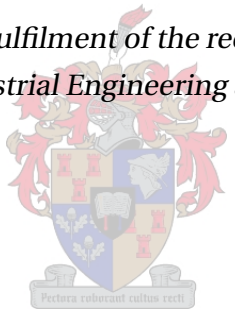
Business Process Modelling Using Model Checking And The Theory Of Constraints

by

Maghiel Jock Odendaal

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Industrial Engineering at Stellenbosch University



Department of Industrial Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Supervisors:

Prof. Dimirti Dimitrov Mr. Corné Schutte Dr. Jaco Geldenhuys

March 2010

DECLARATION

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the owner of the copyright thereof (unless to the extent explicitly otherwise stated) and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

Signature: .....
M. J. Odendaal

Date: 22 February 2010 .....

ABSTRACT

Business Process Modelling Using Model Checking And The Theory Of Constraints

M. J. Odendaal

Department of Industrial Engineering,
University of Stellenbosch,
Private Bag X1, Matieland 7602, South Africa.

Thesis: MScEng (Industrial)

March 2010

Concurrent and distributed business processes are becoming the norm in many organisations. Current modelling techniques do not address the problems faced by concurrent business processes sufficiently. We show how model checking is applied to business processes to prove behavioural properties to address the aforementioned shortcomings.

A method of abstraction is required to construct business process models that can be model checked. In this thesis we show the suitability of the Logical Thinking Process as an abstraction tool.

We call the combination of the Logical Thinking Process and model checking the Complexity Alleviation Method (CAM). We apply CAM to two well-known supply chain and manufacturing problems, and insightful results are obtained. This leads us to the conclusion that CAM allows for the quicker modelling of business processes, as well as providing problem-specific and proven solutions in a manner not possible with simulation or other techniques.

UITTREKSEL

Besigheidsproses-modellering Met Model Verifikasie En Die Teorie Van Beperkinge

("Business Process Modelling Using Model Checking And The Theory Of Constraints")

M. J. Odendaal

Departement Bedryfsingenieurswese,
Universiteit van Stellenbosch,
Privaatsak X1, Matieland 7602, Suid Afrika.

Tesis: MScIng (Bedryfs)

Maart 2010

Gelyklopende en verspreide besigheidsprosesse word 'n alledaagