with name Peter. This can be depicted by:

\[ SD(J_{\text{person}} \mid \text{city} = \text{person} \mid \text{city, name} = \text{Peter}) \]

Example 3.3.5 (Reference Creation) The following operation creates an owns-reference for each student whose name is Peter to the cars of the persons who are called Rosita. This can be depicted by the operation in the following figure. Reference deletion can be performed analogously; the reference to be deleted is depicted by a dashed arrow.
3.4 Advanced Operations

Example 3.3.6 (Container Modification) The following operation moves each car that is owned by a student to the container with label StudCars. This can be depicted by:

\[
\text{RC}\langle J_{\text{student}}[\text{name} = \text{Peter}] \text{car} \text{owns} \quad J_{\text{person}}[\text{name} = \text{Rosita}] \text{owns} \rangle
\]

\[
\text{CM}\langle J_{\text{car}}[\text{owns} \text{student}][], \text{StudCars} \rangle
\]

3.4 Advanced Operations

In this section, we show a number of high-level operations that can be expressed by means of the primitive SCA operations. For specific application domains, this layer of high-level SCA operations may be extended.

We use non-recursive \(\lambda\)-abstraction for defining these high-level operations. This means that it is allowed to assign a name to a certain SCA expression, which may hold some parameters. The mechanism is non-recursive, which means that the defining expression may not contain the defined name, neither directly nor indirectly. For formulating recursive expressions, the recursor operator (cf. Definition 3.2.25) can be used.
3.4.1 Non-recursive and Fix-Point Repetition

Often, a repetitive operation is required that is not recursive. This operation, called For All simple objects (FA) evaluates an sos in an instance and then applies some SCA expression for all these simple objects, in some arbitrary order. In order to simulate this operation, we need a fresh attribute name h.

\[ FA_{\alpha, \eta}(I, \tau[C], E_{\alpha, \eta}) = \]
\[ RA_{\alpha, \eta}(VA(I, \tau[C], h, 1), \]
\[ \tau[h = 1], \]
\[ E_{\alpha, \eta}(VD(I, x, h))) \]

This operation works as follows. First, all simple objects upon which the FA operation is to be applied, are marked by a fresh attribute h with value 1. Then, from all objects with h = 1, one is chosen randomly. Call this simple object σ. The expression that is to be repeated by the FA operation is applied under substitution of x by σ and the h-value of σ is deleted. Then, another object with h = 1 is selected and the process starts all over again. Note that the argument expression is executed exactly once for each simple object satisfying C.

The following example shows how FA can be used to express the operations that were mentioned on pages 59 and 62.

**Example 3.4.1** Consider the schema from Figure 3.13\(^\text{11}\). We suppose that each person participates in at most one marriage and that each marriage concerns two persons, of which one has to be the husband of the other.

The following operation creates a new simple object of type marriage for each pair of persons who are married to each other (i.e., there is a husband of reference from one of them to the other). Moreover, a husband-reference to the husband in the marriage is created.

\[ J' = FA_{\alpha, \eta}(I, person [\text{husband.of} person []], \]
\[ SC(I, marriage, \alpha, \ell, \text{husband})) \]

The following operation creates a wife reference from each marriage to the person with whom the husband in the marriage has a husband of reference.

\[ J' = FA_{m, \eta}(I, marriage [], \]
\[ RC(I, m, person [\text{husband.of} person [\text{husband of} person [\text{husband of} person [\text{husband of} person [\text{husband of} person [\ell m]], wife])]) \]

Another deterministic repetitive operation is the fix-point-operation (FP). Assume that H is a fresh simple type name. If x and J' are variables, sos is a simple object selector and \( E_{x, J'} \) is an SCA expression that may contain x and J', then

\(^{11}\text{Only the aspects that are relevant for this example are depicted.}\)
3.4 Advanced Operations

$FP_x.Y(J, sos, E_{x,y})$ determines the fix-point of $E_{x,y}$ in $J$, if it exists:

$$FP_x.Y(J, sos, E_{x,y}) =$$
$$SD(RA_{w,j}(SC(J, H, true, f),$$
$$\mu([],$$
$$FA_{x,y}(J'', sos, E_{x,y}))),$$
$$\mu([]))$$

This operation gives a deterministic fix-point, because the repetition is executed layer by layer: first, the simple objects are selected for which $E$ is to be applied. Then, $E$ is executed for each of these objects. Only then, the sos is evaluated again and the process is repeated. The process stops when nothing changes anymore (if ever), so when the fix-point has been reached.

If a designer opposes to the non-deterministic nature of RA, he may use FP instead. FP gives him maximal "deterministic expressive power".

3.4.2 Selective Expressions

The next class of expressions that we want to discuss is the class of the so-called selective expressions. The application of such an expression on some SimCon-instance $I$ results in the application of either some SCA expression $E_1$ on $I$ or the application of some SCA expression $E_2$ on $I$, depending on a given condition $C$ (expressed in terms of $I$). Such a selective expression is written as

$$\text{if } C \text{ then } E_1 \text{ else } E_2 \text{ fi}$$

The condition $C$ is an expression $(sos_1 \mathcal{R} sos_2)$, where $\mathcal{R} \in \{\subseteq, =, \supseteq\}$ and where $sos_1$ and $sos_2$ evaluate to sets of simple objects of the same type. Conditions can also be combined using conjunction, disjunction and negation. We demonstrate that selective expressions can be expressed in SCA.
Consider the following selective expression \( E \):

\[
\begin{align*}
\text{if } & sos_1 \subseteq sos_2 \\
& \text{then } E_1 \\
& \text{else } E_2 \\
\end{align*}
\]

Let \( I \) be some \( \text{SimCon} \)-instance, which holds a container with label \( \ell \). We construct an expression in \( \text{SCA} \) that is equivalent to \( E(1) \). This expression is a functional one, although we split it into parts for presentation purposes. Suppose that \( sos_1 \) and \( sos_2 \) evaluate to sets of simple objects of type \( \tau \) and let \( x \) be a fresh attribute name.

Informally, the \( \text{SCA} \) expression is such that we first add to all simple objects of type \( \tau \) that satisfy \( sos_1 \) an \( x \)-attribute with value 1. Then we add/change the \( x \)-value of all objects in \( sos_2 \) into the value 2. It is clear that the absence of simple objects of type \( \tau \) with \( x = 1 \) implies that the set satisfying \( sos_1 \) is a subset of the set satisfying \( sos_2 \). In that case, \( E_1 \) has to be applied. Otherwise, \( E_2 \) has to be applied.

The formal expression realizing the assignment of the attribute values is:

\[
I_1 = VA(VA(I,sos_1,x,1),sos_2,x,2)
\]

Then, we create a new simple object of a new type \( \tau' \), and an \( \tau \)-reference to all objects with \( x = 1 \), if any. Afterwards, \( E_2 \) is executed iff there are any of such references. Otherwise, \( E_1 \) is executed:

\[
\begin{align*}
I_2 &= SC(I_1,\tau',\text{true},\ell) \\
I_3 &= RC(I_2,\tau'[\ell,1],\tau) \\
I_4 &= RA_{\chi}(I_3,\tau'[\ell,1],\tau,\text{SD}(E_2(I,\tau'[\ell,1]))) \\
I_5 &= RA_{\chi}(I_4,\tau'[\ell,1],\tau,\text{SD}(E_1(I,\tau'[\ell,1])))
\end{align*}
\]

\( I_5 \) is the instance that is obtained by applying the selective expression \( E \) on \( I \): \( E(1) = I_5 \).

It is obvious that an analogous construction is possible if \( \subseteq \) is replaced by \( \supseteq \) in \( E \). The results for \( \subseteq \) and \( \supseteq \) can be combined to cover the situation where \( \subseteq \) is replaced by \( = \) in \( E \).

Using the logical connectives \( \wedge, \vee, \neg \), the above mentioned simple conditions can be combined into more complex ones. These more complex conditions are also expressible, as can be concluded from the following equations:

\[
\begin{align*}
&\text{if } \neg C \text{ then } E_1 \text{ else } E_2 \text{ fi } \equiv \text{ if } C \text{ then } E_2 \text{ else } E_1 \text{ fi } \\
&\text{if } C_1 \wedge C_2 \text{ then } E_1 \text{ else } E_2 \text{ fi } \equiv \text{ if } C_1 \text{ then } \\
&\quad \text{ if } C_2 \text{ then } E_1 \text{ else } E_2 \text{ fi } \text{ else } E_2 \text{ fi } \\
A \vee B \equiv \neg(\neg A \wedge \neg B)
\end{align*}
\]
It is also possible to simulate a selective expression which compares two value selectors in its 12-part.

3.4.3 Picking an Element from a Set

Often, we need an operation that picks an arbitrary element from a set. In our case, this would be an operator that takes an arbitrary simple object of a specified type from a given container. We implement this by the operator \( \text{pick}(J, T[C], x, v_1, v_2, \ell) \).

Here, \( J \) is a SimCon-instance, \( T \) a simple type, \( C \) is a sos-condition, \( x \) an attribute name, \( v_1 \) and \( v_2 \) values belonging to the type of \( x \) and \( \ell \) a container label. This operation assigns to one simple object of type \( T \) satisfying \( C \) in the container with label \( \ell \) in \( J \) the attribute \( x \) with value \( v_1 \) (if any) and to all other \( T \)-objects in the \( \ell \)-container \( x = v_2 \).

Let \( v \) be a value of the same type as \( v_1 \) with \( v \neq v_1 \land v \neq v_2 \). The \textit{pick}-operation is implemented as follows:

\[
\text{pick}(J, T[C], x, v_1, v_2, \ell) = I'' \text{ where } \\

I' = \text{ VA}(J, T[C], x, v) \\
I'' = \text{ RA}(\text{VA}(I', T[C], x, v), \text{VA}(\text{VA}(I', T[C], x, v_2), t, x, v_1))
\]

If \( x \) is an attribute of the simple type \( T \), this \textit{pick} is persistent. Otherwise, the information can only be used temporarily “within” the SCA expression that holds the \textit{pick}-expression.

This \textit{pick}-operation resembles the \textit{choice}-operation of Abrial [1992]. The functional language of ExSpect also contains a \textit{pick}-function [Van Hee, Somers & Voorhoeve, 1988; Verkoulen, 1989]; like this \textit{choice}-function, it returns an arbitrary but fixed element of a set. This is useful for simulation purposes.

3.4.4 Negation of Selectors

As we have seen, selectors can be used to specify which simple objects have to be affected by a certain operation. This selection is done upon positive criteria, i.e. the existence of certain object-reference constructions and/or attribute values. However, sometimes we want to be able to select simple objects that do not have certain references and/or attributes. Let \( x \) be a fresh attribute name. Then, an operation of the form \( \text{O}(J, T[-C], \ldots) \) is implemented as follows\(^\text{12}\):

\[
\text{O}(J, T[-C], \ldots) = I_4 \text{ where } \\

I_1 = \text{ VA}(I, T[\cdot], x, 1) \\
I_2 = \text{ VA}(I_1, T[C], x, 0) \\
I_3 = \text{ O}(I_2, T[x = 1], \ldots) \\
I_4 = \text{ VD}(I_3, T[\cdot], x)
\]

\(^{12}\text{O} \text{ can be any SCA operator, except for RA.}\)
All objects of type \( \tau \) are marked with \( x = 1 \). Then, the ones that satisfy \( C \) are assigned \( x = 0 \). So, all objects of type \( \tau \) that do not satisfy \( C \) have \( x = 1 \).

### 3.4.5 Ordering an Unordered Relation

Another non-deterministic operation that can be useful is the ordering of an unordered relation. If the simple objects to be ordered are all equivalent (in some sense), all possible results of this operation are indistinguishable from each other. A simple example of such a situation is an instance that only contains simple objects without any attributes or references. However, if the objects to be ordered are not equivalent, all possible results are not indistinguishable. The designer who uses this operation should decide himself whether this is a problem or not.

Consider a schema with a simple type \( X \) and a reference \( r \) from \( X \) to \( X \). Suppose we want to order a set of \( X \)-objects by making an \( X \)-\( r \) chain. Let \( a \) and \( b \) be fresh attributes. Then, the instance \( I_4 \) is the required instance, where

\[
\begin{align*}
I_1 &= VA(I_{\chi[\cdot]}, a, 0) \\
I_2 &= pick(I_1, x[\cdot], b, 1, 0, \ell) \\
I_3 &= VD(I_2, x[b = 1], a) \\
I_4 &= RA_{\chi[\cdot]}(I_3, x[a = 0], \left. \begin{array}{l}
VD(VA(VD(RC(/I_4, x[b = 1], x), x), x), x, b), \\
x, b, 1)
\end{array} \right. \\
& \quad x, a)
\end{align*}
\]

We first assign \( a = 0 \) to each \( X \)-object. Then, we set \( b = 1 \) for one \( X \) (and delete the \( a \)-value for this object) and \( b = 0 \) for all other \( X \)-objects. Now, as long as there are \( X \)-objects with \( a = 0 \), we chose one (say \( x \)), create a reference from the \( X \) with \( b = 1 \) to \( x \), delete the \( b \)-attribute of the \( X \) with \( b = 1 \) and change the \( a = 0 \) of \( x \) into \( b = 1 \).

It is obvious that this realizes the required operation.

### 3.4.6 Transitive Closure of a Relation

References in the SimCon Object Model may represent relations. Often, we want to be able to compute the so-called transitive closure of a relation. We illustrate this by means of an example. Consider the simple schema from Figure 3.14. It consists of the simple type person and two references, viz. child.of and descendant.of. Given the child.of-relation, we now want to be able to compute the descendant.of-relation.
The instance $I_2$ contains this information, where

$$I_1 = FA_{p_1,I_1}(\{person\},$$
$$FA_{p_2,I_1}(\{I'_i, person[I_i, child_of = p_1],$$
$$RC(I'', p_2, p_1, descendant_of))))$$

$$I_2 = RA_{p_1,I_1}(\{I_1, person[\],$$
$$RA_{p_2,I_1}(\{I'_i, person[I_i, descendants_of = p_1],$$
$$RA_{p_3,I_1}(\{I''_i, person[I_i, child_of = p_2],$$
$$RC(I''_i, p_3, p_1, descendant_of))))$$

In $I_1$, a descendant_of-reference is added for each pair of nodes between which the child_of-relation holds. In $I_2$, a descendant_of-reference is added from each person $p_3$ to each person $p_1$ such that there is a descendant_of-reference from some $p_2$ to $p_1$ and a child_of-reference from $p_3$ to $p_2$.

Some of the RA-applications might have been replaced by FA-applications. For reasons of symmetry, we have not done so. With respect to the complexity, both alternatives are equivalent.

3.4.7 The Train-Bag-Letter Example

We have seen that SimCon does not allow container objects to hold containers again: containers may only hold simple objects. We have also explained our reason for this choice. However, it is possible to simulate the occurrence of containers in containers. We demonstrate this by means of an example.

Consider the object schema from Figure 3.15. It is reasonable that someone wants to model a situation where bags that are filled with letters are loaded into a train. In practice, this is not done by shaking the bags out, but by putting the bags into the train with all the letters in them. In this case, this would mean that containers of type C_bag would have to be inserted into a container of type C_train. Below, we introduce operations load and unload. These operations can be used to realise this situation.

Let $b$ be the label of a container of type C_bag and $t$ the label of a container of type C_train.
3. The SimCon Algebra

\[ \text{load}(i, b, t) = \]
\[ \text{CM}(\text{CM}(\text{CM}(1, \text{bag}, b \{\}, t), \text{letter}, b \{\}, t)) \]

\[ \text{unload}(i, t) = \]
\[ \text{CM}(\text{CM}(\text{CM}(\text{pick}(1, \text{bag}, x, 0, 1, t), \text{c.bag}, \{b\}), \text{bag}(x = 0), b), \text{letter}^{\text{holds}} \text{bag}_b \{\}, b) \]

3.5 Expressive Power of the SimCon Algebra

In this section, it is shown that GOOD and the Nested Relational Algebra can be expressed by SimCon\(^\text{13}\). This means that SimCon is at least as expressive as these two models. Moreover, we prove that SimCon is really more powerful than GOOD by showing a manipulation that is expressible in SimCon, but that cannot be expressed by GOOD.

We start by showing that SimCon is at least as powerful as GOOD. For this purpose, we give a short introduction to GOOD.

A GOOD object base scheme \( S \) is a tuple \((\text{NPOL}, \text{POL}, \text{FEL}, \text{NFEL}, \mathcal{P})\) where
- \( \text{NPOL} \) is a finite set of non-printable object labels;
- \( \text{POL} \) is a finite set of printable object labels;
- \( \text{FEL} \) is a finite set of functional edge labels;
- \( \text{NFEL} \) is a finite set of non-functional edge labels;

\(^\text{13}\) For proving the latter one, we apply a result stating that GOOD can express each expression from the Nested Relational Algebra and our result stating that SCA is at least as strong as GOOD.
3.5 Expressive Power of the SimCon Algebra

- \( \mathcal{P} \subseteq NPOL \times (NFEL \cup FEL) \times (NPOL \cup POL) \)

The sets \( POL, NPOL, FEL \) and \( NFEL \) are pairwise disjoint. Moreover, there is some function \( \pi \) assigning to each printable object label some set of constants. It may have been more elegant to include this \( \pi \) in the tuple \( S \). However, we have complied ourselves to the definition of \( GOOD \) [Gyssens, Paredaens & Van Gucht, 1990], where there is one abstract function \( \pi \) which "holds" for all \( GOOD \) schemas. Example 3.5.1 gives a \( GOOD \) object base scheme.

**Example 3.5.1 (GOOD object base scheme)** In Figure 3.16, an example \( GOOD \) object base scheme is depicted.

![Diagram](image)

Figure 3.16: An example \( GOOD \) schema

There are two non-printable object types, viz. person and birth. Furthermore, there are three printable nodes gender, string and date. There are five edges between nodes; child is a non-functional (multi-valued) edge, whereas the other ones has, is, country and day are functional. It cannot be depicted in the picture that string contains all text strings, date consists of all valid dates and gender has two values, viz. M and F.

First, we show that a \( GOOD \) object base scheme (a \( GOOD \) schema) can be represented by a SimCon object schema.

**Lemma 3.5.2 (representation of \( GOOD \) schema)** Let \( S \) be a \( GOOD \) object base scheme as described above. Then, it is possible to define an equivalent SimCon object schema \( \Sigma_S \).

**Proof**

For representing an object base scheme \( S \), we have to define a SimCon object schema \( \Sigma_S \). However, we only describe a simple schema and one container type \( C \) consisting of all simple types in that simple schema. The other aspects of the object schema are not relevant. The tuple \((ST, AT, RF, IS)\) represents this simple schema, with

- \( ST = NPOL \cup POL \)
$AT = (AN, AS, DD)$ where

$AN = POL$

For $a \in AN$, $AS.a.\alpha = \pi(a)$, and there are no other elements in $AS$.

$DD = \bigcup_{a \in POL} \pi(a)$.

$RF = (RN, RS)$ where

$RN = FEL \cup NFEL$

$RS.\alpha.\xi = \eta$ for $(\xi, \eta, y) \in P$ and this defines all elements in $RS$ completely.

$IS = \emptyset$

For functional edges, an injectivity constraint should be added. However, we do not pay attention to this aspect here, as the semantics of GOOD does not really cover functional edges: if an operation is applied upon an instance such that the functionality constraint would be violated, the result is undefined. So, it is the responsibility of the designer that this constraint remains invariant.

This transformation is rather straightforward. Only the simulation of printable nodes deserves some additional explanation. In GOOD, an edge to a printable node has an edge label. In SimCon, the attributes are connected to the simple types with an unlabelled edge. Because of this difference, we have included for each printable a simple type with exactly one attribute. So, the printable gender from Example 3.5.1 is simulated by a simple type gender with one attribute gender. If we would not have done this, it would not have been possible to simulate a GOOD schema where two non-printables both have an edge with a different label to one and the same printable.

A GOOD instance consists of a set of nodes and a set of edges, according to some GOOD scheme. More formally, let $S = \langle NPOL, POL, FEL, NFEL, P \rangle$ be a GOOD object base scheme. An instance over $S$ is a labelled graph $I = \langle N, E \rangle$ where

- $N$ is a finite set of nodes. If $n$ is a node in $N$, its label\(^{14}\) $\lambda(n)$ should be in $NPOL \cup POL$.
- There is a function $\text{print}$ that assigns to each printable node its print label. This is the value of the node. It should hold that $\text{print}(n) \in \pi(\lambda(n))$.
- $E$ is a set of edges of the form $(n, \alpha, m)$. It has to hold that $m, n \in N$ and that $(\lambda(n), \alpha, \lambda(m)) \in P$.
- If $\alpha \in FEL$ and $(m, \alpha, n_1) \in E$ and $(m, \alpha, n_2) \in E$, then $n_1 = n_2$.
- If $\lambda(n_1) = \lambda(n_2) \in POL$ and $\text{print}(n_1) = \text{print}(n_2)$, then $n_1 = n_2$.

Given the transformation of Lemma 3.5.2, it is obvious how a GOOD instance can be represented by a SimCon-instance. Non-printable nodes are represented by simple objects, edges by references and printable nodes by simple objects with an attribute value. All these simple objects are element of one container object. This

\(^{14}\)There is a global function $\lambda$, assigning a label to each node.
way, the aforementioned requirements for a GOOD instance are satisfied.

All GOOD operators work in the context of a schema \( S \). Our simulation is defined in the context of \( \Sigma_S \) (cf. Definition 3.5.2).

The first GOOD operator we are going to simulate is the Node Addition. If \( J \) is a GOOD pattern, \( S \) an object base scheme, \( I \) an instance, \( K \) a node label, \( \alpha_1, \ldots, \alpha_n \) edge labels and \( m_1, \ldots, m_n \) the corresponding node labels, then \( \text{NA}[J, S, I, K, \{(\alpha_1, m_1), \ldots, (\alpha_n, m_n)\}] \) creates for each embedding of \( J \) in \( I \) a \( K \)-node with an \( \alpha_i \)-edge to the \( m_i \)-node (which is part of \( J \)), if \( I \) does not hold yet a \( K \)-node with these properties.

**Lemma 3.5.3 (simulation of NA)** Let \( J, S, I, K, \alpha_1, \ldots, \alpha_n \) and \( m_1, \ldots, m_n \) be as given above. The Node Addition \( \text{NA}[J, S, I, K, \{(\alpha_1, m_1), \ldots, (\alpha_n, m_n)\}] \) is expressible in SCA.

**Proof**

Let \( I \) be the SimCon-instance representing \( I \). Suppose that the pattern \( J \) consists of the nodes \( p_1, \ldots, p_j \), for some \( j > 0 \). Suppose these nodes have type \( T_i \), for \( 1 \leq i \leq j \). Suppose also that \( p_i = m_i \) for \( 1 \leq i \leq n \); we assume a certain ordering of the pattern \( J \). However, this ordering does not influence the final result. Finally, we use the following two abbreviations:

- \( \text{at}(p_i) \) denotes the value constraint upon \( p_i \), i.e. if \( p_i \) is a printable, this can be used to express that it has to have a certain value;
- \( \text{rf}(\{p_1, \ldots, p_{i-1}\}) \) denotes the reference constraints upon \( p_i \), i.e. it demands \( p_i \) to have certain edges to and/or from the nodes \( p_1, \ldots, p_{i-1} \). The relationships with nodes \( p_{i'} \) for \( i' > i \) are covered by the conditions upon \( p_{i'} \).

The following SCA expression realizes this node addition:

\[
\text{FA}_{p_1, i}(I, T_i, \text{at}(p_i)), \ldots, \text{FA}_{p_1, i}(I_{i-1}, T_i, \text{at}(p_i), \text{rf}(\{p_1, \ldots, p_{i-1}\})), \\
\quad \text{if } k^{\geq i} (m_1, \ldots, m_n) = \emptyset \\
\quad \text{then } c(I_i) \\
\quad \text{else } I_i \\
\quad \text{fi} \\
\text{fi} \\
\text{fi} \\
\ldots)
\]

Here, the expression \( c \) creates one new object of type \( K \) and adds edges \( \alpha_k \) from this object to the matched objects \( p_k = m_k \):

\[
\text{CM}(\text{RC}(\ldots \text{RC}(\text{SC}(I, K, \text{true}, t_n), \\
\quad K, t_n[\cdot] p_n, \alpha_n), \ldots)), \\
\quad \ldots)
\]

\]
\( k, \ell, [\lambda, p_1, \alpha] \).

Intuitively, this simulation works as follows. In the first \( j \) lines of the first expression, the embedding of the pattern \( j \) is simulated. Each of the nodes in the pattern gets a label \( p_i \) that can be used to refer to that node in the "remainder" of the SimCon expression. If for this embedding, \( I \) holds already a node with label \( K \), then nothing happens. Otherwise, such a node is created, together with the \( \alpha_i \)-references to the nodes \( m_i = p_i \).

**Example 3.5.4 (NA)** Consider the object base scheme from Example 3.5.1. Furthermore, consider the NA-operation from Figure 3.17, which adds an object of the "free" non-printable object type \( \text{NLbirth} \) to each person born on the same date as some person who has been born in the Netherlands.

![Diagram](image)

Figure 3.17: An example Node Addition

The construction from the previous lemma gives the following SCA expression simulating this Node Addition:

\[
\begin{align*}
FA_{p_1, \lambda}(I_{p_{\text{person}}} [\lambda]), \\
FA_{p_2, \lambda}(I_{p_{\text{person}}} [\lambda]), \\
FA_{p_3, \lambda}(I_{\text{birth}} [\lambda] \text{ p}_1), \\
FA_{p_1, \lambda}(I_{\text{birth}} [\lambda] \text{ country = "NL", p}_2, \text{ day} \text{ p}_3]), \\
\text{if NLbirth}[\lambda] = \emptyset \\
\text{then CM(RC(SC(I_{\lambda}, NLbirth, \text{true, } \ell))}, \\
\text{NLbirth}[\lambda] \text{, p}_1, \ell), \\
\text{else } I_{\lambda} \\
\text{fi}
\end{align*}
\]

The next GOOD operator we are going to simulate is the Edge Addition. If \( j \) is a GOOD pattern, \( S \) an object base scheme, \( I \) an instance, \( \alpha_1, \ldots, \alpha_n \) edge
labels and $m_1, \ldots, m_n$ and $m_1', \ldots, m_n'$ node labels, then the result of applying $EA[J, S, I, \{(m_1, \alpha_1, m_1'), \ldots, (m_n, \alpha_n, m_n')\}]$ is the creation, for each embedding of $J$ in $I$, of an $\alpha_i$-edge from $m_i$ to $m'_i$ (which are part of $J$), for $1 \leq i \leq j$.

**Lemma 3.5.5 (simulation of EA)** Let $J, S, I, m_1, \ldots, m_n, \alpha_1, \ldots, \alpha_n$ and finally $m_1', \ldots, m_n'$ be as described above. In that case, it is possible to express the Edge Addition $EA[J, S, I, \{(m_1, \alpha_1, m_1'), \ldots, (m_n, \alpha_n, m_n')\}]$ in SCA.

**Proof**

Let $I$ be the SimCon-instance representing $I$. Suppose that the pattern $J$ consists of the nodes $p_1, \ldots, p_j$, for some $j > 0$. Suppose that $p_i = m_i$ and $p_{t(i)} = m_i'$, where $f$ is a automorphism on $\{1, \ldots, n\}$ and $1 \leq i \leq n$. Furthermore, we use the abbreviations $rf$ and $at$ from Lemma 3.5.3. Then, the following SCA expression realizes this edge addition:

$$
\begin{array}{c}
FA_{D, I}(I_{D}, [at(p_1)], \\
\ldots \\
FA_{p_i, I}(I_{D}, [at(p_i)], rf([p_1, \ldots, p_{j-1}]), \\
RC(\ldots (RC(I_{D}, p_{n}, p_{t(n)}, \alpha_n), \ldots), \\
p_i, p_{t(i)}, \alpha_i) \\
) \\
\ldots
\end{array}
$$

The next GOOD operator we are going to simulate is the Node Deletion. If $J$ is a GOOD pattern, $S$ an object base scheme, $I$ an instance and $m$ a node label, then $ND[J, S, I, m]$ deletes for each embedding of $J$ in $I$ each $m$-node (which is part of $J$) with all its incoming and outgoing edges.

**Lemma 3.5.6 (simulation of ND)** The Node Deletion $ND[J, S, I, m]$ is expressible in SCA.

**Proof**

Let $I$ be the SimCon-instance representing $I$. Suppose that the pattern $J$ consists of the nodes $p_1, \ldots, p_j$, for some $j > 0$ and that $p_j = m$. Let $x$ be a free attribute name. Furthermore, we use the abbreviations $rf$ and $at$ from Lemma 3.5.3. Finally, suppose that according to the schema $S$, all objects of a type in the set $\{\tau_1, \ldots, \tau_s\}$ may have a reference to $m$. Then, the following SCA expression realizes this node deletion:

$$
\begin{array}{c}
SD(RD( \\
\ldots (RD( \\
FA_{p_1, I}(I_{D}, [at(p_j)], \\
\ldots \\
FA_{p_j, I}(I_{D}, [at(p_j)], rf([p_1, \ldots, p_{j-1}]), \\
VA(I_{D}, p_1, x, 0) \ldots, \\
\tau_1[I_{D}, [x = 0]]_{p_1, [x = 0], \tau_1}, \\
\ldots)
)\ldots
)\ldots
)\ldots
)\ldots
)\ldots
)\ldots
)
\end{array}
$$
The next GOOD operator we are going to simulate is the Edge Deletion. If \( J \) is a GOOD pattern, \( S \) an object base scheme, \( I \) an instance, \( \alpha_1, \ldots, \alpha_n \) edge labels and \( m_1, \ldots, m_n \) and \( m'_1, \ldots, m'_n \) node labels, then the Edge Deletion of the form \( \text{ED}[J, S, I, \{(m_1, \alpha_1, m'_1), \ldots, (m_n, \alpha_n, m'_n)\}] \) deletes for each embedding of \( J \) in \( I \) the \( \alpha_i \)-edge from \( m_i \) to \( m'_i \) (which are part of \( J \)), for \( 1 \leq i \leq j \).

**Lemma 3.5.7 (simulation of ED)** Let \( J, S, I, m_1, \ldots, m_n, \alpha_1, \ldots, \alpha_n \) and finally \( m'_1, \ldots, m'_n \) be as described above. In that case, it is possible to express the Edge Deletion \( \text{ED}[J, S, I, \{(m_1, \alpha_1, m'_1), \ldots, (m_n, \alpha_n, m'_n)\}] \) in SCA.

**Proof**

Let \( I' \) be the SimCon-instance representing \( I \). Suppose that the pattern \( J \) consists of the nodes \( p_1, \ldots, p_j \), for some \( j > 0 \). Suppose that \( p_i = m_i \) and \( p_{\gamma(i)} = m'_i \), where \( f \) is a automorphism on \( \{1, \ldots, n\} \) and \( 1 \leq i \leq n \). Furthermore, we use the abbreviations \( r_f \) and \( a_f \) from Lemma 3.5.3. Finally, let \( d_1, \ldots, d_n \) be fresh reference names not occurring in \( \Sigma_S \).

We split the SCA expression that realizes this edge deletion into two parts in order to make it more surveyable. The first part gives an instance \( I' \). The second part yields the final result \( I'' \).

\[
I' = FA_{p_1, 1, (I, p_1, [\text{at}(p_1)]},
\]

\[
\ldots
FA_{p_1, 1, (I, p_1, [\text{at}(p_1)], \text{rf}((p_1, \ldots, p_{j-1}))},
\]

\[
\text{RC}(\ldots \text{RC}(I, p_n, p_{\gamma(n)}, d_n), \ldots,
\]

\[
p_1, p_{\gamma(1)}, d_1)
\]

\[
\ldots)
\]

\[
I'' = FA_{p_1, 1, (I', p_1, [\text{at}(p_1)]},
\]

\[
\ldots
FA_{p_n, 1, (\ldots, [\text{at}(p_n)]},
\]

\[
\text{RD}(\ldots \text{RD}(I, p_n, p_{\gamma(n)}, [\text{at}(p_n), d_n], \ldots,
\]

\[
p_1, p_{\gamma(1)}, [\text{at}(p_1), \alpha_1])
\]

\[
\ldots)
\]

The final GOOD operator we are going to simulate is the Abstraction. The Abstraction can be used to group objects in equivalence classes. Each node refers to exactly those elements from one such equivalence class. For instance, it is possible to classify people with respect to their profession, by creating a node for
each profession and grouping all people with the same profession under one and the same profession-node.

If \( J \) is a GOOD pattern, \( S \) an object base scheme, \( I \) an instance, \( N \) and \( K \) are node labels and \( \alpha_1, \ldots, \alpha_n \) and \( \beta \) are edge labels, then the abstraction \( AB[J, S, I, N, K, \{ \alpha_1, \ldots, \alpha_n \}, \beta] \) creates for each equivalence class of \( N \)-nodes selected by \( J \) a \( K \)-node with a \( \beta \)-edge to all elements in the equivalence class. The properties that determine whether two \( N \)'s are in the same class are represented by \( \alpha_1, \ldots, \alpha_n \). A \( K \)-node is only added for equivalence classes for which the original instance does not yet hold a \( K \)-node.

**Lemma 3.5.8 (simulation of \( AB \))** Let \( J, S, I, N, K, \beta \) and \( \alpha_1, \ldots, \alpha_n \) be as described above. The Abstraction \( AB[J, S, I, N, K, \{ \alpha_1, \ldots, \alpha_n \}, \beta] \) is expressible in SCA.

**Proof**

Let \( I \) be the SimCon-instance representing \( I \). Suppose that the type of the node to which \( N \) has a reference \( \alpha_i \) is \( T_i \) (\( 1 \leq i \leq n \)). Finally, let \( old \) and \( t \) be fresh attribute names not occurring in \( \Sigma_S \). The following sequence of SCA expressions realizes the aforementioned Abstraction.

\[
\begin{align*}
I_1 &= VA[I, K[\{ \alpha_i \}], old, t] \\
I_2 &= SC[I, K, N[\alpha_i \in_T \{ \alpha_i \}], \ldots, \alpha_n \in_T \{ \alpha_n \}], t, \beta) \\
I_3 &= VA[I, K[\{ \alpha_i \}], t, 0] \\
I_4 &= RA[I, K[\{ \alpha_i \}], t = 0], \\
& \quad FA[I, K[\{ \alpha_i \}], old, t = 0], \\
& \quad if \quad \tau_i[\alpha_i \in_T \{ \alpha_i \}] = \tau_i[\alpha_i \in_T \{ \alpha_i \}] \wedge \ldots \wedge \tau_n[\alpha_n \in_T \{ \alpha_n \}] = \tau_n[\alpha_n \in_T \{ \alpha_n \}] \\
& \quad then \quad VA[RC(\Sigma_D[I', K[\{ \beta \}, \beta, y], y, t, 1)] \\
& \quad else \quad I' \\
& \quad fi \\
I_5 &= FA[I, K[\{ \alpha_i \}], old = 1], \\
& \quad FA[I, K[\{ \alpha_i \}], \\
& \quad if \quad \tau_i[\alpha_i \in_T \{ \alpha_i \}] = \tau_i[\alpha_i \in_T \{ \alpha_i \}] \wedge \ldots \wedge \tau_n[\alpha_n \in_T \{ \alpha_n \}] = \tau_n[\alpha_n \in_T \{ \alpha_n \}] \\
& \quad then \quad SD[I', \alpha_i] \\
& \quad else \quad I' \\
& \quad fi \\
& \quad fi \\
\end{align*}
\]

The last GOOD primitive that has to be simulated is the method construct. However, Van Rossum [1992] has demonstrated for a variant of GOOD that the method construct can be simulated by the so-called node loop. This node loop recursively applies a sequence of GOOD operations on an instance as long as the instance contains a node with a certain object label. Post [1993] states that this
equivalence between the node loop and the method construct also holds for GOOD. The node loop construct is less complicated, so it can be simulated more easily by an SCA expression.

Let $I$ be an instance, $K$ an object label and $\mathcal{G}$ a sequence of GOOD operations. The Node Loop $\text{NL}[I, K, \mathcal{G}]$ applies this sequence $\mathcal{G}$ to $I$ as long as at least one node with label $K$ occurs in $I$.

**Lemma 3.5.9 (simulation of NL)** Consider the parameters $I$, $K$ and $\mathcal{G}$ as described above. The Node Loop $\text{NL}[I, K, \mathcal{G}]$ is expressible in SCA.

**Proof**

Let $I$ be the SimCon-instance representing $I$. Also, suppose that $\mathcal{G}_{\text{SCA}}$ is the SCA expression simulating $\mathcal{G}$. Then, the following SCA expression simulates the aforementioned Node Loop:

$$RA_{x}(I, K[|], \mathcal{G}_{\text{SCA}}).$$

The instance parameter of $\mathcal{G}_{\text{SCA}}$ is $I$.

**Theorem 3.5.10 (GOOD expressible in SCA)** All GOOD manipulations can be simulated by an SCA expression.

**Proof**

This follows directly from Lemmas 3.5.2 to 3.5.9.

**Theorem 3.5.11 (NRA expressible in SCA)** All expressions in the nested relational algebra (NRA) can be simulated by an SCA expression.

**Proof**

Van Rossum [1992] has proven that the nested relational algebra can be expressed by GOOD. According to Theorem 3.5.10, GOOD can be simulated by SCA. So, NRA can be simulated by SCA too.

The next theorem says that SCA and GOOD are not equivalent: there are operations that are expressible in SCA, but not in GOOD. After the theorem, we deal with the reasons for this.

**Theorem 3.5.12 (SCA more expressive than GOOD)** There exist SCA expressions that cannot be expressed in the GOOD language.
3.5 Expressive Power of the SimCon Algebra

Proof

Consider the schema in Figure 3.18. Also, an instance I consisting of two A-objects is depicted. Andries [1990] has proven that there is no GOOD expression that transforms this instance into an instance consisting of these two A-objects and four new B-objects that are connected as depicted in the figure.

![Diagram of A and B objects connected](image)

Figure 3.18: The 2-A-4B problem

In SCA, it is possible to express this operation. We first transform I into I₁ in order to distinguish the two identical A-objects, using the pick-operation\(^\text{15}\):

\[
I₁ = \text{pick}(I, A[], h, 0, 1, \ell)
\]

Similarly, a chain of four B's can be created which each have an h-attribute with a unique value between 0 and 3 and where a B with \(h = i\) refers to the B with \(h = i + 1 \mod 4\). Now, we can create references from the A-object with \(h = 1\) to the B's with \(h = 0, 2\) and from the A-object with \(h = 2\) to the B's with \(h = 1, 3\). This gives the required instance. □

The reason for this difference in expressive power is caused by the introduction of some non-determinism by the RA operation: recall that from the simple objects selected by its sos parameter, one of them is chosen arbitrarily.

\(^{15}\)Suppose the instance I consists of one container with label \(\ell\).
3.6 Concluding Remarks

In this chapter, we have introduced the SimCon Algebra. This algebra is the manipulation language of the SimCon Object Model. Besides some informal explanations, we have also given formal definitions. The kernel of the algebra has been kept as small as possible, in order to make it more easy to obtain theoretical results for the algebra. E.g., we have proved that SCA is at least as expressive as the Nested Relational Algebra and that it has more expressive power than GOOD.

Apparently, SCA offers sufficient expressive power. However, the kernel of SCA does not provide satisfactory expressive comfort: it may be too difficult to use. We have indicated how this can be solved, viz. by introducing operations that can be expressed in terms of the kernel, but that are on a higher level of abstraction. In the future, this library of high-level constructs may be extended. Also, a graphical version of SCA improves its expressive comfort.

The design of the SimCon Algebra has been such that it is suited for our verification purposes as described in Chapter 5. As can be seen in that chapter, all SCA expressions that do not contain the RA-operation can be dealt with. So we have concentrated the repetitive nature that may cause problems in the verification process in one operation, instead of "spreading" it over all SCA operations. This motivates the difference between our RC-operator and the Edge Addition of GOOD. The latter one is (in some definitions of GOOD) recursive, whereas the former one is not recursive. If RC would have been recursive, this might have caused problems with our verification technique. Furthermore, this RA-operation is not strictly necessary in the integrated framework, for its repetitive function can also be realised in the net structure.

In the next chapter, we deal with the integrated SimCon Net Model. We demonstrate the role of the SimCon Object Model and we show how SCA is used as the manipulation language within the SimCon approach.
Chapter 4

The SimCon Net Model

4.1 Introduction: SimCon and Petri Nets

In the previous two chapters, an object model and its associated manipulation language have been introduced. However, this is not sufficient for describing all dynamic aspects of a system. E.g., it is not possible to describe the order in which two activities should be pursued. We could have extended the SimCon Object Model with concepts like methods and messages. The SimCon Algebra would then also have to be extended in order to be able to deal with those concepts. This approach has been applied often in the field of object-oriented databases and object-oriented programming languages, like Smalltalk-80 [Goldberg & Robson, 1989]. As can be seen in the remainder of this chapter, we have chosen another approach for integrating structure and behaviour modelling. Nevertheless, the notions of a method and a message can be expressed in our framework.

However, this way of describing the dynamic aspects of a system has at least two disadvantages. First, such an approach is still focussed on describing the static properties of a system. Although there are some facilities for describing activities, behaviour modelling is still subsidiary to modelling statics. Second, this notion of objects that execute one or more methods upon receiving a message is not always the best-suited way of modelling a system. A classical example, which may be artificial but which illustrates what we mean, is the following: when we want to compute the sum 5 + 4 in Smalltalk-80, we have to send the message "+ 4" to the object "5". Subsequently, the object "9" is returned.

In order to avoid these disadvantages, we have chosen to develop a model for describing statics and dynamics by integrating an object model and a process model. Thus, we obtain a well-balanced model that is neither focussed on structural nor on behavioural aspects. We developed the SimCon Object Model as the static pillar of our approach. In this chapter, we introduce the integrated model that
is obtained by integrating the SimCon Object Model with a process model for describing dynamics.

The process model that we are going to use has to meet several demands. First, we prefer a model that is as simple as possible and that can be integrated with the SimCon Object Model. We have stated before why we only want to use formal models. Moreover, we would like a model that is intuitive and that is supported by a tool. Finally, we prefer to use a model for which a lot of theoretical results have been accomplished already. Then, these theoretical properties can be interpolated into our approach.

Because of these requirements, we have chosen a Petri net model as the dynamic pillar in our approach. Petri nets combine the benefits of a formal method with the expressive comfort of a model that is intrinsically graphical. The advantages of a formal approach have already been mentioned in Chapter 1. Because of the graphical nature of Petri nets and the fact that they have a mathematical semantics, it has been possible to build good tools for them. Two important ones are Design/CPN and ExSpect. We have used the latter as a basis for our SimCon tool [Van den Broek & Verkoulen, 1992; Van Hee, Rambags & Verkoulen, 1993]. Using such tools, it is relatively easy to develop large Petri nets. These specifications can be used for simulation purposes, as they have a well-defined, unambiguous semantics. Moreover, many analysis techniques have been developed for Petri nets (some important ones are mentioned in Section 5.1). Some of these techniques have already been implemented in Petri net tools. Finally, Petri nets are well-suited for modelling concurrency and distribution aspects.

As one of the "competitors" of our integrated approach, existing high-level Petri net models have been mentioned in Section 1.4. The tools that support these high-level nets provide a functional language like ML [Harper, 1986] for describing the data aspects of a Petri net. However, this way of modelling statics will not always be satisfactory, especially in specifications with a large and complex state space; in this case, it is difficult to survey the relationships between the various token types. Also, it may be difficult to define the expressions that determine the functionality of the transitions. From a Petri net view, you could say that we have solved this problem by adding an object model with its corresponding manipulation language (cf. Chapters 2 and 3). In this chapter, we describe how this integration is obtained.

We start with an informal description in the next section. Afterwards, we deal with the assignment of unique identifications to newly created objects. We show that no additional constructs are needed to guarantee the uniqueness of the identities. This section is followed by a formal definition of our model. We conclude this chapter by describing aspects of the tool that supports our model. More details about how to apply our approach in practice can be found in Chapter 6.
4.2 Informal Description

In this section, we introduce the SimCon Net Model (SCNET) (cf. [Van Hee & Verkoulen, 1991; Van Hee & Verkoulen, 1992; Houben & Verkoulen, 1991a; Van Hee, Rambags & Verkoulen, 1993]). In SCNET, the SimCon Object Model is integrated with a high-level Petri net model.

In fact, the SCNET model is a special kind of Petri net model. For those who are unfamiliar with Petri nets, we give a very short introduction to the most important concepts from Petri net theory. There is a huge amount of literature concerning this research area. A good paper explaining the most important concepts is [Murata, 1989]. This paper also contains a large amount of references. If one is more interested in high-level Petri net models, one should read the work of Jensen regarding Coloured Petri Nets (CPN) or some papers regarding ExSpect.

Basically, a Petri net is a bipartite graph, consisting of transitions and places. Transitions are active; they communicate by sending tokens to each other over the places. As the topology of the net is static and a transition cannot "hold" a token, the state of a Petri net at a certain moment in time is given by the configuration of the tokens in the places. When the input places of a transition contain a specified token configuration, this transition may fire or occur. If it does so, it consumes tokens from its input places and produces tokens for its output places. So, when a transition fires, the state of the net may change.

In so-called Place/Transition Nets [Reisig, 1985], a token does not contain any information besides the fact that it exists. The functionality of a transition is described by stating how many tokens will be generated for which places if that transition fires. As it was observed that these nets do not give a modeller of large systems sufficient expressive comfort, so-called high-level Petri nets have been introduced. Some important examples are Predicate/Transition Nets [Genrich, 1987], CPN and ExSpect. In these models, tokens get a value. To each place, a type is assigned, denoting the kind of tokens that may occur in that place. The behaviour of the transitions is often described by functional expressions. Moreover, the concept of time has been added to some of these models. Finally, most of them allow some kind of hierarchical nets. In CPN, this hierarchy has been included in the formal model. In ExSpect, the hierarchy only occurs in the tool. However, Verkoulen [1989] gives a formal semantics of this hierarchical model in terms of a flat one. Therefore, we will use this flat model for our formal definitions.

The additional facilities that are provided by the ExSpect tool, but that are not part of the formal model, are hierarchy and preconditions (guards). The hierarchy concept is implemented using so-called systems: a system is a subnet and may be used instead of a transition. Preconditions can be used to specify additional restrictions upon the tokens that are consumed by a transition. E.g., we may specify that a transition TripPlanning can only assign a driver to a truck for which he is qualified. Preconditions can be simulated using if-then expressions as an arc-inscription, so we do not incorporate them (yet) in the formal model.
The SCNET model is a combination of the SimCon Object Model and ExSpect. Instead of defining the token structure by means of type definitions, we do this by designing a SimCon object schema and assigning a container type to each place in the net. Thus, the tokens in the net become container objects. As there is only one object schema, it is easy to observe the relationships between the tokens (the containers) that may occur in these places. Moreover, the facilities for describing the static aspects of a system are on the same level of abstraction as the features for constructing net models. The functionality of a transition \( t \) is defined by assigning an SCA expression to \( t \). This expression determines which container \( t \) puts into its output place(s) when it fires. The labels of the containers determine into which place a generated container will be put. Of course, a transition can only generate containers for its output places.

The reader may remember that the CC-operation does not allow for creating an arbitrary large number of container objects at a time. The reason for this is that this would make it possible to define a transition that can produce an arbitrary number of tokens, e.g., it would be possible to define a transition that consumes a set from a place and puts all the elements of this set "in one pass" into an output place. This is not possible using ExSpect and its functional language, so we do not want to make it possible in an SCNET either.

We illustrate the SCNET model by means of a small example. In Section 6.2, we deal with a comprehensive example.

**Example 4.2.1** In this example, we specify a transport company that carries bulked goods.

First, consider the object schema from Figure 4.1. It contains simple types driver, truck and order. An order specifies the volume of goods to be transported; the meaning of the other attributes and references is obvious from their name. For each simple type, there is a singleton container type; these container types are not depicted in Figure 4.1. Moreover, there is a container type trip.

In Figure 4.2, a small SimCon net is depicted\(^1\). The system works as follows. The TripPlanning department schedules a trip upon arrival of an order, provided that a suitable truck and a driver with a licence for that truck are available. If so, the trip is initiated by letting the driver get into his car and leave for the storehouse. After some time, the truck arrives at the storehouse. There, the truck is loaded, according to the order that has been passed by TripPlanning in the meantime. When the truck has been loaded, it leaves for its destination. We do not want to go into detail about this part of the model, so we model it by the blackbox TripExecution. However, we assume that the truck is emptied in TripExecution and that the empty truck arrives at TripControl afterwards. TripControl separates the truck and the driver and puts them into the garage and into the waiting-room, respectively. Moreover, a signal is sent to the administration, which sends an invoice to the client.

\(^1\) The dots \( \bullet \bullet \bullet \) indicate that this net is part of a larger one.
4.2 Informal Description

The transition TripPlanning may only fire when it can "consume" a truck from the garage with sufficient capacity for the order and a driver from the waiting-room that has a licence for that truck. This is formalised by the following precondition:

\[ \text{order[quantity]} \leq \text{truck[capacity]} \wedge \text{truck[driver][licence for driver]} \]

The expressions that define the transitions are trivial.

In this chapter, we only deal with the SCNET model. We do not deal explicitly with the way this model should be applied. This is postponed until Chapter 6. There are many ways in which an SCNET can be interpreted. The two extreme interpretations are the following ones:

(i) Each Petri net place corresponds one-to-one to a (physical) location in the system to be modelled; e.g. a place representing a storehouse or a computer memory. The transitions correspond to active entities in the real world, like machines or CPUs.

(ii) The Petri net represents a kind of life cycle model: the places represent (classes of) possible states of the objects in those places. The transitions are the "object managers" which apply methods onto objects, send messages to each other and move complex objects from one place to another. Chapter 6 deals with this approach.
4. The SimCon Net Model

Figure 4.2: SimCon net for Trip example
4.3 Object Creation

In fact, the first way of applying our model is really an object-oriented way of working, as each object in the net directly represents one entity in the Universe of Discourse. However, the second way of working may be applied by people who tend to think in terms of life cycles and the like; this way of working is often also called object-oriented. In practice, often a mixture of these two approaches will be used.

4.3 Object Creation

This section addresses the assignment of unique identities (names) to objects. All identities in a SimCon net should always be different. This section describes a mathematical structure which can be used for assigning unique identities to newly created objects. Theorem 4.4.5 states that this way of working indeed guarantees identity uniqueness.

Our way of assigning new object identities does not require a central daemon and/or extra memory cells in a specification. This would be very unnatural, because it would not reflect the system as it is in the real world. Moreover, a central daemon does not match with the locality concept of Petri nets. The formal vehicle for making this local assignment of identities possible, is introduced in this section. We demonstrate that this definition is sound.

In Chapter 2, we have seen that a SimCon object schema is defined in the context of an identification set. In this section, we extend this simple set to a so-called identification structure. Such an identification structure consists of a countable set $I$ (an identification set), a set $I_\Lambda \subseteq I$ (the atomic or initial identities) and the functions $F$ and $G$ which are defined upon $I$. The $F$ function assigns to each identity $i$ a set $F(i) \in \mathcal{P}(I)$ of identities which are "children" of $i$. When new objects are created from objects with identities $i_1, \ldots, i_n$, the identities of the new objects are taken from $F(\{i_1, \ldots, i_n\})$. For different $i$ and $j$, the sets of children $F(i)$ and $F(j)$ must be disjoint. This implies that no identity can "survive": an identity $i$ cannot be an element of $F(i)$ as this would violate the aforementioned disjointness constraint (cf. Lemma 4.3.2). This might seem contradictory to the idea of a never-changing unique identification. This is solved by considering one child of a parent as the continuation of its parent. In an identification structure, this is done by defining the continuation function $G$. The identities $i$ and $G(i)$ refer to the same real-world entity, but in different states of the system under consideration. In other words, $i$ and $G(i)$ are the identities of two versions of the same object. That is why we introduce the predicate $\equiv$ (equality modulo $G$). Intuitively, two identities are equal modulo $G$ if one of them is the transitive $G$-continuation of the other. The predicates and operators $\equiv, \cup, \cap$ and $\neq$ can be defined using $\equiv$.

**Definition 4.3.1 (Identification structure)** An identification structure is a tuple $(I, F, G, I_\Lambda)$ where $I$ is a countable set called the identification set, $F$ is called the
child function, \( \mathcal{G} \) is called the continuation function and \( I_A \subseteq I \) is called the initial set. It has to satisfy the following conditions:\(^2\)

1. \( \mathcal{F} \in I \rightarrow \mathcal{P}(I \setminus I_A) \)
2. \( (\forall i, j \in I : i \neq j \Rightarrow \mathcal{F}(i) \cap \mathcal{F}(j) = \emptyset) \)
3. \( (\forall i \in I : (\exists j \in I_A : (\exists n \in \mathbb{N} : i \in \mathcal{F}^n(j)))) \)
4. \( \mathcal{G} \in I \rightarrow I \) such that \( \forall i \in \text{dom}(\mathcal{G}) : \mathcal{G}(i) \in \mathcal{F}(i) \)

Hence if ID is an identification structure with identification set \( I \), initial set \( I_A \) and child function \( \mathcal{F} \), then the elements of \( I_A \) do not occur as \( \mathcal{F} \)-children, all other elements of \( I \) are \( \mathcal{F} \)-descendants of some element in \( I_A \) and two elements of \( I \) have different \( \mathcal{F} \)-children. Obviously, the following transitivity property holds for \( \mathcal{F} \):

\[
(\forall x, y, z \in I : (\forall m, n \in \mathbb{N} : x \in \mathcal{F}^m(y) \land y \in \mathcal{F}^n(z) \Rightarrow x \in \mathcal{F}^{m+n}(z)))
\]

It is easy to prove the following lemma, which states that an identity can never be an \( \mathcal{F} \)-descendant of itself.

**Lemma 4.3.2** Given an identification structure ID with identification set \( I \) as in Definition 4.3.1, it holds that \( \forall i \in I : (\forall n \in \mathbb{N} : n > 0 : i \not\in \mathcal{F}^n(i)) \).

**Proof**

Because of Definition 4.3.1, we know that for all \( i \in I \) it holds that \( i \in \mathcal{F}^m(a) \) for some \( a \in I_A \) and \( m \in \mathbb{N} \).

Let \( a \in I_A \). We will prove by induction on \( m \) that for each \( i \in \mathcal{F}^m(a) \) the assumption \( i \in \mathcal{F}^n(i) \) for some \( n > 0 \) leads to a contradiction.

Let \( i \in \mathcal{F}^n(a) \) and suppose \( i \in \mathcal{F}^n(i) \) for some \( n > 0 \). There are two cases:

1. \( m = 0 \), which means \( i = a \). But then \( a = i \in \mathcal{F}^n(i) \) for some \( n > 0 \) gives that there is some \( Y \in \mathcal{P}(I) \) with \( \{a\} \cup Y \in \text{rng}(\mathcal{F}) \), which is contradictory to clause 1 of Definition 4.3.1.

2. \( m > 0 \), so some \( k \in \mathcal{F}^{m-1}(a) \) and \( j \in \mathcal{F}^{n-1}(i) \) exist with \( i \in \mathcal{F}(k) \land i \in \mathcal{F}(j) \).

There are again two cases:

1. \( j \neq k \), but this contradicts clause 2 of Definition 4.3.1.

\(^2\)For a function \( f \) and \( S \subseteq \text{dom}(f) \), \( f(S) = \{f(s) : s \in S\} \).

\(^3\)We define \( \mathcal{F}^n \) for \( n \in \mathbb{N} \) as follows:

- For all \( n \in \mathbb{N} \), \( \mathcal{F}^n \in \mathcal{P}(I) \rightarrow \mathcal{P}(I) \) is such that
  - \( \forall B \in \mathcal{P}(I) : \mathcal{F}^0(B) = B \land \)
  - \( \forall k \in \mathbb{N} : \mathcal{F}^{k+1}(B) = \{j \in I | (\exists i \in \mathcal{F}^k(B) : j \in \mathcal{F}(i))\} \).

We write \( \mathcal{F}^n(S) \) instead of \( \mathcal{F}^n(\{x\}) \).
2. \( j = k \), but this means \( k \in \mathcal{F}^m(k) \) as \( k \in \mathcal{F}^{m-1}(i) \land i \in \mathcal{F}(k) \), using the transitivity of \( \mathcal{F} \). But as \( k \in \mathcal{F}^{m-1}(a) \), this gives a contradiction using the induction hypothesis.

In the definition below, we introduce the notion of equality modulo \( \mathcal{G} \). We have introduced the \( \mathcal{G} \) function for representing the identities of different versions of one and the same object. However, this would mean that we have to update all references to an object when the identity \( i \) of that object becomes \( \mathcal{G}(i) \). This would mean a global check of all objects in the system, destroying the local state transition paradigm which is intrinsic to a Petri net approach. Therefore, we define the notion of equality modulo \( \mathcal{G} \).

**Definition 4.3.3 (equality modulo \( \mathcal{G} \))** Let \( \mathcal{D} \) be an identification structure with identification set \( I \) and continuation function \( \mathcal{G} \). For \( i, j \in I \), we define
\[
(i \doteq j) \overset{def}{=} (\exists n \in \mathbb{N} : i = \mathcal{G}^n(j)) \lor j = \mathcal{G}^m(i).
\]

We conclude this section with an example of an identification structure. It can be verified easily that this construction satisfies the constraints of Definition 4.3.1.

**Example 4.3.4 (Construction Identification Structure)** We propose the following construction of an identification structure \( \mathcal{D} = (I, \mathcal{F}, \mathcal{G}, I_\lambda) \).

- \( I = \mathbb{N}^+ \), the finite (non-empty) sequences of natural numbers.
- \( I_\lambda = \mathbb{N} \).
- For \( i \in I \), \( \mathcal{F}(i) \subseteq \{ j \mid j \in \mathbb{N} \} \), so new identities are created from old ones by adding a natural number in front of the old identity.
- For \( i \in I \), it holds that \( \mathcal{G}(i) = 0 \cdot i \).
- For \( i, j \in I \) it holds that \( i \doteq j \) iff:
\[
(\exists n \in \mathbb{N} : i = 0^n \cdot j \lor 0^n \cdot i = j).
\]
Informally, this means that all identities which are equal after deleting (some) leading zeros refer to the same real-world entity.

Below, we introduce the concept of a one-generation state. Intuitively, a one-generation state does not contain two different objects with identities \( i \) and \( i' \) where \( i' \) is a direct or indirect \( \mathcal{F} \)-descendant of \( i \). This is necessary in order to keep the identities in a state different, as will be explained after the definition.

**Definition 4.3.5 (One-generation state)** Consider an object schema \( \Sigma \). A state \( \sigma \in S \Sigma \) (cf. Definition 2.3.3) is called a one-generation state iff
\[
(\forall \tau, \tau' \in \sigma : \forall x \in \tau, x' \in \tau' : x \neq x' \Rightarrow \neg(\exists m \in \mathbb{N} : \text{id}.x \in \mathcal{F}^m(\text{id}.x'))) \land \\
\tau \neq \tau' \Rightarrow \neg((\exists n \in \mathbb{N} : \text{id}.\tau \in \mathcal{F}^n(\text{id}.\tau'))) \tag*{\Box}
\]
This means that in a one-generation state the identity of a container cannot be the descendant of the identity of another container in the same state, and analogously for simple objects. As a consequence, all containers generated from containers in a one-generation state will have different identities from the remaining containers in the state, because ID is an identification structure. If a state holds containers x and y and the identity of x would be a descendant of that of y, then in a state transition y could generate another descendant with the same identity as x. In a one-generation state, such an "identification clash" cannot occur. In particular, a one-generation state cannot contain two different containers c₁ and c₂ with the same identifications, because this would mean c₁ ≠ c₂ ∧ id.c₁ = F(id.c₂).

We have shown that there is no theoretical problem with assigning new identities to simple objects and containers when a transition fires and that it is not necessary to introduce extra memory or other model contaminations. However, in a tool supporting our approach, we may choose another, more ad hoc solution.

### 4.4 Formal Definition of the SCNET Model

In this section, we give the formal definition of the SimCon Net Model; this definition contains some informal remarks in order to give the reader some more intuition. After that, the formal semantics of this model are defined in terms of finite state machines. We have chosen to formalise the notion of a SimCon net in one definition. A disadvantage of this choice is that the definition is rather long. However, splitting it into smaller parts would not have improved its comprehensibility.

An SCNET is represented by a tuple \((\Sigma, P, T, A, C, R, M_0)\). The R-component (the reaction function) needs some explanation. It defines which container objects are generated by a transition t when it fires, depending upon the container objects that are consumed by t. So, this R-component is the formal representation of the firing rule which governs the creation, modification and deletion of container objects.

**Definition 4.4.1 (SimCon net)**. Let ID be an identification structure. We define a **SimCon net** to be a tuple \((\Sigma, P, T, A, C, R, M_0)\), where

- \(\Sigma\) is a SimCon object schema (cf. Definition 2.3.6);
- \(P\) is the set of places. It holds that \(P = \text{LN}\);
- \(T\) is the set of transitions. The sets of places and transitions must be disjoint: \(P \cap T = \emptyset\);
- \(A\) is the bag\(^4\) of arcs: \(A \in B((P \times T) \cup (T \times P))\);
- \(C\) is a function from \(P\) into \(CT\) (\(C \in P \rightarrow CT\)), assigning to each place \(p\) the container type denoting which containers can be contained by \(p\).

\(^4\)\(B(S)\) denotes the bags containing elements of \(S\). More details about this bag notation are given by Jensen [1992a] and also by Van der Aalst [1992].
• $R$ is a set of functions, one for each transition in $T$. For $e \in \mathcal{P}(C_D)$ and for $t \in T$, $R_t : \mathcal{P}(C_D) \rightarrow \mathcal{P}(C_D)$ is such that

$\text{dom}(R_t) = \{ e \in \mathcal{P}(C_D) \mid (\forall p \in P : \#(\tau \in e : \text{l.c.} \tau = p) = \#(p, t) \in A) \}$

Intuitively, this means that for each transition $t$ and for each place $p$, the number of containers in $e$ with place identification $p$ is equal to the number of input arcs from $p$ to $t$. This is the so-called firing condition, which has to be satisfied for a transition to be enabled.

• $(\forall \tau \in R_t(e)) : \text{l.c.} \tau \in \{ p \mid (t, p) \in A\}$.

Intuitively, this means that only containers are generated for output places of $t$.

• $(\forall \tau \in e : (\forall \tau' \in R_t(e) : \text{at.} \tau' > \text{at.} \tau))$

This means that the time stamps of the generated objects are equal to or larger than the largest time stamp of all consumed ones.

• $(\bigcup_{e \in R_t(e)} \text{id.} \tau) \subseteq (\bigcup_{e \in e} \mathcal{F}(\text{id.} \tau))$

This means that the way identities are assigned to generated containers satisfies the conditions of Definition 4.3.1.

• $(\forall \tau \in R_t(e)) : (\forall \sigma \in \mathcal{S}_E : \text{id.} \sigma \in \mathcal{F}(\text{id.} (\bigcup_{e \in e} x)))$

This means that the way identities are assigned to generated simple objects satisfies the conditions of Definition 4.3.1.

• $(\forall \sigma \in R_t(e)) : (\forall s \in \mathcal{S}_c : (\forall t \in 1 : t \notin \text{ref.} s) : (\exists \sigma \in \bigcup_{e \in e} \mathcal{S}_E : \text{id.} \sigma \equiv 1) \lor (\exists \sigma \in \bigcup_{e \in e} \mathcal{S}_E : t \notin \text{ref.} \sigma) \lor (\exists \sigma \in \bigcup_{R_t(e)} \mathcal{S}_E : \text{id.} \sigma \equiv 1))$

Intuitively, this means that each simple object $s$ in a generated container $c$ can only refer to some simple object $x$ with identity $t$ if $x$ was involved in the production of $c$, if $x$ is referred to by some other object involved in the production of $c$ or if $x$ is produced together with $c$.

• $M_0$ is a set of container objects: the initial marking of the net: $M_0 \in \mathcal{S}_E$.

The identity of an object in the initial marking has to be atomic.

It has to hold in a state of such a net that each place contains only objects that are element of the container type of that place. Formally:

$(\forall \sigma \in \mathcal{S}_E : (\forall \tau \in \sigma : \text{ct.} \tau = C(\text{l.c.} \tau)))$

This formal model is not hierarchical. However, it has been recognised that the hierarchy concept adds much expressive comfort to a Petri net model. This hierarchy construct can be used to hide details of a specification at the upper level. The ExSpect tool, which we have extended, does allow for hierarchical Petri nets. In [Verkoulen, 1989], we have shown already that a flat model like that of Definition 4.4.1 can be used to express the formal semantics of the net model upon which the ExSpect tool has been based. As we are not interested in this monograph.
in all kinds of theoretical aspects of this hierarchy concept, we did not include it into the formal semantics (yet).

Regarding the expressive power of our model, we mention a result of [Houben & Verkoulen, 1992a]. There, we have shown that an SCNET can be constructed that simulates a Turing Machine [Lewis & Papadimitriou, 1981].

In the following definition, we give some other auxiliary notions. We need them to be able to express the formal semantics of the model from Definition 4.4.1 in terms of finite state machines. Given some SCNET, its semantics are given by its transition relation \( \text{Tr} \): the transition relation determines which sequences of states are possible for some SCNET. The transition relation is defined in terms of the transition function: given a state and an event, the transition function determines the next state of the net. These concepts are defined below.

**Definition 4.4.2** Let \( \langle \Sigma, P, T, A, C, R, M_0 \rangle \) be a SimCon net. We define

- the event set \( E := \{ e \in T \rightarrow \mathcal{P}(C_E) \mid \text{dom}(e) \neq \emptyset \wedge (\forall t \in \text{dom}(e) : e_t \in \text{dom}(R_e)) \wedge (\forall t \in \text{dom}(e) : (\forall t' \in \text{dom}(e) : t \neq t' \rightarrow e_t \cap e_{t'} = \emptyset)\} \};

- for \( e \in E \), \( i_e := \bigcup_{t \in \text{dom}(e)} e_t \), the input set of \( e \);

- for \( e \in E \), \( \tau_e := \max\{\{at. \tau \mid \tau \in i_e\}\} \), the event time of \( e \);

- for \( s \in SS_T \), \( \text{tt}(s) := \min\{\{u \in TP \mid (\exists e \in E : i_e \subseteq s \wedge \tau_e = u)\}\} \), the transition time of \( s \);

- the transition function \( \text{Tr} \) such that
  - \( \text{dom}(\text{Tr}) = \{ (s, e) \in SS_T \times E \mid i_e \subseteq s \wedge \tau_e = \text{tt}(e) \} \);
  - \( \text{Tr}(s, e) = (s \setminus i_e) \cup (\bigcup_{t \in \text{dom}(e)} R_e(e_t)) \);

- \( \text{Tr} := \{ (s, s') \in SS_T \times SS_T \mid (\exists e \in E : (s, e) \in \text{dom}(\text{Tr}) \wedge s' = \text{Tr}(s, e)) \} \), the transition relation of the SimCon net.

In the remainder of this section, we prove that the use of an identification structure guarantees object identity uniqueness in a SimCon net. We prove a theorem stating that the identities of simple and container objects in a SimCon net will always be different provided that the initial state of the SimCon net satisfies the following condition: the identities of the containers in the state have to be different and there may not be any container with an identity which is the continuation of the identity of another container in the state (cf. Section 4.3).

We prove the aforementioned theorem using the notion of a one-generation state (cf. Definition 4.3.5): Lemma 4.4.4 states that if a SimCon net is in a one-generation state, it will remain in a one-generation state if ID is an identification structure as in Definition 4.3.1. This implies that all identities in the net will remain different. In order to prove Lemma 4.4.4, we need one auxiliary lemma. It is in fact a reformulation of the requirement that for different identities \( i \) and \( j \) the sets of their \( \mathcal{F} \)-descendants (\( \mathcal{F}(i) \) and \( \mathcal{F}(j) \)) are disjoint.
Lemma 4.4.3  Given an identification structure ID with identification set I as in Definition 4.3.1, it holds that

\[(\forall i, j \in I : (\forall X \subseteq I : (i \in \mathcal{F}(i) \land j \in \mathcal{F}(X)) \Rightarrow (i \in X \land \mathcal{F}(i) \subseteq \mathcal{F}(X)))].\]

Proof

Let i, j \in I and X \subseteq I such that j \in \mathcal{F}(i) and j \in \mathcal{F}(X). We prove i \in X by induction to X.

If X = \{x\}, then j \in \mathcal{F}(i) \land j \in \mathcal{F}(X) implies x = i because of clause 2 of Definition 4.3.1, giving i \in X.

If X = \{x\} \cup Y, then there are two possibilities:

1. If x = i, then i \in X holds.
2. If x \neq i, then j \notin \mathcal{F}(x), because j \in \mathcal{F}(x) would contradict clause 2 of Definition 4.3.1. So in this case, j \in \mathcal{F}(Y) holds. But then (using the induction hypothesis) i \in Y holds, so i \in X holds too.

Now that we have proved i \in X, it is trivial that \mathcal{F}(i) \subseteq \mathcal{F}(X) as we have that \mathcal{F}(\bigcup_{x \in X} x) = (\bigcup_{x \in X} \mathcal{F}(x)).

Lemma 4.4.4 (Invariance of Unicity)  Let \(O = (\Sigma, P, T, A, C, R, M_0)\) be a SimConnect and \(\sigma\) a one-generation state of \(O\). Then the following holds:

\[(\forall t \in T : (\forall e \in \text{dom}(R_t) \cap \sigma : (\sigma \setminus e \cup R_t(e)) is a one-generation state)).\]

Proof

Let t \in T and assume that O makes a state transition from a one-generation state \(\sigma \in \mathcal{SS}_\Sigma\) to another state under the firing of t which consumes a set of container objects \(\alpha \in \text{dom}(R_t) \cap \sigma\). We now have to prove that \(\sigma' = (\sigma \setminus \alpha) \cup R_t(\alpha)\) is again a one-generation state. It follows from Definition 4.4.1 that \(\sigma'\) is a state.

Let c, c' \in c'. In order to prove that \(\sigma'\) is a one-generation state, we have to prove the following:

\[c \neq c' \Rightarrow \neg(\exists n \in \mathbb{N} : \text{id.c} \in \mathcal{F}^n(\text{id.c}')).\]

and the same analogously for simple objects. We will give only the proof for container objects here.

Suppose that \(c \neq c'\). Suppose furthermore that \text{id.c} \in \mathcal{F}^m(\text{id.c}') for some \(m \in \mathbb{N}\). We will derive a contradiction now. We do this by distinguishing three cases:

1. Suppose c and c' are both elements of \(\sigma \setminus \alpha\). But in that case, \(\sigma\) is not a one-generation state. This contradicts our earlier assumption.

2. Suppose c and c' are both elements of \(R_t(\alpha)\). In that case, according to Definition 4.4.1, there exist some v, w \in \alpha such that \text{id.v} \in \mathcal{F}(\text{id.v}) and that \text{id.c'} \in \mathcal{F}(\text{id.w}). We have assumed moreover \text{id.c} \in \mathcal{F}^m(\text{id.c'}) = \mathcal{F}(\mathcal{F}^{m-1}(\text{id.c}')) for some \(m \in \mathbb{N}\) with \(m > 0\) (\(m = 0\) would give \text{id.c} = \text{id.c'}, which is contradictory to
our assumption $c \neq c'$ as a consequence of Definitions 2.3.8 and 2.3.9). But using Lemma 4.4.3, it holds then that
\[
id.v \in \mathcal{F}^{m-1}(\text{id}.c'),
\]
but also
\[
id.c' \in \mathcal{F}(\text{id}.w),
\]
so it holds that \(\text{id}.v \in \mathcal{F}^{m}(\text{id}.w)\), using the transitivity of \(\mathcal{F}\).
As \(v \neq w\) holds (which follows from \(\text{id}.v \in \mathcal{F}^{m}(\text{id}.w)\) for some \(m > 0\) and Lemma 4.3.2), this contradicts the assumption that \(\sigma\) is a one-generation state.

3. In the third case one of the containers \(c\) and \(c'\) belongs to \(R_{i}(\alpha)\) and the other one to \(\sigma \setminus \alpha\). We assume that \(\text{id}(c) \in \mathcal{F}^{m}(\text{id}(c'))\) for some \(m \in \mathbb{N}\), so \(c'\) was generated before \(c\), so in this case we have \(c \in R_{i}(\alpha)\) and \(c' \in (\sigma \setminus \alpha)\). Here \(m > 0\) holds for the same reasons as in clause 2 of this proof.
It holds that \(\text{id}(c) \in \mathcal{F}(\text{id}(v))\) for some \(v \in \alpha\), because of Definition 4.4.1.
On the other hand, we have assumed \(\text{id}(c) \in \mathcal{F}^{m}(\text{id}(c')) = \mathcal{F}(\mathcal{F}^{m-1}(\text{id}(c'))), \)
where \(c'\) is an element of \(\sigma \setminus \alpha\). This means that \(v \neq c'\) holds. But, using Lemma 4.4.3, this also implies that \(\text{id}(v) \in \mathcal{F}^{m-1}(\text{id}(c'))\) where \(v \neq c'\) and \(c', v \in \sigma\). But this means that \(\sigma\) is not a one-generation state, which contradicts our earlier assumption.

\[\square\]

**Theorem 4.4.5** Let \(O\) be a SimCon net with initial marking \(M_{0}\). Let \(M_{0}\) be a one-generation state. Then all reachable states of \(O\) are one-generation states.

**Proof**

Let \(\sigma\) be a state of \(O\) which can be reached from \(M_{0}\) in \(n\) steps. We will prove that \(\sigma\) is a one-generation state using induction to \(n\).

1. \(n = 0\): Then \(\sigma = M_{0}\), which is a one-generation state;

2. \(n > 0\): Then \((\sigma', \sigma) \in \mathcal{T}_{r}\) and \(\sigma'\) is reachable from \(M_{0}\) in \(n - 1\) steps. This means that \(\sigma'\) is a one-generation state using the induction hypothesis. Then, according to Lemma 4.4.4, \(\sigma\) is also a one-generation state.

\[\square\]

It might seem that the way we assign identities to objects is quite cumbersome, whereas we could simply have chosen a solution with one or more memory cells recording “the first free number”. However, we do not want to introduce memory cells for this assignment of identities. Moreover, we want to maintain the local state transition paradigm: when a transition fires, the state change which is a consequence of this firing can be determined by removing the consumed containers and adding the generated ones. For this purpose, no global information is needed.
4.5 Tool Support

In this section, we give a short introduction to the tool facilities for making Petri nets. These facilities are taken from the ExSpect tool [Van den Broek & Verkouwen, 1992; Van Hee, Rambags & Verkouwen, 1993]. The tool consists of a graphical editor, a type checker, an interpreter and some analysis tools.

The graphical editor has a standard mouse-driven interface and offers the features that one expects nowadays from such a tool. Elements like places, transitions, and nets can be created and changed. Of course, the graphical symbols of the tool are customisable. Transitions and nets can be installed in other nets; their positions can be changed. It is possible to zoom in and out subnets. The arcs that connect places and transitions can be drawn automatically in such a way that they do not cross other objects. When a subnet is installed within another net, the editor checks the correct “wiring” of the connecting places. Of course, the designer may also create text elements like types and functions, which are used to support the definition of the transitions. Finally, the designer can chose his own symbols for representing (not necessarily all) transitions, places, stores and subnets. The features for designing the SimCon schema that is used to describe the token structure of the net, have been mentioned in Chapter 2. Eventually, it will be possible to assign a container type to a place by clicking on the place and the container type. Also, some places may be defined directly using the ExSpect type system. This is useful for defining control flow places, e.g. for defining synchronisation requirements.

The type checker has as input a source file containing definitions. Type checking is done on a file by file basis. Each module can be checked separately, since the declarations of functions defined in other modules should be explicitly imported. The type checker checks the type correctness of the definitions in a file. We do not want to present the type checking mechanism here; this can be found in [Van Hee, Somers & Voorhoeve, 1988; Verkouwen, 1989]. The type checker also checks the type correctness of the SCA expressions defining the transitions in a net. The algebra does not provide a mechanism to compute new attribute values from existing ones. But this is not necessary, as the functional language of ExSpect can be used for this purpose.

The task of the interpreter is to simulate a specification. The interpreter offers several features in order to support a simulation. An important one is animation. Also, a running specification may communicate with external applications, e.g. a spreadsheet or a program that computes the mean occupation rate of a place.

Finally, there are some analysis tools that can be used to compute P- and T-invariants and some timing properties. These facilities have been introduced by Van der Aalst [1992].
4.6 Conclusion

In this chapter, we have introduced the integrated SimCon model both informally and formally. Some remarks about tool support have been made. Also, a small example that illustrates our approach has been dealt with. A more comprehensive example can be found in Section 6.2.

This concludes the first part of this monograph, in which we have developed our approach. In the remaining two chapters, we deal with some aspects of using our approach.

Chapter 5 deals with automatic verification of SimCon specifications. We provide a mechanism for deciding automatically whether the net part of SimCon specification leaves the constraints from the object model side invariant.

Chapter 6 is involved with applying our approach onto problems in practice. By means of a case study, we illustrate the application of SimCon onto practical problems. Moreover, it provides guidelines about the way SimCon could be used. However, we do not want to prescribe obligatory guidelines about the use of our model.
Chapter 5

Automatic Constraint Verification

In the previous chapters, an integrated model for describing static and dynamic aspects of a system has been described. We have accomplished integration at the level of modelling. In this chapter, we describe integration at the level of verification. We introduce the Prime State Method that can be used for verifying whether a constraint that has been formulated in terms of SCOM, will always be maintained by the corresponding SCNET. Many approaches exist that can be used for verifying properties; some important ones are mentioned below. However, only few methods provide automatic verification. In most cases, the user of the verification method (the designer) has to participate actively. If he has a firm mathematical background, this may not be a problem. Otherwise, it can be too difficult to use such a verification method. In that case, the designer can only use simulation for obtaining some insight in the behaviour of the system he has developed. Obviously, this way one can only obtain confidence that the system satisfies the desired property, but it cannot be guaranteed. As our approach provides a means for automatic verification, it does not suffer from this disadvantage. Of course, the problem of formulating the constraints a designer has in mind, remains non-trivial. Although some constraints may be entered graphically within SimCon, most of them will have to be formulated in (a language that looks like) predicate logic. However, advanced support for constraint design falls outside the scope of this monograph.

There are several kinds of constraints. For the moment, we only consider static constraints. Static constraints impose some conditions which must be satisfied by a state in order to be admissible. We do not cover dynamic constraints (as yet). A dynamic constraint prohibits certain kinds of state transitions. Of course, a certain class of dynamic constraints can be expressed in the definition of the net structure.

---

1We only consider finite SCOM schemas in this chapter. This means that the number of types, attributes and references in the schema must be finite.
and the behaviour of the transitions.

We distinguish three classes of static constraints: process constraints, time constraints and data constraints. Process constraints only involve the dynamic aspects of a system. Properties like (absence of) deadlock, liveness, (absence of) individual starvation and statements about the order in which activities may be pursued are process constraints. There are many techniques for proving process constraints, especially in the fields of Petri nets and process algebra. In Section 5.1, we deal with some of them.

The second class of static constraints consists of time constraints. Examples are deadlines, arrival times and mean times between failures. In the field of Coloured Petri Nets, some interesting results in this field have been accomplished by Van der Aalst [1992]. He has introduced so-called Interval Timed Coloured Petri Nets (ITCPN). Actually, an ITCPN is an ExSpect net where the delays are not specified by a real number but by an interval. In his thesis, he has described some algorithms for determining e.g. the earliest possible arrival time of the \( n^{th} \) token at a certain place in the net. Some of these algorithms can be used directly within the SimCon approach. Another approach that also works with interval timing is that of Berthomieu & Diaz [1991]. In the field of Petri nets, much research has been done also regarding stochastic Petri nets [Ajmone Marsan, Bablo & Conte, 1986]. Although stochastic Petri nets are useful for performance analysis purposes, they cannot be used for constraint verification, as their behaviour is stochastic: one cannot determine the probability of a certain form of behaviour, but one cannot guarantee it.

The third class of constraints is formed by the data constraints. In the context of SimCon, all constraints concerning attributes and references of simple objects and the contents of containers are data constraints. Examples are referential integrity, key constraints and attribute constraints.

Within the SimCon approach, we are especially interested in data constraints. It is obvious that these constraints are the most important ones in the context of an integrated framework, as process constraints do not consider data aspects at all and the same holds for most timing properties. However, data constraints clearly involve both structural and behavioural aspects of a system. There are not many automatic verification techniques that cover data constraints: they are merely suited for proving process or time constraints by leaving out the data aspects. E.g., [Van der Aalst, 1992] describes such an approach. Although this is not the kind of verification that we want to accomplish in this chapter, it may be interesting to integrate our technique with one or more of these process and timing approaches in a future research effort.

In this chapter, we first present some existing techniques for verifying specifications. Afterwards, we present the Prime State Method for verifying data constraints.

---

1 Referential integrity means in this monograph that if an object "thinks" it knows another object \( x \), then \( x \) must exist.
within the SimCon framework. We demonstrate how our approach can be used by means of some examples in Section 5.3. Someone who first wants to see an example illustrating our approach could have a look at this Section 5.3 before studying our approach thoroughly. After Section 5.3, a generalisation of our technique for transactions is illustrated, by means of some examples. This chapter is concluded by summarising our results and suggesting the directions for further research.

5.1 Introduction

Suppose one has designed an integrated SimCon specification. Constraints have been formulated in terms of SCOM. Moreover, an SCNET has been developed. When validating and verifying the design, the designer wants to know whether this net preserves these constraints. In other words, when the net is in a state satisfying the constraint and a state transition occurs, will the resulting state again satisfy the constraints?

The area of formal verification of software systems is relatively new. In this introduction, we deal with some existing verification techniques. Many of these techniques have been developed in the context of Petri net models. Others have been developed in the context of a process algebra formalism. We also deal with theorem proving techniques. Problem with most of these techniques is that they do not deal with automatic verification. Other techniques are automatic indeed, yet not concerned with the kind of analysis that we are interested in or they are not sound and complete.

Many approaches for proving properties involve a state space search. Problem with such approaches is the so-called state space explosion: the state space grows exponentially (or even worse!) in the size of the system to be examined. Techniques are being developed that reduce the state space drastically under preservation of the relevant properties of the system. An example of such a method can be found in [Huber, Jensen, Jepsen & Jensen, 1986]. A reduction has to be such that it is sound and complete with respect to the properties that we are interested in. If it is sound, it means that all (relevant) properties that can be derived in the reduced state space also hold in the original one. The completeness means that all properties that hold in the original reachability tree can also be derived in the reduced one. Our approach is in fact such a kind of state space reduction technique. Before introducing our approach, we first mention some (more or less) related research.

Schneider [1990] describes TCSP (Timed CSP), which is an extension of the process algebra CSP [Hoare, 1985]. Some decomposition rules are given for reducing proof obligations. However, this paper focusses merely on liveness and deadlock properties and it misses the aspect of automatic verification.

In [Mauw, 1991], the PSF tool is described. PSF is also based on process
algebra. Mauw [1991] pays attention to the verification of properties using his tool. In specifications without datatypes, the tool can be used to prove bisimulation-equivalence. This means that it can be determined automatically whether two PSF specifications have the same behaviour. This way, it can be proved that the system that has been developed is equivalent (in a sense) to a system with known properties. If this is not the case, a clue is given that indicates what is wrong. As this applies only for specifications without datatypes, this can only be used for verifying process constraints. For checking data constraints (in systems with datatypes), PSF only provides a simulator: such constraints can only be validated, not verified.

A similar approach is described by Lynch & Vaandrager [1991]. They also describe techniques for proving that some automaton has, in some sense, the same behaviour as another automaton of which we know that it has a certain desired property. However, [Lynch & Vaandrager, 1991] does not provide automatic verification.

In [Clarke, Emerson & Sistla, 1986], one provides a mechanism that can be used to prove a temporal logic formula for a finite-state structure. This is a very useful approach, although it only allows for verifying finite systems, i.e. systems with a finite state space. It is especially suited for proving properties of protocols, like the Alternating Bit Protocol (ABP).

In the Petri net literature, much research dealing with the verification of properties has been described. For classical Petri nets, linear algebraic techniques have been developed for deriving Place- and Transition-invariants (P- and T-invariants) [Lautenbach, 1987]. A P-invariant states that the weighted sum of the number of tokens in certain places in the net does not change. A T-invariant gives a firing sequence for the transitions in the net which restores the state in which it is started. These techniques have been generalised for high-level net models [Jensen, 1992a]. Although the invariant techniques are interesting, they do not satisfy our needs because the designer has to formulate and interpret them himself in order to obtain the most interesting results. It might be argued that specifying properties in a language like predicate logic is not easy either, but in our opinion the latter is still more natural and can be supported better in general than formulating properties in terms of invariants. In high-level nets, it is impossible to compute all invariants and it is very difficult to select the interesting invariants from a basis of invariant vectors. In this case, the designer has to propose an invariant (partially), which can be completed and/or verified by the supporting tool.

The work of Christensen & Petrucci [1992] offers a mechanism to make it more easy to compute the invariants of a Coloured Petri Net, as it allows for modular analysis: invariants that have been computed for subnets can be used to obtain the invariants for the complete net. Although invariant analysis will remain a task for specialists, this certainly improves the usability of invariants within high-level net models.

---

3P-invariants have also been called S-invariants: the German word for 'place' is 'Stelle'.
A Petri net approach applying invariant theory is [Voss, 1987], where Petri nets are used to describe properties of database systems. Voss [1987] applies Petri nets for describing process constraints. He also introduces fact transitions or facts: transitions that should never be enabled. P- and T-invariants are used for proving that some transition is indeed a fact. It is demonstrated how facts can be expressed by positive nets, so they are merely an abbreviation instead of a modelling construct that adds expressive power. A disadvantage of this approach is that only manual analysis has been dealt with. By means of examples, the reader is shown how to apply these facts for proving properties, but there is no means of automatically verifying constraints.

Sibertin-Blanc [1985] uses P-invariants to prove properties of a specification that has been designed using his Petri net model. He formalises some plain invariants for proving properties that only involve the net structure. He states that data invariants, where the values of the tokens also are taken into account, are too difficult to deal with in a systematic way. Therefore, he only provides an example of how data invariants could be used in practice.

Another Petri net related approach is that of Tuomiesi [1989]. To be precise, he uses Elementary Net (EN) systems. This is a class of classical Petri nets, with unstructured, valueless tokens. Such an EN-system is translated into Deterministic Propositional Dynamic Logic (DPDL). Proving a property \( \phi \) can then be done by checking whether the corresponding DPDL description \( \Gamma \) implies this property. Generally, it is difficult to prove \( \Gamma \models \phi \), but proving \( \Gamma \nvdash \neg \phi \) often is feasible. This check can be performed using a theorem prover. A well-known theorem prover is OTTER; an example of how OTTER can be used to prove properties can be found in [McCune & Wos, 1987]. As is stated by Plaisted [1990], it appears that theorem provers can be respectable aids to mathematicians, although most of the significant results still require careful user guidance.

In [Leih, 1990], so-called Parallel Object-based Transition systems (POTs) have been introduced. Such a POT is a classical Petri net, extended with some functions. Each place represents the occurrence of an object, the transitions represent events. Functions are defined for describing which objects are represented by which places, which objects (places) are unborn, active and dead, and which objects are known to each active object. Each POT is safe, i.e. a place always contains at most one token. The author describes conditions which must hold for a POT to satisfy certain constraints. An example is the property NODANGL, which means that referential integrity holds: if an object knows another object, the latter one has to be active. However, the conditions that are described for NODANGL to hold are much too strong: new references can only be created via reference passing, which means that references cannot "appear out of thin air" and objects that may be known by other objects may never die. Obviously, these conditions are sufficient, but they are so strong that they exclude many interesting systems.

In [Leveson & Stolay, 1987], the application of so-called Time Petri Nets for modelling and analysing real-time systems is explored. Time Petri Nets are classical Petri nets as defined in [Reisig, 1985], extended with the concept of time.
Furthermore, the authors describe procedures which allow analysis of safety, recoverability and fault-tolerance. Again, automatic verification is not mentioned.

Menasche [1985] describes the PARBED tool for design and analysis of Petri nets. It claims that analysis facilities have been implemented partially in this tool. However, it seems that this tool can only answer classical questions like boundedness and safeness [Murata, 1989]. The version of the tool as described in [Menasche, 1985] can by no means be used for automatic verification of arbitrary properties.

In [Huber, Jensen, Jepsen & Jensen, 1986], the reachability tree is used for proving properties of a Petri net. The nodes of the reachability tree of a system are formed by the states the system can reach. The initial state of the system constitutes the root of the tree. The edges in the tree are determined by the possible state transitions. By studying the reachability tree, many interesting properties of a system can be deduced. However, most reachability techniques suffer from the state space explosion problem: even for relatively small systems, the reachability tree becomes very large (or even infinite). Huber, Jensen, Jepsen & Jensen [1986] deal with the state space explosion problem by identifying so-called equivalent and/or covering states of which only one is included in the reachability tree.

Another contribution to the field of reducing the reachability tree of a net while preserving its characteristic properties is [Valmari, 1991]. He describes the so-called stubborn set method. The stubborn set method reduces the number of nodes in the reachability tree by taking advantage of the fact that different interleavings of concurrent events have the same net effect.

[West, 1992] also claims that he describes a reachability analysis technique. In his paper, reachability analysis is applied for protocol validation. Because of the state explosion problem, West [1992] proposes the so-called random walk validation. This technique reduces the reachability tree as it considers for each state only one next state. This state is chosen randomly from the possible state transitions. He calls it a reachability technique, but in fact it is merely simulation. This technique is only a means to find errors, but it cannot be used to prove a protocol correct. It is difficult to determine at which moment the analysis can be stopped.

All methods that have been mentioned suffer from one or more objections, which make that they cannot be used for our purposes. That is why we have developed a new technique. Of course, our technique is not superior in general to all aforementioned techniques; there are many cases for which you would not apply our method. But for checking (a certain class of) data constraints (as will be presented below), our technique is well-suited.

In principle, our method is also a reachability technique. The state space explosion problem is solved as follows. First, we check a net by checking each transition separately. This means that we do not regard all possible interleavings of transition occurrences, thus reducing the number of states that has to be checked. However, this does not reduce the state space explosion sufficiently. Therefore, a second reduction is obtained by investigating only so-called prime states. We prove that all
states can be constructed from one or more prime states. Furthermore, we prove a theorem stating that when we want to verify the invariance of a constraint, this can be done by checking all prime states. This theorem does not hold for arbitrary constraints: a constraint has to be compositional. What this means exactly, is explained in the next section. We give a characterisation of the class of compositional constraints and we indicate a subclass that can be recognised syntactically.

The underlying idea of our method, is the following. When a state of a SimCon net changes, this is caused by the occurrence of a transition that executes its SCA expression upon "consuming" some container objects from its input places. This set of container objects comprises a so-called SimCon-instance. The SCA expression which defines the transition is applied onto this SimCon-instance. This way, the net moves to a new state. We show that in order to verify a constraint-preservation property for an arbitrary state, it is necessary and sufficient to check some set of primitive states, which we have called prime (SimCon-)instances or prime states. Each state can be described as some composition of prime states. Proving (or disproving) the constraint locally for each transition gives a proof for a complete net, i.e. this guarantees that no transition in the net will ever violate the constraint in a valid state. The only thing that remains to be done is to check whether the initial state satisfies the constraint. This is rather straightforward.

Thus constraints are checked locally for each transition. However, a constraint could be violated by a transition that disturbs the properties of an object that is not directly involved in its firing. Maybe, it is not even in an input or output place of that transition. E.g., consider a schema with simple types A and B and a reference \( r \) from A to B. Furthermore, suppose that the constraint we want to verify is referential integrity. If a transition \( T \) deletes some B-objects, it may have violated the constraint for some A-object. It is possible that this A-object is not in an input place of \( T \). Our method also covers this situation, as will be demonstrated in Example 5.3.1.

An additional advantage of our approach is the following: when a transition violates a constraint, our method gives a (relatively) small counter-example that illustrates what went wrong. This is useful when one is "debugging" a specification: it is far more easy to understand why a construction is not correct if a small example is given for which it gives an erroneous result.

A final advantage of our technique is its independence of the initial state of the specification to be verified. In a plain reachability approach, the complete reachability graph has to be computed again when the initial state is changed. In our approach, one must only check that the new initial state satisfies the constraints, but the prime states do not have to be regenerated and checked again.

\footnote{We do not cover the so-called RA-operation here. This is not a problem, because a transition defined by means of an RA-operation can be replaced by a chain of transitions and places without using this RA-operation.}
The work described in this chapter is rather recent, so it is not completely mature yet. This means that there are some things that have to be improved and a number of things that should be worked out in a further research effort. We mention some important ones below.

First, our approach covers so-called compositional constraints. Whether or not a constraint is compositional is decided on a rather ad hoc basis. As the reader will see, we have described a certain subclass of compositional constraints which can be recognised automatically. This work should be extended. The least we can do is to provide some tool support for deciding whether or not a constraint is compositional.

Also, we have not yet paid sufficient attention to the time complexity of our method. We have started investigations in this direction (about which something is said in the remainder of this chapter), but this is not sufficient yet.

In the remainder of this chapter, we will encounter a number of examples demonstrating how our approach can be applied in practise. Investigation of more realistic (both in size and in complexity) case studies should prove that the work described in this chapter can be applied in practical situations. As the first results look promising, this should only be a matter of time.

In Section 5.4, we deal with an extension of this idea. Often, a system executes transactions, i.e. all states in a certain class have to satisfy some constraint, but in the “intermediate states” this constraint does not have to hold. E.g., when raw materials are delivered at a factory, the administration of that factory will not be up-to-date immediately, but it will take some time (and effort) to achieve this. During this process, the constraint that the administration of the company should represent the actual state of the company, does not have to hold. However, in all other states, we do want our company to satisfy this constraint.

5.2 The Prime State Method

Our technique for automatic constraint verification is a kind of reachability technique. In general, reachability techniques suffer from the state space explosion problem, as has been mentioned in the introduction to this chapter. We solve this problem as follows. By investigating each transition separately in order to check whether it may violate the constraint\(^5\), we reduce the number of states to be investigated. However, this does not reduce the number of states to be checked sufficiently. Therefore, we perform a second reduction step. The general idea is as follows: when an SCA expression without the RA-operation is applied onto a state, this state is changed locally for each simple object that satisfies the simple object selector(s) in the expression. Such a local change may disturb the constraint. For a certain class of constraints, it is necessary and sufficient to investigate some set

\(^5\)Sometimes, a constraint may be violated by one single transition but this does not happen because some other transition(s) compensate(s) this. This situation is covered in Section 5.4.
of prime states in order to check whether the constraint will be satisfied in an arbitrary state. We call these constraints compositional. Intuitively, a compositional constraint satisfies the following property for each state $s$: if $s$ can be composed of a set of prime states that all satisfy the constraint, then $s$ also satisfies the constraint. We only allow expressions without a 'free' occurrence of the RA-operation (the so-called non-recursive expressions), in order to keep this number of prime states finite.

In the remainder of this section, the ideas behind the Prime State Method are described. Formal definitions are explained and illustrated by means of examples. After some auxiliary definitions, we describe an algorithm that can be used to generate the set of prime states in the context of a non-recursive SCA expression and a compositional constraint. We prove two theorems, stating that this algorithm terminates correctly. Finally, we prove the main theorem of this chapter. This theorem states that checking all prime states is necessary and sufficient for checking a compositional constraint. As we prove that the number of prime states is always finite (and not too large, as we will see), this gives a basis for automatic constraint verification that is supported by a tool. Such a tool must generate the prime states and check whether the constraint is violated in these states in order to verify the constraint. The tool must also check whether the constraint holds in the initial state of the net.

**Definition 5.2.1 (non-recursive SCA expression)** A non-recursive SCA expression is an SCA expression without a free occurrence of the RA-operation. However, the RA-operation may occur in such an expression.

In order to be able to formalise the notion of the composition of an instance, we first define what it means that a SimCon-instance $I$ is the restriction of another SimCon-instance $I'$. Informally, $I$ is a restriction of $I'$ if the graph representing $I$ can be embedded in the graph representing $I'$. The following example illustrates this notion.

**Example 5.2.2 (restriction)** Consider a simple schema that contains simple types A and B. There is a reference $r$ from A to B and A has an attribute $x$ of type integer. We are only concerned with simple types in this example; container types are not taken into account. In Figure 5.1, two instances according to this schema are depicted. The instance on the lefthand in this figure is a restriction of the instance on the righthand: it is obvious that the instance on the righthand contains the instance on the lefthand (the boldface part) and some extension (the dashed part).

**Definition 5.2.3 (restriction of an instance)** Let $I$ and $I'$ be two SimCon-instances according to a schema $S$. We say that the function $\rho \in \mathcal{N}(I) \rightarrow \mathcal{N}(I')$ is a restriction of the set of simple objects $\mathcal{N}(I)$ in the set of simple objects $\mathcal{N}(I')$ if
(i) \( \text{dom}(\rho) = \mathcal{N}(I) \): \( \rho \) is total;
(ii) \((\forall x, y \in \mathcal{N}(I) : x \neq y \Rightarrow \rho(x) \neq \rho(y))\): \( \rho \) is injective;
(iii) \((\forall x \in \mathcal{N}(I) : \text{st}(x) = \text{st}(\rho(x)))\): \( \rho \) preserves object types: it maps each simple object \( x \) onto an object that belongs to the same type as \( x \);
(iv) \((\forall x \in \mathcal{N}(I) : \text{av}(x) = \text{av}(\rho(x)))\): \( \rho \) preserves attribute values;
(v) \((\forall x \in \mathcal{N}(I) : \forall r \in \text{dom}(\text{rf}(x)) : \{\text{id}(\rho(y)) | \text{id}(y) \in \text{rf}(x).r\} \subseteq \text{rf}(\rho(x)).r\}): the set of objects obtained by applying \( \rho \) onto all objects to which some \( x \in \mathcal{N}(I) \) refers, is a subset of the objects to which \( \rho(x) \) refers.

If such a function \( \rho \) exists, we say that \( I \) is a restriction of \( I' \) (according to \( \rho \)) or that \( I \) can be embedded in \( I' \) (according to \( \rho \)).

We generalise a restriction function \( \rho \) for instances: the result of applying a restriction \( \rho \) onto an instance \( I \) is obtained by applying \( \rho \) onto the objects in \( I \) and assemble the results. Such a restriction must satisfy an additional requirement:

(vi) \((\forall x, y \in \mathcal{N}(I) : (\exists c \in \mathcal{C}(I) : x \in cs.c \land y \in cs.c) \Leftrightarrow (\exists c' \in \mathcal{C}(I') : \rho(x) \in cs.c' \land \rho(y) \in cs.c'))\):

If \( x \) and \( y \) are in the same container in \( I \), then \( \rho(x) \) and \( \rho(y) \) are in the same container in \( I' \) and vice versa.

The set of all restrictions from \( I \) in \( I' \) is denoted by \( I \Rightarrow I' \).

\( \square \)

As a restriction \( \rho \) is injective and total, we also use the notion of the inverse \( \rho^{-1} \) of a restriction. The domain of \( \rho^{-1} \) is equal to the range of \( \rho \).

Below, we define what it means that a SimCon-instance \( I \) is the composition of SimCon-instances \( I_1, \ldots, I_n \). This composition must be such that application of some non-recursive SCA expression \( \mathcal{E} \) onto this instance gives in some sense the same result as the composition of the results of applying \( \mathcal{E} \) onto those component instances. Later, it will be illustrated why we need this non-trivial definition of
the composition of an instance, instead of just composing an instance by putting
the nodes of the composing instances together.

Definition 5.2.4 (composition of an instance) Let I be a SimCon-instance, C a
constraint and \( E \) a non-recursive SCA expression. We say that I is the composition
of the instances \( I_1, \ldots, I_n \), modulo C and E if

(i) \( (\forall j : 1 \leq j \leq n : (\exists \rho_j \in I_j \sim I)) : \) for each \( j \), there is a restriction of \( I_j \) in \( I \);
(ii) \( C(I) \Rightarrow (\forall j : 1 \leq j \leq n : C(I_j)) : \) if \( C \) holds in \( I \), it holds in each \( I_j \);
(iii) \( (\forall x \in \mathcal{N}(I) : x \in \bigcup_{j=1}^n \text{rng}(\rho_j)) \) (with restrictions \( \rho_j \) as in (i)) : for each simple
object in \( I \), there is at least one corresponding simple object in one of the \( I_j \);
(iv) \( (\forall x \in \mathcal{N}(I) : (\forall r \in \text{dom}(rf.x) : \)
\[ rf.x.r = \bigcup_{j=1}^n \bigcup_{i \in I_j} \{ (i, \rho_j(y)) \mid (x \in \text{dom}(\rho_j^{-1})) \wedge (i, y \in \text{rf.}\rho_j^{-1}(x), r)) \} \] (with restrictions \( \rho_j \) as in (i)) : for each reference in \( I \), there is at least one
corresponding reference in one of the \( I_j \);
(v) the restrictions \( \rho_j \) (as in (i)) can be extended according to Definition 5.2.3 in
order to cover newly created objects by an SC-operation in \( E \); we denote these
extended restrictions by \( \tilde{\rho}_j \);
(vi) \( (\forall j : 1 \leq j \leq n : E(I_j) = \tilde{\rho}_j^{-1}(E(I))) \) (with restrictions \( \rho_j \) as in (i)) : the order
in which \( E \) and the \( \tilde{\rho}_j \) are applied, is not relevant.

In that case, we denote \( I = I_1 \uplus \ldots \uplus I_n \) and \( E(I) = E(I_1) \uplus \ldots \uplus E(I_n) \). Also, we
denote this by \( I = \{ I_j \mid j \in J \} \), with \( J \) a set of indices, e.g. \( J = \{ 1, \ldots, n \} \).

Definition 5.2.5 (minimal composition and prime instance) We call a composi-
tion \( \{ I_j \mid j \in J \} \) of an instance \( I \) in the context of a non-recursive operation \( E \) and
a compositional constraint \( C \) minimal if each deletion of one of the components \( I_j \)
disturbs the composition. Formally: \( I = \{ I_j \mid j \in J \} \) and for each \( K \) that is a real
subset of \( J \) it is not possible to compose \( I \) from the instances \( I_k \) with \( k \in K \).

We call a SimCon-instance \( I \) minimal if each minimal composition of \( I \) does not
contain equal sub-instances (cf. Definition 3.2.2).

We say that a SimCon-instance is prime if there is no other minimal composition
of \( I \) than \( I \) itself (in the context of \( C \) and \( E \)).

From the previous definitions, it follows directly that, in the context of a non-
recursive operation \( E \) and a constraint \( C \), each SimCon-instance can be composed
of prime instances.

We make a notational remark: if the value of \( n \) is known from the context or we
do not want to make it explicit, we will sometimes omit it. Moreover, we simply
say that "\( I \) is the composition of \( I_1, \ldots, I_n \)" if \( E \) and \( C \) are known from the context.
In the definition of a prime instance $I$, it is demanded that no other minimal composition of $I$ exists than $I$ itself. If we would not have demanded this minimality, no instance would be prime because each instance $I$ is the composition of $I$ and $I$.

We need the aforementioned non-trivial definition of the composition of an instance, instead of just composing an instance by putting the nodes of the composing instances together. The latter way of "composing" would cause the number of prime states to be infinite in some cases. E.g., consider the schema of Example 5.2.2. If we would use the union as composition operator, it would not always be possible to decompose (in a non-trivial way) an instance consisting of one $A$-object and an arbitrary number of $B$-objects to which this $A$ refers (cf. Figure 5.2). So, there would be no upper bound for the number of nodes in a prime state, implying that the number of prime states would be infinite.

![Diagram](image)

Figure 5.2: An instance that cannot be decomposed using "union"-composition

**Example 5.2.6 (composition)** Let $\mathcal{E}$ be the operation that deletes all $B$-objects to which an $A$-object with $x = 3$ refers and then deletes all these $A$-objects. Formally, $\mathcal{E} = SD(SD(\text{FN}_{A}[x = 3]), A[x = 3])$. Let $C$ be the constraint referential integrity. Now consider Figure 5.3. On the right-hand, we see a SimCon-instance $I$ that is composed of the two SimCon-instances $I_1$ and $I_2$ on the left. Below the dotted line, we see the effect of $\mathcal{E}$, both on the composing instances and on the composed instance.

It holds that $I$ is the composition modulo $C$ and $\mathcal{E}$ of $I_1$ and $I_2$ as

(i) There exist restrictions $p_i \in \mathcal{N}(I_i) \rightarrow \mathcal{N}(I)$ (cf. Example 5.2.2).
(ii) $C$ holds in $I$, $I_1$ and $I_2$.
(iii) It can be verified that for each simple object $x \in \mathcal{N}(I)$ there is a corresponding object $y$ in at least one of the $I_i$ ($i = 1, 2$) which is mapped onto $x$ by the aforementioned $p_i$.
(iv) It can be verified that for each reference in $I$ there is at least one corresponding reference in one of the $I_i$.
(v) The operation $\mathcal{E}$ does not create new simple objects.
(vi) As can be seen in Figure 5.3 the order in which \( \mathcal{E} \) and the aforementioned \( \rho_i \) are applied, is irrelevant: \( \mathcal{E}(I_i) = \rho_i^{-1}(\mathcal{E}(1)) \) for \( i = 1, 2 \).

\[ \begin{align*}
I_1 & \quad \begin{array}{c}
A & \quad r \quad \bullet \quad B \\
\bullet & \quad \bullet
\end{array} \\
I_2 & \quad \begin{array}{c}
A & \quad r \\
\bullet & \quad x = 3
\end{array}
\end{align*} \]

\[ \begin{align*}
\mathcal{E}(I_1) & \quad \begin{array}{c}
A & \quad r \quad \bullet \\
\bullet & \quad B
\end{array} \\
\mathcal{E}(I_2) & \quad \begin{array}{c}
A & \quad r
\end{array}
\end{align*} \]

Figure 5.3: The composition of an instance before and after application of \( \mathcal{E} \)

Consider \( \mathcal{E} \) and \( C \) as described above. The instances \( I_1 \) and \( I_2 \) in Figure 5.3 are prime in the context of \( \mathcal{E} \) and \( C \), whereas \( I \) is not prime.

The next definition introduces the notion of compositional constraints. Intuitively, this means the following. Suppose \( I \) is an arbitrary SimCon-instance. Let \( \mathcal{E} \) be a non-recursive SCA expression. Suppose furthermore that \( I \) can be written as the composition \( \exists I_j \), where all \( I_j \) are prime. Now we say that a constraint \( C \) is compositional if the following holds: if all prime SimCon-instances \( I_j \) satisfy \( C \), then \( C \) also holds in \( I \). E.g., it is clear that the constraint “each \( A \) should have a reference to a \( B \)-object which has an attribute \( x \) with value 3” is compositional: if it holds in some instance, it also holds in each larger instance. However, the constraint “an instance should contain exactly 10 \( A \)-objects” is obviously not compositional.
Definition 5.2.7 (compositional constraints) A constraint $C$ is compositional if for each $\text{SimCon}$-instance $I$, for each non-recursive SCA expression $\mathcal{E}$ and for each composition of $I$ into prime $\text{SimCon}$-instances $\bigcup_{i=1}^{n} I_i$, it holds that

$$(\forall j : 1 \leq j \leq n : C(I_j)) \Rightarrow C(I)$$

\[\Box\]

In the remainder of this chapter, we will use Definition 5.2.4 only in the context of compositional constraints.

It would be interesting to generalise Definition 5.2.7 by giving it in the context of one fixed non-recursive operation $\mathcal{E}$. This would increase the number of situations in which the Prime State Method can be used, but it also makes the method more ad-hoc, making a general classification of compositional constraints fairly impossible.

Techniques for optimising databases often use so-called dependency constraints [Furedaens, De Bra, Gyssens & Van Gucht, 1988]. An example of such a constraint is the functional dependency constraint, which prohibits two different objects of one and the same type to have the same references and/or attributes, regarding a specified set of reference- and attribute-names. E.g., two different persons may not have the same name and address. These constraints are not compositional, as the composition of two different objects with the same attributes and references may give an instance that does not satisfy the constraint, whereas it is satisfied in each composing subinstance. However, one could argue that this is not a severe problem, as our approach will be used in the context of a high-level specification framework. We stated already in Chapter 1 that our approach aims at the analysis and design phase of IS development. Dependency constraints and the like are particularly important in the realisation phase. Possibly, this problem can be solved (partially) by formulating precise requirements for an SCA expression to satisfy such a dependency constraint or even by transforming expressions (automatically) into expressions that do not violate the constraint.

Of course, it would be useful when a more general classification of compositional constraints could be given. A syntactic check would be preferable, as this can be pursued fairly fast. However, we think it is not likely that such a characterisation can be given for the complete class of compositional constraints. Below, we give a characterisation of a certain class of constraints, the so-called existential constraints. We prove that they are compositional. This illustrates how classes of compositional constraints can be classified syntactically.

Definition 5.2.8 (existential constraints) Let $\Sigma$ be an object schema, $X_1, \ldots, X_n$ simple and container types defined in $\Sigma$. 

A constraint of the form \((\exists x_1 \in X_1 : \ldots (\exists x_n \in X_n : C(x_1, \ldots, x_n)))\ldots\) is called existential, where \(C\) is a proposition that only holds positive information (without negation) about references, attributes and contents of \(x_1, \ldots, x_n\), i.e. it may contain conjunction and disjunction symbols and atomic formulae (formulae without quantors, conjunction and disjunction symbols, that may contain functions ranging over \(x_1, \ldots, x_n\)), stating properties regarding the contents, attributes and references of \(x_1, \ldots, x_n\).

\[\square\]

**Lemma 5.2.9 (existential constraints compositional)** Let \(\Sigma\) be an object schema and \(C\) an existential constraint in the context of \(\Sigma\). Then \(C\) is compositional.

**Proof**

Existential constraints demand that an instance contains certain objects that have certain relationships with each other. If such a constraint holds in an instance, it also holds in each larger instance, as the existence of objects that have to satisfy some positive constraints can never be violated by adding objects to an instance. So, if all instances \(I_i\) satisfy the constraint, the composition of these \(I_i\) also satisfies the constraint, as it contains all composing instances.

For universal quantifiers, such a result is generally not possible. E.g., the constraint that each \(A\)-object should have a \(r\)-reference to each \(B\)-object is not compositional, as the composition of the two instances \(I_1\) and \(I_2\) in Figure 5.4 satisfy this constraint, but the composition \(I\) of \(I_1\) and \(I_2\) in the same figure does not.

![Figure 5.4: Three SimCon-instances \(I_1\), \(I_2\) and \(I\)](image)

Below, we use the notion of two simple objects being equal modulo \(C\) and \(\mathcal{E}\).

**Definition 5.2.10 (equality modulo \(C\) and \(\mathcal{E}\))** Let \(C\) be a compositional constraint and \(\mathcal{E}\) a non-recursive SCA expression. Two simple objects \(x\) and \(y\) are equal modulo \(C\) and \(\mathcal{E}\) in an instance \(I\) if
for each simple object selector \( s \) that occurs in \( \mathcal{E} \), \( x \) is selected in \( I \) by \( s \) iff \( y \) is selected in \( I \) by \( s \): \( x \in \mathcal{O}[s]_I \iff y \in \mathcal{O}[s]_I \).

- for each subformula \( f \) of \( \mathcal{C} \), \( x \) satisfies \( f \) iff \( y \) satisfies \( f \): \( [f(x)]_I \iff [f(y)]_I \).

\[\Box\]

**Example 5.2.11 (equality modulo \( \mathcal{C} \) and \( \mathcal{E} \))** Let \( \mathcal{C} \) be the constraint demanding that each A-object has an \( x \)-attribute with value 3: \( (\forall a \in A : a.v.a.x = 3) \). Moreover, consider the operation \( \mathcal{E} = \text{SD}(1_B [y = 4]) \). Consider the instance in Figure 5.5. It contains three A-objects and three B-objects.

Figure 5.5: An instance with six simple objects

\( B_1 \) and \( B_3 \) are equal modulo \( \mathcal{E} \) and \( \mathcal{C} \). \( B_2 \) differs from the other two, because of its \( y \)-attribute. \( A_1 \) and \( A_2 \) differ modulo \( \mathcal{E} \) and \( \mathcal{C} \) because of their \( x \)-attribute. \( A_1 \) and \( A_3 \) differ modulo \( \mathcal{E} \) and \( \mathcal{C} \) because of their \( x \)-attribute. \( A_2 \) and \( A_3 \) are equal modulo \( \mathcal{E} \) and \( \mathcal{C} \).

Before being able to describe the algorithm for determining the set of prime SimCon-instances in the context of a schema \( \Sigma \), a compositional constraint \( \mathcal{C} \) and a non-recursive SCA expression \( \mathcal{E} \), we introduce one auxiliary notion, viz. the impact of a simple SCA expression. Therefore, we also need the notion of the size of a simple object selector and the size of a compositional constraint.

**Definition 5.2.12 (size of a selector)** The size of a simple object selector \( sos \), denoted by \( \sigma(sos) \), is defined as 1 plus the number of occurrences of reference and attribute names in \( sos \).

The size of a value selector \( vs \), denoted by \( \sigma(vs) \), is defined as 1 plus the number of occurrences of reference and attribute names in \( vs \). \[\Box\]

We illustrate this by means of an example: \( \sigma([a = \zeta x [b], \zeta' y []]) = 5 \).

**Definition 5.2.13 (size of a compositional constraint)** Let \( \mathcal{C} \) be a compositional constraint. The size \( \sigma \) of \( \mathcal{C} \) is defined as the sum of:
5.2 The Prime State Method

- the number of type names in C;
- the number of occurrences of a reference name in C;
- the number of occurrences of an attribute name in C.

\[ \sigma((\forall x_1 \in T_1 : (\forall x_2 \in T_2 : av.x_1.a = av.(rf.x_1.r_1).a' \land av.x_2.a' = 0))) \]

So, \( \sigma((\forall x_1 \in T_1 : (\forall x_2 \in T_2 : av.x_1.a = av.(rf.x_1.r_1).a' \land av.x_2.a' = 0))) \) is \( 2 + 1 + 3 = 6 \).

**Definition 5.2.14 (impact)** Let \( \mathcal{E} \) be a non-recursive SCA expression and \( C \) be a compositional constraint. The impact of \( \mathcal{E} \) and \( C \) (denoted by \( \#(\mathcal{E}, C) \)) is the maximum of \( \sigma(C) \) and the sum of the sizes of the argument selectors in \( \mathcal{E} \).

Often, this value for the impact of a non-recursive SCA expression in the context of a compositional constraint will be larger than necessary for obtaining correct answers, e.g. when \( C \) is of the form \( (x = y) \land (x = y) \). This does not disturb the soundness nor the completeness of our method. However, it would be interesting to try to bring this quantity down, as this would reduce the computational costs of our method. A first step would be to normalise the constraints and the expressions.

The correct termination of Algorithm 5.2.15 is proved by the Theorems 5.2.16 and 5.2.17. Finally, Theorem 5.2.18 states that checking these prime SimCon-instances is necessary and sufficient to prove a compositional constraint for an arbitrary non-recursive SCA expression. This is the main theorem of this chapter.

**Algorithm 5.2.15 (construction of prime SimCon-instances)** Let \( \Sigma \) be a SimCon schema, \( \mathcal{E} \) a non-recursive SCA expression in the context of \( \Sigma \) and \( C \) a compositional constraint. The set of prime SimCon-instances in the context of \( \Sigma \), \( \mathcal{E} \) and \( C \) can be constructed as follows.

(i) Create all SimCon-instances with at most \( \#(\mathcal{E}, C) \) nodes:
- The possible values for an attribute \( a \) of such a node \( x \) are all constants in an sos in \( \mathcal{E} \) for the same attribute and one “fresh” value for each attribute name\(^6\).
- The possible references are all references to other objects in the instance and/or to objects outside of this instance, according to \( \Sigma \). For each reference name \( r \), an object may contain at most one \( r \)-reference to an object outside the SimCon-instance (an sos-expression cannot “see” the difference between an object with one \( r \)-reference to an object outside of the instance and an object with more than one such an \( r \)-reference).
- The containers are “filled” with simple objects according to \( \Sigma \).

If all instances with \( \#(\mathcal{E}, C) \) nodes are not prime, then stop. Otherwise, introduce a variable \( n \) denoting the maximum cardinality of the instances that have already been generated (so, \( n = \#(\mathcal{E}, C) \) now) and continue.

---

\(^6\)Note that not all attributes of a simple object have to have a value.
(ii) Create all instances with \( n + 1 \) nodes (with the same restrictions as mentioned in (i)). Stop if none of these instances is prime. Otherwise, increase \( n \) by one and repeat step (ii) with this new value for \( n \).

The set of all prime SimCon-instances induced by \( E \) and \( C \) is denoted by \( P^E_E \). 

Algorithm 5.2.15 generates at least all instances with at most \( \#(E, C) \) nodes, because it may be possible that all instances with \( n' < \#(E, C) \) nodes are not prime because they are not changed by \( E \). In that case, the algorithm might stop too soon, so we might miss some prime instances.

For the construction of prime SimCon-instances, we have to prove the following two theorems. They are concerned with the termination of Algorithm 5.2.15. The first one says the following: if all SimCon-instances with \( n \geq \#(E, C) \) nodes are not prime, then also all instances with more than \( n \) nodes are not prime. This theorem justifies the termination criterion of the algorithm described above.

**Theorem 5.2.16** Let \( E \) be a non-recursive SCA expression in the context of a schema \( \Sigma \). Let \( C \) be a compositional constraint in the context of \( \Sigma \). We then have

\[
(\forall n \geq \#(E, C) : (\forall I \in \mathcal{I}_E : \#I = n \Rightarrow I \not\in P^E_E) \Rightarrow (\forall I \in \mathcal{I}_E : \#I > n \Rightarrow I \not\in P^E_E))
\]

**Proof**

We are going to prove the following formula for all \( n \in \mathbb{N} \):

\[
n \geq \#(E, C) \Rightarrow (\forall I \in \mathcal{I}_E : \#I = n \Rightarrow I \not\in P^E_E) \Rightarrow (\forall I \in \mathcal{I}_E : \#I > n \Rightarrow I \not\in P^E_E))
\]

We do this by proving the following formula, using induction to \( n \):

\[
n \geq \#(E, C) \Rightarrow (\forall I \in \mathcal{I}_E : \#I = n \Rightarrow I \not\in P^E_E) \Rightarrow (\forall I \in \mathcal{I}_E : \#I = n + 1 \Rightarrow I \not\in P^E_E))
\]

First, let \( n = 1 \). If \( 1 < \#(E, C) \), then the formula obviously holds, as the premise \( 1 = n \geq \#(E, C) > 1 \) is false.

If \( 1 = \#(E, C) \), then consider an instance \( I \) which contains exactly one simple object without incoming or outgoing references. There is no other minimal composition of \( I \) than \( I \) itself (cf. Definition 5.2.5). So \( I \) is prime. This means that \( (\forall I \in \mathcal{I}_E : \#I = n \Rightarrow I \not\in P^E_E) \) yields false and thus the complete formula yields true.

Now suppose \( n > 1 \) and \( n \geq \#(E, C) \) (for \( n < \#(E, C) \), the formula holds obviously). Furthermore, suppose that all instances with cardinality \( n \) are not prime, i.e. \( (\forall I \in \mathcal{I}_E : \#I = n \Rightarrow I \not\in P^E_E) \). Let \( I' \in \mathcal{I}_E \) with \( \#I' = n + 1 \). We now have to prove that \( I' \) cannot be prime, i.e. \( I' \not\in P^E_E \).
As \( n \geq \#(\mathcal{E}, C) \) and thus \( n + 1 > \#(\mathcal{E}, C) \), we know that \( I' \) contains two simple objects \( x \) and \( y \), such that the result of applying \( \mathcal{E} \) onto \( x \) does not depend on \( y \) and vice versa, i.e. whether or not \( x \) is selected by some selector in \( \mathcal{E} \) is irrespective of the presence of \( y \) (and vice versa).

Now let \( I_1 \in \mathcal{F}_2 \) with \( \#I_1 = n \) such that \( I' \) can be formed out of \( I_1 \) by adding \( y \) to \( I_1 \) and possibly some references to/from \( y \) from/to objects in \( I_1 \). Analogously, let \( I_2 \in \mathcal{F}_2 \) with \( \#I_2 = n \) such that \( I' \) can be formed out of \( I_2 \) by adding \( x \) to \( I_2 \) and possibly some references to/from \( x \) from/to objects in \( I_2 \). By assumption, we know that \( I_1 \) and \( I_2 \) are not prime.

All this means that \( I' \) is not prime, as \( I' \) can be composed from \( I_1 \) and \( I_2 \) according to Figure 5.8, also using the fact that \( I_1 \) and \( I_2 \) are not prime.

![Figure 5.8: Decomposition of \( I' \) into \( I_1 \) and \( I_2 \)](image)

The second theorem says that Algorithm 5.2.15 will always end, i.e. for each operation and each schema, there is only a finite number of prime SimCon-instances.

**Theorem 5.2.17 (number of prime SimCon-instances is finite)** Let \( \mathcal{E} \) be a non-recursive SCA expression in the context of a schema \( \Sigma \) and \( C \) a compositional constraint. The set of prime SimCon-instances \( \mathcal{P}_C^{\mathcal{E}} \) is finite.

**Proof**

If the set of prime SimCon-instances in the context of \( \mathcal{E} \), \( C \) and \( \Sigma \) would be infinite, then the number of different objects modulo \( C \) and \( \mathcal{E} \) would have to be infinite. For if this number would be finite, then all instances which are sufficiently large would have to contain more than \( N \) simple objects (for a certain number \( N \)) which are equal modulo \( C \) and \( \mathcal{E} \). But for \( N \) sufficiently large, such an instance \( I \) is not prime, as the impact \( \#(\mathcal{E}, C) \) is finite (cf. Definition 5.2.14), so \( I \) can be decomposed into two instances that each contain less than \( N \) such nodes.
However, the number of different objects modulo $C$ and $E$ is finite because of the following reasons:

1. We know that $E$ is a non-recursive SCA expression. So the simple objects that will be modified by $E$ in a certain instance are determined and then $E$ is applied, giving the resulting instance. This means that application of a non-recursive SCA expression will always terminate. So, it will not remain generating new (different) objects forever.

2. $E$ is finite, which means that the number of simple and container types, the number of references and the number of attribute names of each simple type is finite$^7$.

3. By construction, the number of relevant attribute values is finite, as we only consider the attribute values that occur in the selectors in $E$ and $C$ (and $E$ and $C$ are finite).

4. The number of different references of an object is finite, as the number of reference names is finite, the number of different objects modulo $E$ and $C$ to which an object can refer is finite (induction) and the length of a relevant reference path from an object is finite, because the simple object selectors in $E$ are finite, $E$ is non-recursive and $C$ is finite.

\[ \square \]

In the aforementioned theorem, we have shown that the number of prime SimCon-instances in the context of a non-recursive SCA expression $E$ and a compositional constraint $C$ is finite. This is a necessary requirement for our method to be applicable in practical situations. However, this is not sufficient: if the number of prime instances would grow very fast (e.g., exponentially) in the "size" of $E$ and $C$, it would be very difficult, if not impossible, to apply our method in real-life situations. Fortunately, the time-complexity of the Prime State Method is not that bad: the number of prime instances in the context of $E$ and $C$ is determined by the impact of $E$ and $C$, which depends in a linear way upon the complexity of $E$ and $C$ (cf. Definitions 5.2.12 and 5.2.13). Thus, although further research is needed to determine the computational complexity of the Prime State Method more exactly, our claim that our method is applicable in practical situations, seems to be justifiable.

Below, the main theorem of this chapter is formulated. Informally, it says the following. Let $C$ be a compositional constraint, $E$ a non-recursive SCA expression and $I$ a SimCon instance. When we want to know whether $E$ preserves the invariance of $C$ in $I$, it is necessary and sufficient to check all prime SimCon-instances induced by $C$ and $E$.

$^7$In this chapter, we only consider persistent objects and properties that "fit" the schema in the context of which the expression is applied. If one wants to verify a property regarding temporary properties, the relevant object types, attributes and references have to be included in $E$. As an SCA expression is finite, the number of auxiliary objects and properties is finite, so the extended schema is still finite too.
5.3 Examples

Theorem 5.2.18 (Checking prime instances necessary and sufficient) Let $C$ be a compositional constraint. Let $E$ be a non-recursive SCA expression. Then:

$$(\forall i \in \mathcal{P}^E_C : C(i) \Rightarrow C(E(i)))$$

$\iff$

$$(\forall I \in \mathcal{I}^E : C(I) \Rightarrow C(E(I)))$$

Proof

Proving this theorem is equivalent to proving the following:

$$(\exists i \in \mathcal{P}^E_C : \neg(C(i) \Rightarrow C(E(i))))$$

$\iff$

$$(\exists I \in \mathcal{I}^E : \neg(C(I) \Rightarrow C(E(I))))$$

Here, the $\Rightarrow$-part is trivial: let $i$ be a prime $\text{SimCon}$-instance in which $E$ does not leave $C$ invariant, then we also have a $\text{SimCon}$ instance $I = i$ in which $E$ does not leave $C$ invariant, by construction of Algorithm 5.2.15.

For proving the $\Leftarrow$-part, let $I$ be a $\text{SimCon}$-instance which satisfies $C$. Let $E$ be a non-recursive SCA expression that violates $C$ in $I$. So, we know that $C(I) \land \neg C(E(I))$ holds. Now we have to prove that there exists a prime $\text{SimCon}$-instance $i \in \mathcal{P}^C$ such that $C(i) \land \neg C(E(i))$ holds.

If $I$ is not a prime $\text{SimCon}$-instance, then we know that $I$ can be described as a composition of prime $\text{SimCon}$-instances $\{I_j\}$, for some $n > 1$. Using Definition 5.2.4 and the fact that $C$ holds in $I$, we then have that $C(I_j)$ holds for all $1 \leq j \leq n$. Furthermore, we have $E(I) = E(I_1) \cup \ldots \cup E(I_n)$. But using Definition 5.2.7 and the fact that $\neg C(E(I))$ holds, we then know that $\neg C(E(I_j))$ holds for some $j$. So we have $C(I_j) \land \neg C(E(I_j))$ for some $1 \leq j \leq n$. So a prime $\text{SimCon}$-instance $i = I_j$ exists that satisfies $C$ but for which $E$ violates $C$.

If $I$ is itself a prime $\text{SimCon}$-instance, then we have $i = I$.

\[\Box\]

5.3 Examples

In this section, we present two examples in which the Prime State Method is used to verify a design, detect errors and suggest improvements to the designer. In order to keep the examples as clear (and thus as illustrative) as possible, we have chosen to give a few examples that are rather small. Our method can be applied onto larger specifications in an analogous way, for it works for each transition separately. So, the number of transitions in a $\text{SimCon}$ model we want to verify does not really augment the complexity of the problem.
The first example specifies part of the immigration office (IO) of an international airport. The immigration officers have to take care that someone who has been declared *persona non grata* does not enter the country. In order to decide whether someone may enter the country, the immigration office disposes of a small information system that holds information about all undesirable persons. We describe this situation by means of a SimCon model. The constraint that has to be satisfied is the following: no *persona non grata* will ever cross the border.

**Example 5.3.1 (airport)** The object schema from Figure 5.7 holds the simple and container types that are necessary to model the aforementioned situation in SimCon. It contains three simple types *person*, *passport* and *png-reg*. We model persons who have a *face* and may bear a passport with a *photo* on it. A *png-reg* object refers to the passport of a person that is undesirable and it records the *reason* for this. The simple types *person* and *passport* constitute one container type *passenger*. Moreover, there is a singleton container type *png*, consisting of *png-reg*.

![Diagram](image)

*Figure 5.7: Airport object schema*

The immigration office is depicted in Figure 5.8. Passengers arrive at the queue. Upon arrival of a passenger, IO decides whether he may cross the border, based upon information from the database *pngDB*. If this is the case, the passenger is put into exit. Otherwise, he has to go to *to-plane*.

The constraint $C$ that we have to check is the following: no person who is *persona non grata* will ever cross the border. Formally:
(\forall (id_1, ct_1, cs_1, pngDB, at_1) \in png : \\
(\forall (id_2, ct_2, cs_2, exit, ct_2) \in passenger : \\
(\forall (id_3, person, rf_3, av_3) \in cs_1 : \\
(\forall (id_4, png, rf_4, av_4) \in cs_2 : rf_3.bears \cap rf_4.concerns = \emptyset)))

So, an instance I satisfies C if it does not contain a reference from a simple object in pngDB to a simple object in exit. Now suppose that we have an instance I which is composed from I_1, \ldots, I_n. For C to be compositional, it should hold that 
(\forall i : 1 \leq i \leq n : C(I_i)) implies C(I), i.e. if no composing instance contains such a "forbidden reference", the composition I does not either. But this obviously holds because of (iv) in Definition 5.2.4, which states that for each reference in I there has to be at least one corresponding reference in one of the I_i.

The designer has specified the functionality of IO in terms of SCA. If J is the instance upon which the body E of IO is applied, the resulting SimCon-instance is J'' where

\[ J' = CM(J \cdot person \rightarrow passport \rightarrow png-reg [], \\
\text{photo} = person [face], to-plane) \]

\[ J'' = CM(J' \cdot person, queue [\cdot, exit]) \]

Intuitively, all persons bearing a passport which is registered as belonging to a persona non grata are sent back to the plane. All other people in the queue may go to the exit of the airport, thus entering the country.

Suppose the designer uses the Prime State Method to prove the correctness of his specification. The impact of E and C is 7, so we have to check at least all states
containing less than eight simple objects. In each state that contains at most two simple objects and that satisfies the constraint, the constraint is not violated by $E$. However, consider the instance that is depicted in Figure 5.9. This instance consists of three simple objects and three containers. Suppose the container with label $\ell_1$ is in the place $\text{pngDB}$, that with label $\ell_2$ is in queue and the container with label $\ell_3$ is in another place in the system. It is obvious that this instance satisfies the constraint, as the place $\text{exit}$ is empty in this instance.

![Diagram](image)

Figure 5.9: An example instance in the Airport example

The reader may verify that the instance that is depicted in Figure 5.10 is the result of applying $E$ upon the SimCon-instance from Figure 5.9, if the container with label $\ell_1$ is put back into $\text{png}$ and the one with label $\ell_2$ into $\text{exit}$.

In this instance, the constraint does not hold anymore. So, the designer has made a mistake. This counter-example gives him a clue about what he has overlooked: a passenger who has left his passport at home is allowed entrance to the country. Of course, this is a mistake: people without a passport should be sent back anyway.

The designer changes the functionality of $\text{IO}$ by defining a new SCA expression. When $\text{IO}$ occurs upon a SimCon-instance $I$, the result is then the SimCon-instance $I''$, where
Figure 5.10: Instance after occurrence of IO
5. Automatic Constraint Verification

\[ \begin{align*}
J' &= \text{CM}(\text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person}) \\
\text{photo} &= \text{person}[\text{face}], \text{to-plane} \\
J'' &= \text{CM}(J', \text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person}) \\
J''' &= \text{CM}(J', \text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person} \rightarrow \text{passport} \rightarrow \text{mug-reg} \rightarrow \text{person})
\end{align*} \]

Using the Prime State Method, the reader may verify that I0 behaves correctly this way: this second SCA expression leaves the constraint invariant. \(\Box\)

The second example considers an information system of a university. We describe the office that has to take care of striking off graduates. The IS contains some information about the exams the students have passed. On the other hand, a student knows which exams he has passed. For graduates, all information is deleted (probably it would be transferred to another database, but we do not care about that here). The obvious constraint we want to hold is that the IS does not "forget" exams that have been passed by students.

Example 5.3.2 (university IS) The object schema from Figure 5.11 holds the simple and container types that are necessary to model the aforementioned situation in SimCon. There are four simple types student, card, adr-card and res-card. So, we model students with a name who know that the university database holds some cards concerning them. Such a card may be an adr-card holding information about the address of a student or a res-card concerning an exam the student has passed. Such a card refers to the student about whom it contains some information.

![Figure 5.11: University object schema](image)

The desk that strikes the information about a graduate is depicted in Figure 5.12. Students arrive at the desk. Upon arrival of a student, all cards that concern him are removed. Afterwards, the student goes to the exit.
The constraint we have to check is the following: each card of which a student knows that it should exist, does really exist (referential integrity regarding knows). This constraint is obviously compositional: it is not possible to compose an instance from a number of instances which all satisfy this constraint in such a way that a "dangling reference" is introduced.

The designer has specified the functionality of the desk in terms of SCA. If $J$ is the instance upon which the body of desk is applied, the resulting SimCon-instance is $Y''$ where

$$
Y' = SD(J, \text{card} \rightarrow student, graduate[]) $
$$

$$
Y'' = RD(Y',student, graduate[], card[], knows) $
$$

$$
Y''' = CM(Y'', student[], exit) $
$$

Intuitively, this expression means that all cards about a graduate are deleted and that this student is moved to the exit.

Again, the designer uses the Prime State Method to verify his design. The impact of the expression is 5, so he has to check all instances with at most five nodes. For all instances with less than three nodes, he will not find any errors. However, if two students have done an assignment together and only one card has been made to record this, this design gives an erroneous result: if one of them graduates and the other one does not (yet), the card concerning their joint assignment is removed and thus the constraint is violated.

The designer changes the expression as follows: all references from the graduate to any card are removed, then all cards without outgoing arcs are deleted and finally the graduate goes to the exit. $J_6$ is the instance that is obtained by applying this
new SCA expression upon an instance }

\[
\begin{align*}
J_1 & = \text{RD}(\{\text{card} \rightarrow \text{student,graduate} \}, \text{student,graduate} \}, \text{about}) \\
J_2 & = \text{CC}(J_1, \text{udb}, \text{save}) \\
J_3 & = \text{CM}(J_2, \text{card} \rightarrow \text{student} \}, \text{save}) \\
J_4 & = \text{SD}(J_3, \text{card}, \text{udb} \}) \\
J_5 & = \text{CM}(J_4, \text{card} \}, \text{udb}) \\
J_6 & = \text{CM}(J_5, \text{student} \}, \text{exit})
\end{align*}
\]

Application of the Prime State Method yields that this expression is correct in the sense that it does not violate the constraint stating that the IS may not "forget" relevant information.

\section{5.4 Transaction Analysis: Examples}

In this section, we demonstrate how the theory from Section 5.2 can be used for transaction analysis. When specifying transactions (instead of "atomic" transitions), one wants some constraints to be satisfied in certain classes of states. However, in intermediate states, these constraints do not necessarily have to hold. Also, situations are possible where a larger part of a net does not violate a certain constraint, whereas when looking at individual transitions, the net seems to be incorrect, as (at least) one of the transitions may violate the constraint. Especially when modelling according to the principles of Section 6.3, the approach of the present section can be useful. An interesting item for further research is the generalisation and automation of the kind of verification from this section.

The Prime State Method checks for each transition \( t \) in a net whether \( t \) may ever violate some compositional constraint \( C \). However, in the example below, a situation is sketched where transition \( t_2 \) might violate some constraint \( C \), but which can never happen as \( t_1 \) preserves the net from going to a state from which \( t_2 \) would violate \( C \). When the Prime State Method is applied to this example, it will say that the specification is not correct, as \( t_2 \) may violate \( C \). However, the specification as a whole does not violate \( C \).

\begin{example}[sequential transaction] (sequential transaction) Consider a schema with simple types \( A \) and \( B \) and one container type \( C \), containing \( A \) and \( B \). Furthermore, \( A \) has an attribute \( x \) of type integer and a reference \( r \) to \( B \). Consider the net from Figure 5.13. Suppose that the constraint we want to check is referential integrity.

If the Prime State Method is used, this gives an indication that \( t_2 \) contains an error, as it will violate the instance \( I_2 \) in Figure 5.3 (this instance contains two \( A \)-objects referring to one and the same \( B \)-object; moreover, exactly one of the \( A \)-objects has an \( x \)-attribute with value 3). However, when we consider the net from Figure 5.13 as a transaction, the constraint will never be violated, as \( t_1 \) simply deletes all incoming \( A \)-objects, so \( t_2 \) will never be applied onto \( I_2 \) (nor onto a similar instance).
\end{example}
Example 5.4.1 gives us inspiration about how the Prime State Method could be applied for verifying transactions. What we have to do is combine the defining SCA expressions of the transitions in the transactions. In Example 5.4.1, this means that we get one transition $t_{1,2}$; the behaviour of $t_{1,2}$ is specified by the composition of the expressions defining $t_1$ and $t_2$. So, the expression defining $t_{1,2}$ is

$$\mathcal{E} = \text{SD}(\text{SD}(J_{1A}[|]), \text{BS}[L_{1A}[x = 3]])$$

If we want to use this composition method for transactions (which may produce incorrect intermediate results), we only have to apply the Prime State Method onto $\mathcal{E}$. If we also prohibit incorrect intermediate results, it has to be applied both on $\mathcal{E}$ and on the expression defining $t_1$, as $t_1$ might cause an intermediate result that violates the constraint.

In Example 5.4.1, we saw the simplest way of constructing a larger net, viz. by connecting two transitions. In this case, it is obvious how the composite transition looks like: simply compose the defining expressions of the two transitions.

The next example illustrates a more complex transaction, viz. one that contains a loop. In the example, we show how such a transaction can be replaced by one transition. We do this by defining the body of this transition using the FA-operation. As the Prime State Method only is applicable for non-recursive expressions, this method cannot be used when the transaction is such that an RA-operation would be necessary.
Example 5.4.2 (transaction with repetition) Consider the net that is depicted in Figure 5.14. We want to deal with this net as one transaction, meaning that we only require that all states in which the intermediate place $P_2$ is empty, satisfy some constraint $C$. The expression $E_2$ is a selective expression: if there are no objects satisfying some sos, it puts something into $P_3$ (executing some expression $E_{2,1}$). Otherwise, it takes one of these objects satisfying sos, applies some expression $E_{2,2}$, putting the result into $P_2$. The result is such that the chosen simple object satisfying sos will not be chosen again.

\[ E_{2,1}(FA_x,y)(j, sos, E_{2,2}(y', x))) \]

\hfill $\Box$

5.5 Conclusion

In this chapter, we have presented a method for automatic verification of constraints within the SimCon framework. We have restricted ourselves to so-called
compositional constraints. Although we think this method is very promising, it is not completed yet. Future research should cover at least the following aspects. As has been mentioned, we would like a more general characterisation of the class of compositional constraints. Also, we should pay some attention to the computational complexity of our approach. When the method will have come to full maturity, we will implement it within the SimCon tool. Then, we also have to cover some implementation issues. E.g., we have to make the algorithm that generates the prime states more explicit. In particular, an efficient algorithm for checking whether some instance is prime or not (a combinatorial problem) should be developed. Implementation in the tool may also force us to take tighter criteria in our method, e.g. for the impact of an expression and a constraint.

Concluding, we want to mention three alternative ways of validating/verifying a specification. These can be used in case the Prime State Method cannot be used for some reason. First, the notion of type checking a specification can be extended to SimCon, e.g. in order to check statically whether the SCA expressions that define the transitions in the net, satisfy the schema. E.g., it can be checked whether created references satisfy the reference structure of the schema in the context of which the operation is executed.

From a pragmatic point of view, it may also be useful to implement some runtime checks that give an alarm signal as soon as a constraint is violated during simulation of a specification. This does not provide complete certainty, but it helps the designer when validating a specification.

Finally, it should be possible to develop a verification technique that is based on the theory of T-invariants. Recall that a T-invariant specifies some firing sequence of the transitions in the net such that the initial state of the system is recovered. For certain classes of constraints, it may be possible to define an automatic expansion of the net, such that this augmented net has a T-invariant if it satisfies some constraint C. The idea behind this is that if the transition destroys certain information in a way that the constraint is violated, the augmented net will never be able to recover the initial state. However, if the constraint is not violated, this recovery should be possible. It would be interesting to try to work this idea out.
Chapter 6

Modelling Aspects

In this chapter, we make some remarks about the way of using the SimCon approach. We start by giving some guidelines for applying SimCon. After that, we demonstrate how SimCon can be used in practice by working out a case. Section 6.3 describes a layer that can be used on top of SimCon. This gives a particular way of working with SimCon.

6.1 Modelling Guidelines

In the previous chapters, we have presented the SimCon approach. Generally, we think that it is desirable for a modelling approach to provide some guidelines about how it should be used. However, we do not want to give strict rules which must be obliged. Moreover, we do not have sufficient experience yet to justify one specific way of working: we will simply have to pursue a number of case studies in order to obtain practical experiences with our framework. This may cause us to change some things, which will again have to be validated in practice. When this process will have been finished, we might prescribe a more strict way of working with our model. However, even in that case, we will probably let the designer choose for himself how he wants to work. Of course, we will give some advise, but we will not prohibit him from using his own method.

In the present section, we give some guidelines for applying our approach. These guidelines have been inspired by [Van der Aalst, Van Hee, Trèves & Di Giovanni, 1993; Van Hee, 1993]. In Section 6.3, we work out one possible way of applying our framework in more detail.

In software engineering literature, one distinguishes three well-known ways of developing a specification. We explain these approaches in the context of our framework:

- The process-oriented approach. The dynamic aspects of the system prevail when modelling this way. One starts by describing dynamic properties of a
system. Only later, some data aspects are added. This approach consists of four steps:

1. Design a hierarchical net model, up to the level of transitions. The "typing" of the places and the behaviour of the transitions are not yet defined formally at this stage; however, some informal description is given. In this informal way, we try to capture the most relevant aspects of the net. The result of this stage is called an (incomplete) net model. Generally, one would perform this step in a top-down way, meaning that one starts by making a net model on a high level of abstraction which is worked out until all necessary details have been added. On the lowest level of abstraction, the behaviour of the transitions is described. Sometimes, (part of) the design will be developed bottom-up, e.g. when reusing existing components with a known semantics.

2. Design a SimCon object schema for the total system. In this schema, the entities that play a role within the system have to be defined. Moreover, we specify some type definitions for covering those "entities" that we do not want to model by means of the object model, e.g. the tokens that are used for realising control flow functions. Afterwards, we formulate some constraints upon the object schema. This way, some additional properties of the objects in the system are described. Finally, we assign a container type or a type from the functional language to each place in the net model, defining which kinds of tokens (objects or values) may occur in each place.

3. Design an SCA expression for each transition, defining the exact behaviour of that transition. The transition body may also contain some expressions from the functional language, with the purpose to model control flow and the like. In fact, this is a formalisation of the informal description that results from (1).

4. Use the Prime State Method to check whether the constraints from the second step are satisfied by the model that has been obtained after step 3. Also, other validation and verification techniques may be used for unearthing errors in the design. This will result in a repetition of the previous three steps.

- The *data-oriented* approach. In this way of working, the static aspects of a system play the most important role. First, all static system aspects are described. Only then, a description of the dynamic aspects is added. In fact, the data-oriented approach consists of the same steps as the process-oriented approach. However, the steps are performed in a different order: the first two steps are exchanged.

- The *integrated* approach. Here, the same activities are pursued as in the previous ways of modelling. However, in the integrated approach, they are mixed. How these steps are mixed is up to the designer. E.g., one could first develop a partial object schema and a partial net structure, covering all aspects of a certain object type O. Schema and net are then extended to cover some object type O', etc. One could also develop both a net model and an object schema in a top-down way, but more or less in parallel, contrary to the process-oriented
and the data-oriented approaches. The case study that is presented in the next section illustrates the integrated approach. Moreover, in Section 6.3, we present a special way of applying the integrated approach, which is sometimes called the object-oriented approach.

Of course, a set of modelling guidelines could be provided that is much more detailed. Van der Aalst, Van Hee, Trèves & Di Giovanni [1993] have made a first effort in this direction. E.g., they say something about the differences between using (ordinary) places and applying stores\(^1\). Also, some statements are made about modelling something by means of a token versus modelling it by means of a subnet. Finally, some typical situations are sketched, together with some suggestions for specifying them.

The aforementioned issues indicate that it may be desirable to have a specification framework which allows for postponing decisions about modelling some part of a system either by means of a subschema of by means of a subnet. A typical example illustrating this problem is the following: suppose we have to model some organisation in which machines play a role. Should we model a machine by means of a transition or as a container object? Opting for the first choice puts more emphasis on the dynamics of a machine. On the other hand, this would mean that it would be less straightforward to model a dynamic machine configuration. In the requirements analysis phase, it might be useful to be able to postpone such decisions. We have some ideas about extending the SimCon approach with a diagramming technique that could be used in the early stage of the requirements analysis process. Part of such a diagram would then be worked out into a SimCon object schema; the other part would be represented by a SimCon net. Together, they constitute an integrated SimCon model. This is a topic for further research.

The next section illustrates how SimCon can be applied to larger modelling problems. For that purpose, we use a case study that has been defined within the ISDF project, which has been described among others in [De Brouwer & Verkoulen, 1992].

6.2 Case Study

In this section, we present an overview of a case study using SimCon. We sketch a model of a virtual transport company TransCorp. Because of space limitations, we do not describe our solution in full detail. However, the results presented in this section illustrate how SimCon can be used in practice. Besides presenting a solution to the problem, we also demonstrate how this solution has been obtained.

\(^1\)A store is a special kind of place that contains always exactly one token.
6.2.1 An Informal Description of the Case

The primary function of TransCorp is the distribution and temporary storage of goods. For carrying the goods, the firm has a number of different trucks at its disposal; for the temporary storage of the goods, the firm possesses a number of depots, with different configurations and capacities. The company needs different kinds of trucks and stores as there are different kinds of cargo:

- Two kinds of solid goods, viz. plain goods, like boxes, and refrigerated goods.
- Two kinds of liquid goods, viz. dangerous liquids (e.g. acid) and harmless liquids, like milk.

The firm possesses three kinds of trucks, viz. plain trucks, refrigerated trucks and tankers. The trucks for carrying liquids have to be cleaned after usage, as liquids are not wrapped up. An exception to this rule is raised when the new cargo is of the same kind as the previous one. Cleaning a tanker takes about half an hour when the cargo was not dangerous and an hour otherwise. A truck may contain several cargos of possibly different clients, provided that the cargos do not interfere with each other (e.g., it is not possible to carry two different kinds of fluids together).

Each depot contains three kinds of stores: plain stores, cold-stores and tanks. The tanks must again be cleaned between storing two different fluids. This takes about two hours for dangerous goods and about one hour otherwise. Of course, depots may contain several cargos of possibly different clients. Finally, in general the depot has several stores, so it is possible to store different kinds of goods in one depot at the same time.

The transport firm consists of an administration department, a planning department and an execution department. The administration records all kinds of information, e.g. the arrival of an order or the absence of a driver. The planning department checks whether an order can be accepted and schedules it for execution if this is the case. The execution department is responsible for the actual execution of the order.

Each depot has a local administration which records the goods which are present at the depot or which are to be expected in the future.

TransCorp works according to the following global procedure:

1. A client contacts the administration and places an order. If the client is solvent, the order is preliminarily accepted. Otherwise, the order is postponed until the client has made an initial payment (upon request of the administration). Moreover, the administration performs a rough check whether the capacity is sufficient for accepting the order.

2. If the order is preliminarily accepted by the administration, the planning department checks whether there is sufficient capacity (trucks and stores) to accept the order definitively. This means that there has to be sufficient truck capacity and sufficient depot capacity. The planning department has to take into ac-
count that a transport may take several days and that a cargo may be stored temporarily in one or more transit depots before it reaches its final destination. The planning department sends a message to the client, confirming the acceptance or rejection of the order. Of course, the planning department tries to make such a schedule that the costs of executing an order are minimalized.

3. Each 24 hours, the planning department sends a plan to the execution department. The execution department modifies the plan by taking into account absent drivers and trucks which are out of order. Because of this, it is possible that the plans cannot be executed completely. Therefore, the execution department sends a message at the end of each day to the planning department saying which plans have been realized and which have not. The latter ones can then be rescheduled.

4. The planning department also sends messages to the depots about which trucks are to be expected. When making the plans, the planning department has to account for the capacity of the depots, the number of people working at the depots, the opening and closing times, etc.

5. When an order has been realized, the execution department sends a message to the administration. The administration sends a bill to the client and stores the information about the order in a file concerning completed orders.

Besides this global way of working, there are also some secondary requirements. Moreover, we have made certain assumptions when making the specification. We first present the additional requirements:

1. Each order has to be executed on the day that has been agreed upon with the client.

2. Roads can be blocked, e.g. because of an accident.

3. After a month, a truck is in repair for 3 to 12 hours.

4. Drivers work 40 hours a week. They may not work more than 8 hours a day and they have to rest after this for at least 6 hours. When it is absolutely necessary that a driver works longer, it is allowed. However, the company has to pay him extra then, so this has to be avoided as much as possible.

5. A driver has to be authorized to drive the trucks he does. This also holds for transporting dangerous goods. Moreover, not all roads may be used to transport dangerous goods.

6. The costs have to be minimalized, so trucks should drive as less as possible and have a cargo as large as possible.

We conclude the introduction to the TransCorp case by presenting some assumptions we have made while working out the case:
1. An order can be contained by one truck. If a client wants to place larger orders, he has to split them himself.
2. The planning department works on a daily basis.
3. An order has to be placed at least one day before its execution date.
4. Data about trucks which are out of order or absent drivers come from outside the system; they fall outside the scope of this specification.
5. We assume that the depots are open 24 hours a day. If necessary, the specification can be changed in a relative easy way to overcome this assumption.
6. We do not specify a planning algorithm. We assume that the routes come from outside and are stored in a table somewhere in our company. These routes take into account the aforementioned constraints about minimizing truck kilometres and roads which may not be used for transporting dangerous goods.

In the next section, a (partial) solution of the TransCorp case is presented.

6.2.2 The SimCon Specification

In this section, we sketch a solution for the TransCorp case, using SimCon. In Section 6.1, we have described three ways of applying SimCon. The solution that is presented here has been obtained by applying SimCon in the so-called integrated way. First, we describe the global structure of the system by defining a global SCNET. The components of that SCNET are defined informally at that stage. Subsequently, a (partial) SimCon schema is introduced. This schema defines the most important object types and some of their relationships. Finally, the specification is completed by refining both the net and the schema. This has been done in a more or less parallel way.

On the highest level of abstraction, TransCorp consists of three parts (cf. Figure 6.1), viz. the company itself, the roadnet which contains the depots of the company and the roads connecting the depots, and the clients.

The clients have been modelled by a system that generates orders 3 and communicates about them with the company. The roadnet consists of depots and roads. First, we develop the company specification.

The company receives orders and payments from outside and generates messages for its clients. The administration department sends questions and plans to the depots in the roadnet and receives answers from these depots. This is necessary, as the actual situation at the depots may differ from the planning. So, a situation

---

2 It would also be possible to adapt the specification such that orders are split when they enter the administration, when necessary.
3 We leave the exact way of generating orders unspecified. When prototyping the design, we may apply some specific way of generating orders, based on real-life data. Also, it is possible to generate orders simply by using a random generator.
might occur where an order can be executed according to the planning, but which still has to be rejected because older plans have not been realised completely.

The company again consists of three subsystems, viz. the administration, the planning department and the execution department (cf. Figure 6.2).4

The administration receives orders and payments from the clients and it sends messages to the clients. Accepted orders are sent to the planning department, which adapts the plans on basis of a received order. The planning department sends plans and questions to the depots and concept plans to the execution department. This department sends final plans to the depots and collects information about finalised orders from the depots. This information is being sent to the planning department and to the administration, which sends a bill to the client.

In Figure 6.3, we present a first approximation of a SCOM schema for TransCorp. It only contains simple types with some of their relationships (without names and directions). These types have been derived directly from the informal description that has been given above. In a later stage, we complete the schema, by adding missing relationships and attributes, by refining some simple types and by adding container types.

We refine this schema in two stages. First, some attributes are added and the directions of the references are determined. This gives the schema from Figure 6.4. There are goods of four kinds. Clients place orders, requesting the transport of goods from one place to another, on a certain date. Moreover, an order has a status. For carrying these goods, TransCorp possesses different kinds of trucks, with different capacities. Moreover, there are drivers with licences for certain kinds of trucks. Also, information about trips that have been made by a driver is stored.

---

4Input and output pins, which have to be connected with places on a higher net level, are depicted by circles with a dashed incoming or outgoing arrow, respectively.
Figure 6.2: The transport company TransCorp

Figure 6.3: A first approximation of the TransCorp schema
When executing an order, the company sends a truck according to a given route. This route consists of connected roads. Roads have a status (blocked or free; only some roads may be used for carrying dangerous goods). Each depot is on a road.

![Diagram of the TransCorp schema]

**Figure 6.4**: A second approximation of the TransCorp schema

The company communicates with the clients about their orders. Besides placing orders, the clients also make payments.

Now, we present the administration (cf. Figure 6.5). Orders come in and are judged in the system receive order. If the company has globally sufficient capacity and the client is solvent, then the order is accepted immediately. Otherwise, a message is sent to the client, stating that the company cannot handle the order or that the client has to pay first. In the latter case, the order is postponed and will be dealt with as soon as a payment will have arrived. When an order is accepted, this is recorded in the store being treated and the order is passed to the planning department. Upon getting the message that the order has been finished completely, it will be removed from this store by the transition \(^b\) dispatch.

The next part of the firm that we present, is the planning department (cf. Figure 6.6). The system representing the planning department consists of three

---

\(^b\) In this section, transitions are depicted by rectangles with a filled triangle. Systems are depicted by plain rectangles.
subsystems, viz. checkcapacity which checks whether there is sufficient truck and depot capacity to deal with a certain order, makeplan, which makes the plans and sends them to the depots and to the execution department. The third system in planning is the system reportfinish. This system sends messages to the administration about finalised orders. It also passes this information to makeplan, which has to check which planned orders have really been finished.

For checking the capacity, checkcapacity has to be able to send questions to the depots. These questions are sent via the output place question, whereas answers arrive on the input place answer.

The last component of the company is the execution department. It modifies incoming concept plans, based upon data about absent drivers and trucks which
are out of order. Moreover, it reports daily to the planning about finalised orders.

Finally, we refine the object schema by adding names to the references in previous approximation. Also, we add a simple type tanker, which is a specialisation of truck. For a truck, we record its last cargo, as this determines whether the tanker has to be cleaned before loading it again. This illustrates how existing schemas can be extended by using the inheritance concept.

Without further explanation, we now give our final approximation in Figure 6.7.

![Diagram](image)

Figure 6.7: The TransCorp schema

In our opinion, this treatment of the TransCorp case should provide sufficient insight in how SimCon can be used in practise. Besides adding some more details, we also should introduce container types and assign them to the places in the net. Moreover, the transitions have to be defined by means of SCA expressions. As Chapter 3 contains a large amount of examples illustrating the use of SCA, we have not included an example here.

Of course, we should use the Prime State Method to verify this design. However, we think that Chapter 5 contains (within the scope of this monograph) sufficient material for illustrating this verification method. When real case studies are going to be performed, of course the Prime State Method will be used, which will provide suggestions for improvements of the method.
6.3 SimCon/IOM: A Layer on Top of SimCon

In Section 6.1, several ways of applying the SimCon approach have been sketched. One of them is the integrated approach, where data and process modelling are mixed. In this section, we demonstrate a specific way of applying this integrated modelling method. Many methods for object-oriented modelling contain concepts like objects, life cycles, actions, etc. In this section, we describe a model that consists of such concepts. The model has been introduced in [Ramackers & Verkoulen, 1992], where it has been called Integrated Object Model (IOM). We build upon these results; however, the model that we present in this section differs in some respects from the one described by Ramackers & Verkoulen [1992]. We use IOM as an extra layer on top of SimCon. In fact, this gives a way of applying SimCon. The combination of the two is called SimCon/IOM in this monograph.

This section starts with a description of the concepts of IOM. For a more complete introduction, we refer to [Ramackers & Verkoulen, 1992]. We illustrate SimCon/IOM by working out the problem from Example 4.2.1. In order to be able to make a comparison between SimCon/IOM and plain SimCon, we try to adhere to the problem description from that example as close as possible. Also, some remarks about tool support for SimCon/IOM are made. We conclude this section by giving a formalisation in terms of SimCon.

6.3.1 Informal Description

In the SimCon approach, we usually model active entities by means of Petri net transitions and passive entities by means of container objects flowing through the net. As we have seen, this strict separation cannot always be maintained, but one may want to apply this way of modelling as much as possible. Another way of modelling is an approach where "everything" is modelled by means of one kind of objects. Such approaches are often called object-oriented, as their main modelling construct is the object concept. IOM is such an approach.

An IOM object description consists of
- a unique identification. The benefits of using object identities have already been mentioned in previous chapters.
- a number of structural properties (attributes and relationships with other objects).
- a life cycle. This life cycle consists of a number of actions (with pre- and postconditions). The order in which these actions may be pursued by the object is described by its life cycle. The life cycle of an object may be partitioned, giving roles of the object.

All objects with the same attributes, relationships and life cycle constitute one object type. Often, we impose constraints upon (types of) objects. Constraints
are conditions ranging over the attributes and relationships of objects. There are two kinds of constraints, viz. the ones that may never be violated and the ones that may. The latter ones are called triggering constraints, as their violation triggers some action to be executed in order to "repair" this violation. In literature, such constraints have also been called deontic. Constraints on objects are defined as preconditions of special transitions that are added. The functionality of such a transition is straightforward: it simply passes all incoming objects to its output place, provided that they satisfy the constraint that has to be checked by that transition. In Chapter 4, we have mentioned that these guards can be expressed in the formal SCNET model. Of course, one may also apply the Prime State Method from Chapter 5 to check whether some deontic constraint is really satisfied by the net. This may be even more easy than in plain SimCon, as the behaviour of each object type is specified separately to some extent.

The description of the structural properties of the objects in the system is often called the data perspective, whereas the life cycle description covers the behaviour perspective. We also describe how processes model the interaction between objects; this covers the process perspective.

Example 6.3.1 (data perspective) Actually, the data perspective of IOM is covered by a textual representation of the attributes, references, specialisations and constraints of each object type. However, in SimCon/IOM, we cover these aspects by one object schema, as this gives a better overview of the data perspective. As can be seen below, only the simple types of this schema are really important: for each IOM object type, a simple type is introduced. Therefore, the container types have been omitted in Figure 6.8.

Figure 6.8 shows which object types exist, viz. driver, truck, order and trip. They all have some attributes and references; they do not require any further explanation. Observe that in comparison with the plain SimCon solution, an extra simple type trip is required, on the same level as the other types, to model that a truck and a driver may participate in a trip.

The structural properties of an object \( x \) constitute the state of \( x \). The state of an object may change in time. An object changes its state by performing a so-called assignment step: an element of an action where one or more of the attributes and/or relationships of the object are changed.

Besides assignment steps, there is a second kind of steps that may be part of an action, called interaction steps. When an object performs an interaction step, it sends some message to another object. Objects may also send messages to themselves. The receiving object performs some actions, which may result in a state change of that object and/or the transmission of messages to (other) objects. The

---

\[6] For the moment, we only cover local constraints, i.e. constraints that can be formulated at the level of one object type.
execution of an action with associated pre- and postconditions and a number of input and output-parameters is called an event. This definition of an event is only one of many that can be found in literature. It focuses on what happens, rather than what causes something to happen. The latter is the interpretation of an event as a trigger.

The next example illustrates the behaviour perspective.

**Example 6.3.2 (behaviour perspective)** Consider Figure 6.9. In this figure, the life cycle of the object type trip is depicted. It states that a trip is created after receiving some message (an order). Then it performs a claim action (it claims a truck (that can be used for pursuing the order) and a driver with a licence for this truck). Afterwards, it loads its cargo. After having moved its cargo, it releases the truck and its driver. The life cycle of trip is partitioned into two roles, viz. administration and shipping.

The circles in Figure 6.9 represent abstract state classes; think of them as if they hold all objects of a certain type that are all at the same stage in their life cycle. Each action (depicted by a box) has at most one incoming arc from and at most one outgoing arc to such a circle: we do not want to allow a situation where two objects are fused or where two objects are generated out of one. Figure 6.10 illustrates why we do not need multiple output arcs: the left-hand part can be replaced by the right-hand part.
6.3 SimCon/IOM: A Layer on Top of SimCon

Figure 6.9: Behaviour perspective: the trip life cycle

Figure 6.10: Branching in life cycles
The life cycle of an object describes its local behaviour. Messages are assumed to arrive "out of the blue". Hence, no attention is paid to aspects like synchronisation of events and other global dynamic properties of the system. For this purpose, the process perspective is introduced. In this perspective, the way that objects may trigger each other by sending messages is described. Unless a life cycle, a process is associated with a collection of objects.

The next example illustrates the so-called process perspective. In the process perspective, we describe in which way several objects interact. It is defined which objects are triggered by which objects and in which order the life cycles are passed.

Example 6.3.3 (process perspective) In Figure 6.11 the administration process is depicted. We see that after an order has been created, this causes a trip object to be created. After creation, this trip object performs a Claim action, causing an available truck and an appropriate driver to be claimed. After a while, the trip releases the truck and its driver and it triggers the issuing order object to prepare a bill. Also, the pre- and postcondition transitions are depicted: within the event net Claim of trip we see two darker coloured transitions; they are used to check optional pre- and postconditions for Claim.

\[\square\]

6.3.2 Formalisation

We formalise IOM by defining how a given IOM specification is expressed in terms of SimCon; for each IOM specification, a corresponding SCNET will be described\(^7\). The schema part of this SCNET covers the data perspective of IOM and the structure of the messages that objects may send to each other. The process and behaviour perspectives are covered by the net part of the SCNET. We introduce transitions for representing all "activities" (in the broadest sense) within an IOM specification. In the SimCon/IOM tool, we can do this using the concept of hierarchical nets. Also, we may use graphical facilities, e.g. for marking the different roles in the life cycle of an object type. When defining IOM formally in SimCon, these two aspects are covered by introducing some projection functions: we have to have some means of partitioning the global SCNET that gives the semantics of an IOM into relevant subparts. E.g., we have to be able to distinguish the particular part of the SCNET that covers the life cycle of some object type \(O\). This is illustrated by means of examples after the formal definition.

The data perspective of the IOM model is specified in terms of a SimCon object schema. The first (and for our purposes most important) component of this schema is a simple schema. Therefore, we first give a formal definition of this simple schema. Afterwards, we informally define the complete object schema.

\(^7\)From the viewpoint of SimCon, this gives a special way of applying SimCon.
Figure 6.11: Process perspective: the administration process
Definition 6.3.4 (simple schema for IOM) Given an IOM specification, the simple schema representing the data perspective of this specification is given by the tuple \((ST, AT, RF, IS)\), where

- \(ST\) is a set of simple types: it contains a simple type for each IOM object type and a simple type for each kind of message;
- \(AT = (AN, AS, DD)\) is the attribute structure. Here \(AN\) is the set of all attribute names from the IOM specification and \(AS \in ST \rightarrow (AN \rightarrow \mathcal{P}(DD))\) is a function assigning to each simple type and each attribute of that type the set of all possible attribute values, which is a subset of the attribute domain set \(DD\). For IOM objects, the attributes are described in the data perspective. Messages may also have attributes, in order to pass information between objects. The exact attributes can be derived by studying the process perspective.
- \(RF = (RN, RS)\) is the relationship structure. Here \(RN\) is the set of relationship names; it consists of all relationship names from the IOM model and two special names: \(sender\) and \(receiver\). \(RS \in ST \rightarrow (RN \rightarrow ST)\) assigns to each simple type and each relationship of that type, a simple type denoting the 'range type' of the relationship (for object types, the relationship names are specified in the IOM instance; for message types the relationship names are \(sender\) and \(receiver\) and any specified relationship parameters to be passed).
- \(IS \in ST \rightarrow \mathcal{P}(ST)\) defines the inheritance structure. For each object type \(t \in ST\), the set \(IS.t\) denotes the object types of which \(t\) is a specialisation. Note that this allows for multiple (structural) inheritance; name clashes and cyclic definitions are prevented by Definition 2.3.1. For each message type \(m \in ST\), it holds that \(IS.m = \emptyset\).

As has been mentioned, the tokens that flow through an SCNET are container objects, which are composed of simple objects. Therefore, the simple schema that has been defined above is extended with a singleton container type for each IOM object type and for each message type. Moreover, container types are introduced for each combination of objects and messages that may be aggregated within an action net.

The dynamic aspects of IOM, viz. the behaviour and process perspectives, are formalised in terms of an SCNET, a number of functions defined upon this net and some sets of names. Given an IOM model, we first define an SCNET \(\mathcal{N}\) which comprises, together with the aforementioned functions, a formal definition of the dynamic concepts of IOM. Afterwards, we define the functions. The mentioned sets consist of the action names, event names and process names, respectively. The transitions in the net are defined by SCA expressions. The designer has to construct a formal SCA expression that realizes the required (informally specified) functionality.

\[a\]Note that this does not require all container objects to be singletons.
Definition 6.3.5 (SCNET for IOM) We define an SCNET for an IOM specification to be a tuple \((\Sigma, P, T, A, C, R, M_0)\), where
- \(\Sigma\) is an object schema formalising the data perspective of the IOM specification and also covering the messages from the process perspective;
- \(T\) is the set of transitions. Each transition in this set either represents (i) an assignment step of an object, (ii) the creation and matching of messages and objects in an interaction step, or (iii) the checking of a pre- or postcondition.
- \(P\) is the set of places. Some of them are element of an action subnet of an object type. The containers in the action places within such an (induced) action subnet contain pairs of an object and a message for that object. Furthermore, there are message places between transitions of (possibly different) object types: objects trigger events on each other. There are also places between (induced) action subnets of one and the same object type. These places contain objects of that type and serve as abstract behaviour state places: all objects in such place are in the same abstract behaviour state.
- \(A\) is the bag of arcs, connecting the places and the transitions. These can be derived from the behaviour and the process perspectives.
- \(C\) is the container function, assigning to each place \(p\) the type of the objects that can be contained by \(p\).
- \(R\) is the reaction function. For a transition \(t \in T\) and an assignment \(e\) of container objects to the input places of \(t\), \(R_t(e)\) gives a set of generated container objects for the output places of \(t\). The reaction function \(R\) is determined by the formal semantics of the SCA expressions which realize the IOM operations that have to be executed by the transitions.
- \(M_0\) is the initial marking: it gives the objects that initially exist. In an initial marking, no messages are allowed and all present objects have to be in some abstract state within their life cycle.

We now introduce four projection functions in the context of Definition 6.3.5. The application of each of these functions on a net as described in that definition results in a (sub)SCNET. These result nets are also used to define the syntactic constraints on the structure of the IOM nets. For instance, an action net is a finite, sequential net with optional branching and/or loop constructs, and life cycles may have similar deterministic branching. These detailed syntactic aspects are not discussed here.

The function \(A\) (action net projection) has three parameters: an SCNET \(N\), an object name \(O\) and an action name \(a\). The application \(A_{N,O,a}\) gives the subnet of \(N\) that contains exactly those transitions and internal places that represent the action \(a\) of object type \(O\) in the net \(N\) (i.e., excluding the pre- and postcondition transitions). In Figure 6.12, we see (a part of) an SCNET \(N\) as mentioned above; the application \(A_{N,\text{ObjectA}(\text{ActionX})}\) on this net results in the action subnet within the darker coloured rectangle with label \(\text{ObjectA.ActionX}\). Note that
each IOM interaction step is split into a transition sending a message (cf. the transition with label “send in” in Figure 6.12) and a transition for receiving the answer (“receive out”) in Figure 6.12); it is not possible to model this by means of one transition, because such a transition would only be able to generate the question when receiving the answer: a deadlock situation. The “answer-receiver” also matches the incoming answer with the correct waiting sender object.

The function $E$ (event net projection) has four parameters: an SCNET $N$, an object name $O$, an action name $a$ and an event name $e$. The application $E_{N,O,a}(e)$ gives the same net as $A_{N,O}(a)$, together with the pre- and post-condition transitions. An example is illustrated in Figure 6.12: the result of applying $E_{N,\text{ObjectA},\text{ActionX}(\text{EventX})}$ on this net results in the subnet within the light coloured rectangle with label $\text{ObjectA}.\text{EventX}$. Similarly to interaction transitions, the precondition transition must also match incoming messages with the intended receiver objects.

The function $L$ (life cycle net projection) has two parameters: an SCNET $N$ and an object name $O$. The application $L_N(O)$ results in a net that consists of exactly all subnets $E_{N,O,a}(e)$ and the associated behaviour state places, for each event name $e$ and corresponding action name $a$ of object $O$.

The function $P$ (process net projection) has two parameters: an SCNET $N$ and a process name $p$. The application $P_N(p)$ restricts $N$ to all events (with the required message places) that are part of process $p$. If necessary, it connects these subnets by folding some behaviour places together per object type (cf. the example below: application of $P$ on the net in Figure 6.13(a) and the dark coloured process gives the net in Figure 6.13(b), and not that of Figure 6.13(c)). For each object type $O$ involved in a process $p$, this induces a role of $O$: the net $P_N(p)$ restricted to the actions of $O$ (denoted by $P_N(p)|O$) defines the role $p$ of $O$. Thus, a role is formally defined as a derived projection function.

Note that our approach for the formalisation of IOM can be considered “top-down”, since the dynamic concepts are formalised as functions on the complete net. However, this does not imply that our method can not be used for bottom-up development. One is not obliged to first design a large SCNET, and only then the functions $A, E, L$ and $P$. In the tool, it is possible to work bottom-up while defining these functions and finally, implicitly, the complete SCNET.

---

9 Often, this event name will be equal to the action name.
10 In the tool, the modeller will not see this flat net; each subnet $E_{N,O,a}(e)$ is aggregated in one node there. These nodes can be opened and closed upon request.
Figure 6.12: An abstract example
6.4 Conclusion

In this chapter, we have dealt with some issues regarding the application of the SimCon approach. First, some guidelines suggesting how SimCon should be applied, have been given. Actually, we have presented three approaches: the process-oriented approach, the data-oriented approach and the integrated approach. Of course, in all approaches, data and process modelling have been integrated. However, the different ways of working put emphasis on different aspects. If the dynamic aspects of the system under development are the most important, one would apply the process-oriented approach: one starts with a process description and adds the data aspects later. Similarly, one would apply the data-oriented approach if one wants to emphasize the static aspects. For many systems, there will not be one perspective that is clearly more important than the other one. In such cases, the integrated approach should be used, where data and process modelling are really mixed.

A special way of applying the integrated approach is the object-oriented way of working, in the literal sense of the word. In this approach, the object concept plays a central role: all system aspects are modelled in terms of the involved objects. In this chapter, we have presented such an object-oriented approach, viz. SimCon/IOM. Below, we present the most significant differences between plain SimCon and SimCon/IOM.
In SimCon/IOM, a system is described completely in terms of objects. For a designer who is used to this way of working, this may be very useful. However, it may be slightly artificial in some cases. It may be more straightforward to model some objects by means of transitions whereas other ones are represented by tokens, instead of modelling all different kinds of objects the same way. Another difference is the fact that in SimCon/IOM, the container level is not really used. This results in an additional simple type trip in the example that has been treated. As all kinds of objects have to be modelled as simple types in SimCon/IOM, it is thus not possible to make a difference between those two kinds of objects.

The guidelines for applying SimCon should be worked out in the near future. This can only be done by applying SimCon onto real-life problems. This will indicate which aspects of our model are strong and which things should be changed. Moreover, it will help us when we have to decide upon implementation issues that have not yet been decided. Only practical applications can provide us with the correct answers here.

Also, we will try to obtain some practical experience with SimCon/IOM. This will result in special tool facilities within the SimCon tool. Presently, someone who wants tool support for SimCon/IOM simply has to use the SimCon tool and obey the syntax and ideas from IOM. However, a situation where the designer is forced to follow these rules would be preferable. Moreover, SimCon/IOM requires specific features, which are not (yet) present within the SimCon tool. E.g., some facilities supporting the (implicit) definition of the projections functions are important.
Chapter 7

Concluding Remarks

In this chapter, we summarise the accomplishments of the research that has been described in this monograph. Also, we sketch some directions for further research.

7.1 Conclusions

In this monograph, we have described an integrated framework for information systems analysis and design. It can also be applied for developing other kinds of systems, e.g. logistic systems. We have called our framework the SimCon approach. It consists of a formal model, a verification method, some tool support and a number of guidelines regarding how to apply the approach. In the next paragraphs, we briefly repeat the most important characteristics of the components of our framework.

The first component is the SimCon Object Model (SCOM). This is an object-oriented data model. This object model covers the most important issues that have been recognised in the literature concerning object-oriented data modelling. The object model consists of two layers. The first one is the level of the simple objects. This level can be compared with existing object-oriented data models. It can be used for describing which entities play a role in a system, without paying attention yet to distribution, location and timing properties. A simple object has a unique identity, a number of attribute values and a number of references to other simple objects. Objects with similar properties constitute one simple type. Moreover, structural (multiple) inheritance is supported.

The second SCOM layer is that of the container objects. A container is an aggregate of simple objects. Moreover, a container has a location (it "resides" somewhere) and a time-stamp. Of course, this location may be abstract; it does not have to be a physical location.

The second part of the SimCon framework is the SimCon Algebra (SCA). This algebra is meant for manipulating instances from the SimCon Object Model. Such
an instance can be thought of a directed, coloured, partitioned graph. The nodes in this graph are the simple objects. The edges are the references between simple objects. As each simple object may have a number of attributes, we say that the nodes are coloured. Moreover, as each simple object has to be in exactly one container, the graph is partitioned. With these observations in mind, we have developed the SimCon Algebra. The fundamental operations from this algebra allow for creating and deleting simple objects, adding and removing attribute values to and from simple objects, creating and deleting references between objects, creating container objects and moving simple objects between containers. Finally, there is a recursor operation that can be used to apply other SCA-expressions recursively.

We have proved that this algebra gives sufficient expressive power. In fact, it is more powerful than the GOOD manipulation language and thus at least as powerful as the Nested Relational Algebra. The fact that SCA is really more powerful than GOOD is caused by some non-determinism in the SCA recursor operator. Some deterministic alternatives for this operator have been described, for those people who do not appreciate such non-deterministic operations. Also, in order to increase the expressive comfort of the algebra, we have expressed some high-level operations in terms of the SCA kernel. Finally, we have sketched a graphical version of SCA.

The SCNET model has been obtained by integrating the SimCon Object Model and its algebra within the high-level Petri net model of ExSpect. In this integrated model, the state space of a system is described by means of the SimCon Object Model: the tokens in the net become container objects and to each place, a container type is assigned that determines which kind of containers may occur in that place. Finally, the behaviour of the transitions is defined by means of SCA expressions. In the tool that supports SimCon, the ExSpect functional language is used for manipulating the attribute values of the simple objects. Moreover, we have implemented SCOM and SCA on top of this functional language. This also gives us the possibility to define some places and transitions still by means of this functional language. This is useful in some cases, e.g. for control flow modelling.

In an integrated SimCon specification, one wants to verify whether a constraint that has been defined in the SCOM part of the specification will always be kept invariant by the net part. As it is often difficult for human beings to perform such verification activities, we provide a means of automatic verification of constraints. This method, the Prime State Method, works for a certain class of constraints, the so-called compositional constraints. The method works as follows. First, a way of decomposing a SimCon-instance into so-called prime states is defined. We have proved that for each compositional constraint and each expression, there is only a finite number of prime states. The main theorem of the chapter regarding automatic verification states that it is necessary and sufficient to check all prime states when one wants to prove that an SCNET leaves a certain constraint invariant (or not).
Finally, we have described some guidelines for applying the SimCon approach. One specific way of using SimCon has been called the IOM layer, for Integrated Object Model. In this layer, a system is modelled in terms of objects with static properties (attributes and references) and a life cycle. Moreover, there is a level at which the processes are defined; at this level, the interaction between several object types is modelled.

We have illustrated the use of SimCon by means of the TransCorp case that has been defined in the context of the ISDF project.

During our research, we have also worked on implementing our ideas in a tool. We have done this on top of ExSpect, in order to avoid implementing a tool from scratch. In a future implementation effort, the tool will have to be finished.

Systems design in general and information systems development in particular have become more difficult during the last decade. In our opinion, integrated approaches that cover both static and dynamic aspects and that both have a formal semantics and are easy to use, are able to solve these problems. The SimCon framework is a good candidate, as we may conclude from the work that has been described in this framework. Our work is not yet ready. More experiments will have to be conducted in order to fine-tune our approach. Moreover, some implementation work will have to be finished. In the next section, we mention the most important points for further work.

### 7.2 Further Research

In the previous chapters, we have encountered several subjects for further research. Actually, there are three main streams for future work. We deal with them in this section (in an arbitrary order).

First, the presented theory has to be fine-tuned by applying it to realistic problems. This may result in (minor) changes in the object model, the algebra and/or the net model. Also, the graphical conventions of our framework may be changed and/or extended. In our opinion, the SimCon approach offers a well-balanced integrated model for (information) systems design. On the other hand, we would not be surprised if practical experiences would give rise to some changes, especially concerning the appearance of SimCon.

As we have recognised, application of SimCon onto real-life problems may also lead to the introduction of an additional modelling layer. This layer should be used then in the earliest phase of system analysis. At this layer, we would make no distinction yet between active and passive objects. Only at a later stage, this diagram would be worked out into a SimCon schema covering the static part and a SimCon net representing the dynamic aspects.

The second direction for further work is that of extending and completing the
SimCon tool. As has been sketched, the tool has been built upon the ExSpect tool. An editor for making simple and object schemas has been implemented. Moreover, a translation from SCOM/SCA to the functional language of ExSpect with its type system has been defined. However, a more close connection is desirable; eventually, the tool has to support the definition of the places in the net by allowing to click on a container type in the object schema. Also, it could be investigated whether it would be useful if more constraints could be defined graphically. For formulating the constraints that cannot be defined graphically, a mixture of SCA and the ExSpect functional language can be used; future investigations have to prove that this way of formulating constraints is satisfactory. Finally, the automatic constraint verification method from Chapter 5 has to be implemented in the tool.

This means that the algorithm for generating prime states has to be made concrete. The mechanism to check whether a state satisfies a constraint, can be built directly upon the ExSpect functional language.

We also have to evaluate the use of the IOM layer. It is likely that this layer requires some specific tool support, as has been sketched in Chapter 6.

The third direction for further research is the fine-tuning and completion of the theory for automatic constraint verification from Chapter 5. There are not yet many techniques for automatic data constraint verification. However, we think that this research field is very interesting and that it can help designers in making better systems.

In the context of our research, the following issues have to be investigated. First, we should try to give a static classification of the notion of a compositional constraint. It may be difficult to give a complete classification, but it will be possible to distinguish some interesting subclasses, like the existential constraints we mentioned in Chapter 5. Moreover, it would be interesting to know more about the complexity of our method, in terms of the sizes of the expressions and the constraints to be checked. Also, optimisation should be possible: the upper bound for Algorithm 5.2.15 is correct, but we presume that it is larger than strictly necessary to obtain correct results. As has been mentioned in the previous paragraph, tool support for the Prime State Method also has to be developed as yet. This will also give rise to research for reducing the complexity of our method (even further), in order to make it suitable for solving realistic problems.
Bibliography


ANDRIES, M. [1990], De Expressieve Lracht van GOOD: een Graph-Oriented Object Database Model (in Dutch), Master’s thesis, University of Antwerp.


BATTISTON, E., AND F. DE CINDIO [1993], Class-Orientation and Inheritance in Modular Algebraic Nets, Proceedings of the SMC Workshop on Petri Nets and the Object-Oriented Approach, IEEE.


CHRISTENSEN, S., AND J. TOSKIV [1993], DesignBeta V2.0.1 - BETA codesegments in CP-nets, OØCPN 5, Århus University.


Notes in Computer Science 254, Springer-Verlag, 142–167.
Post, R.D.J. [1993], private communication.
Rossum, J.P.H. van [1992], Hoe goed is GOOD? (in Dutch), Master's thesis,
Eindhoven University of Technology.


SIBERTIN-BLANC, C. [1985], High Level Petri Nets with Data Structure, European Workshop on Application and Theory of Petri Nets, Venice, Italy.


Index

ψ, 115
≜, see equality modulo $\mathcal{G}$
≤, 40
≈, 114
#, 51

$\mathcal{A}$, 155
AN, 38
analysis phase, 13
arc, 98
AS, 38
AS, 40
AT, 38
atomic identity, 95
attribute, 31, 32, 38
transitive closure, 40
automatic verification, 20, 105ff.
introduction, 105–107, 110–111

bottom-up, 138

C, 36, 42
CC, see Container Creation
child function, 96
CM, see Container Modification
composition of prime states, 111, 114
minimal, 115
compositional constraint, 110, 112, 119
constraint, 34, 105
bijectivity, 35
compositional, see compositional constraint
data, 106
deontic, 149
dependency, 119
dynamic, 105

existential, 119
functionality, 35
injectivity, 35
process, 106
static, 105
surjectivity, 35
time, 106
totalness, 35

Container Creation, 65
Container Modification, 66
container object, 18, 35, 42
aggregation, 77
container object universe, 36, 42
container type, 18, 36, 41
continuation, 95, 96
CS, 41
CT, 41

δ, 67
data-oriented approach, 138
DD, 38
design phase, 14

$E$, 156
equality modulo $\mathcal{G}$, 95, 97
equality modulo C and $E$, 120
event set, 100
event time, 100
existential constraint, 119
ExSpect, 16, 20, 25, 43, 103
integration with MOVIE, 26
integration with NIAM, 26

$F$, see child function
FA, 72
fact, 24, 108

171
formal semantics, 15
FP, 72

G, see continuation
GOOD, 78–86
GOOD operators in SimCon, 81–86

horizontal integration, 15

I_B, 51
if [ ], 53, 57
identification set, 95
identification structure, 95
impact, 121, 122
information system, 13
inheritance, 33, 39
inheritance structure, 39
initial identity, 95
initial marking, 99
integrated approach, 138, 142
object-oriented approach, 139
Integrated Object Model, see IOM
integration, 25, 31, 47
IOM, 148
action, 148
action net projection, 155
assignment step, 149
behaviour perspective, 149, 150
formalisation, 154
data perspective, 149
formalisation, 154
event, 150
event net projection, 156
interaction step, 149
life cycle, 148, 150
life cycle net projection, 156
object, 148
process net projection, 156
process perspective, 149, 152
formalisation, 154
projection functions, 155
role, 148
SCNET for γ, 155
triggering constraint, 149

IS, 39
ISDF, 16, 139

kernel, 48, 58

L, 156
LN, 41
location, 35, 41

message, 23
method, 23
MOVIE, 26

name clash, 33
non-determinism, 50, 68, 87
non-recursive expression, 112, 113

of [ ], 53
G, 58
object identity, 31, 95–98, 100–102
object schema, 18, 36, 37, 41
object vs. value-orientation, 32
object-oriented approach, 139
one-generation state, 97, 100, 102
overriding, 33

P, 156
P^F, 124
P-invariant, 108
Petri net, 19, 90, 91
pick, 75
place, 19, 91, 98
prime SimCon-instance, see prime state
construction, 122
prime state, 20, 110
Prime State Method, 105, 112ff.
complexity, 125
process-oriented approach, 137
PROOFS, 26

RA, see Recursive Application
RC, see Reference Creation
RD, see Reference Deletion
reaction function, 98, 99
realisation phase, 14
Index

Recursive Application, 66
reference, 31, 39
transitive closure, 40
Reference Creation, 61
Reference Deletion, 62
referential integrity, 106
relationship, 31
requirements, 14
requirement analysis phase, 14
restriction, 113
inverse, 114
RF, 39
RN, 39
root constraint, 37
RS, 39
RS, 40

SD, see simple object universe, 40
SSD, 36, 42
SC, see Simple object Creation
SCA
introduction, 18–19, 47–50
kernel, see kernel
SCA\(_G\), 18, 68–71
SCNET, 98
formal definition, 98–102
introduction, 19, 91–95
tool support, 103
SCCM
formal definition, 38–42
introduction, 17–18, 29–38
tool support, 42–44
SD, see Simple object Deletion
selective expression, 73
SimCon
modelling guidelines, 137
SimCon Algebra
expressive power, 78ff.
introduction, 18–19, 47–50
SimCon Net Model
formal definition, 98–102
introduction, 19, 91–95
SimCon Object Model
formal definition, 38–42
introduction, 17–18, 29–38
tool support, 42–44
SimCon-instance, 50
cardinality, 51
equality, 51
minimal, 115
prime, 115
SimCon/IOM, 148–156
simple object, 18, 31, 33, 40
Simple object Creation, 59
Simple object Deletion, 60
simple object selector, 49, 51–57
with negation, 75
simple object universe, 34, 40
simple schema, 18, 33, 34, 38
simple type, 18, 33, 38
size, 121
of a constraint, 121
of a selector, 121
sos, see simple object selector
ST, 38
state space, 36, 42
state space explosion, 107
subnet, 20
subtype, 40
T-invariant, 108
target system, 13
time point, 41
time-stamp, 35
token, 19, 91
top-down, 138
TP, 41
Tr, see transition relation
transaction, 107, 133
sequential, 133
with repetition, 135
TransCorp, 139ff.
transition, 19, 91, 98
firing, 91
occurrence, 91
transition function, 100
transition relation, 100
transition time, 100
transitive closure of a relation, 76

tree constraint, 37

\( v( I_i \)\), 53, 55

VA, see Value Addition

validation, 14, 35

Value Addition, 63

Value Deletion, 64

value selector, 49, 51–57

value vs. object-orientation, 32

VD, see Value Deletion

verification, 14, 16, 35

vertical integration, 15

vs, see value selector
Samenvatting

De laatste jaren is de complexiteit van geautomatiseerde systemen in het algemeen en informatiesystemen in het bijzonder, enorm toegenomen. Dit geldt zowel voor de statische eigenschappen van zulke systemen -“de structuur” - als voor de dynamische aspecten - “het gedrag”. Hierbij kan bijvoorbeeld gedacht worden aan reserveringsystemen voor vliegtuigstoelen of aan systemen die electronisch betalen mogelijk maken.

In de literatuur zijn veel modellen beschreven om hetzij statische, hetzij dynamische aspecten van een systeem te beschrijven. In de zogenaamde object-georiënteerde (OO) benadering probeert men zulke modellen te combineren. Sommige van deze OO modellen zijn intuïtief maar niet formeel, waardoor niet vastgelegd wordt wat er nu precies gemodelleerd wordt. Modellen die wel een formele semantiek hebben, staan vaak te ver van de gebruiker af. Het beschreven onderzoek wil voor deze problematiek een oplossing bieden.

In dit proefschrift wordt het zogenaamde SimCon raamwerk beschreven. De eerste pijler hiervan is het SimCon Object Model (SCOM). Dit is een eenvoudig object-georiënteerd data model dat bestaat uit twee niveaus: het eerste voor het beschrijven van de globale toestandssituatie van een systeem en het tweede om distributie-, locatie- en tijdeigenschappen vast te leggen. Bij dit object model is een manipulatietaal gedefinieerd: de SimCon Algebra (SCA). Naast een formele definitie van SCA bevat het proefschrift ook een aantal eigenschappen van de algebra, o.a. omtrent haar expressieve kracht. Om de gebruiker een goede ondersteuning te bieden, wordt zowel voor het object model als voor de algebra een grafische versie geïntroduceerd.

Vervolgens wordt beschreven hoe object model en algebra geïntegreerd kunnen worden met een Petri net model, in dit geval het Petri net model van ExSpect. De toestandssituatie van zo’n Petri net wordt dan beschreven met het SimCon Object Model, terwijl SCA gebruikt wordt voor het beschrijven van de functionaliteit van de transities in het net. Het zo verkregen geïntegreerde model (SCNET) biedt een goed evenwicht tussen gegevens- en procesmodellering, is intuïtief duidelijk en heeft een formele semantiek. Dat laatste maakt executeerbare specificaties en daardoor
een vorm van validatie mogelijk. Het object model en de algebra zijn uitgedrukt in de functionele taal van ExSpect, waardoor de tooling-faciliteiten van ExSpect beschikbaar komen. Hierdoor is het vrij eenvoudig een tool voor SimCon te creëren, waardoor daadwerkelijk ervaring met het werken met SimCon opgedaan kan worden. Het gebruik van SimCon wordt geïllustreerd aan de hand van een case. Ook wordt middels een aantal richtlijnen aangegeven hoe SimCon in de praktijk gebruikt zou kunnen worden.

Tenslotte is aandacht besteed aan formele verificatie van in het raamwerk ontwikkelde specificaties. Hierbij gaat het er om dat geverifieerd kan worden of constraints die in een object schema gedefinieerd zijn, in het bijbehorende Petri net invariant gelaten worden. Bestaande verificatietechnieken doorzoeken hiertoe de toestandsruimte van het Petri net. Deze is vaak echter te groot en meestal zelfs oneindig groot. De in dit proefschrift beschreven techniek reduceert de zoekruimte tot een eindige en relatief kleine hoeveelheid te controleren primitieve toestanden. Hierdoor wordt automatische verificatie mogelijk. We bewijzen dat deze constructie “nodig” en “voldoende” is voor een bepaalde klasse van constraints.
Curriculum Vitae


Per 1 januari 1990 werd hij aangesteld als onderzoeker in opleiding, binnen het interuniversitair NFI-project Information Systems Description Formalisms (ISDF). Hij werd gedetacheerd bij de sectie Informatiesystemen van de TUE. Het onderzoek dat in dit kader uitgevoerd is, is beschreven in dit proefschrift. De lijst van artikelen die hij hierover gepubliceerd heeft, is terug te vinden op pagina 179 e.v.

Sinds enige maanden is de auteur werkzaam als Universitair Docent binnen de faculteit der Informatica van de Universiteit Twente (UT). In de database-groep van prof.dr. P.M.G. Apers werkt hij hier mee aan het opstarten van de nieuwe studierichting Bedrijfsinformatietechnologie (BIT). Zijn onderzoek zal zich gaan richten op de problematiek rond interoperabiliteit van informatiesystemen binnen het Esprit-project TransCoop (Transaction Management Support for Cooperative Applications). Ook is hij in dit kader verantwoordelijk voor de ontwikkeling van het college Productie-automatisering.

Huidig adres:
Peter Verkoulen
Universiteit Twente
Faculteit der Informatica
Postbus 217
7500 AE Enschede

c-mail: P.Verkoulen@cs.utwente.nl
List of Publications


Stellingen

behorende bij het proefschrift

INTEGRATED INFORMATION SYSTEMS DESIGN

An Approach Based on Object-Oriented Concepts and Petri Nets

van

Peter Verkoulen

Eindhoven
17 december 1993

2. In het licht van de vorige stelling geldt voor geïntegreerde methoden en technieken voor systeemontwikkeling: het geheel is meer dan de som der delen.

3. In een datamodel dat gebruikt wordt bij het ontwikkelen van gedistribueerde systemen, is het belangrijk dat onderscheid gemaakt kan worden tussen de situatie waarin twee entiteiten toegang hebben tot elkaars kenmerken en het geval waarin ze elkaar slechts kennen, maar waarbij hun eigenschappen afgeschermd zijn.

De uitspraak in [2] die zegt dat in een conceptueel model geen onderscheid gemaakt mag worden tussen referenties naar objecten en de objecten zelf, impliceert dan ook dat men in dat artikel alleen in centrale databases geïnteresseerd is.

[1] dit proefschrift

4. Wanneer een object model en een Petri net model geïntegreerd worden, dient een mechanisme beschreven te worden voor het genereren van unieke objectidentiteiten voor nieuw gecreëerde objecten. Dit mechanisme moet voldoen aan het localiteitsprincipe dat inherent is aan Petri netten.

De manier van object-creatie zoals beschreven in dit proefschrift, voldoet aan deze eis (zie ook [2,3]).

[1] hoofdstuk 4 van dit proefschrift
5. In hoofdstuk 5 van dit proefschrift wordt een methode beschreven voor het automatisch verifiëren van constraints door een toestand van een systeem op te delen in een aantal primitieve bouwstenen en deze te controleren.

Deze methode kan zodanig geconcretiseerd worden dat implementatie in een tool voor systeemontwikkeling mogelijk wordt, waardoor de methode praktisch toepasbaar zal zijn.

6. De vraag “Hebben Linda en Sander Bakker een vergaderingbijgewoondover het dialoog-systeem?” kan formeel ontkennend beantwoord worden als zo'n vergadering helemaal nooit heeft plaatsgevonden.

Een intelligent electronisch archief dat vragen in natuurlijke taal beantwoordt, dient in zijn antwoord de onjuistheid van een dergelijke vooronderstelling tot uitdrukking te laten komen.


7. De titel van [1] laat zien dat “de wet van de uitgesloten derde” (tertium non datur) uit de logica niet altijd toepasbaar is zonder informatie te verliezen.


8. Het instellen van nieuwe studierichtingen die liggen op het grensgebied van de informatica en de bedrijfswetenschap heeft alleen zin wanneer ze niet slechts bestaande activiteiten onder een nieuwe noemer brengen. De studierichting Bedrijfsinformatietechnologie aan de Universiteit Twente voldoet aan deze eis.


10. Zolang tegenover het recht op studiefinanciering niet de plicht staat om een redelijke hoeveelheid tijd en energie aan de studie te besteden, is het beter om de studiefinanciering te laten ressorteren onder het ministerie van Sociale Zaken en Werkgelegenheid.
11. Eigenwijsheid en eigen wijsheid zijn beide noodzakelijke eigenschappen van een goede wetenschapper.

12. De enige rechtvaardiging voor het schrijven van een proefschrift in het Nederlands wordt gevormd door het feit dat het erin beschreven onderzoek volgens de promovendus internationaal niet bijster interessant is.

13. Iemand die een fanfare een "hoempa-orkest" noemt, geeft niet zo zeer blijk van muzikaal gevoel als wel van culturele armoede.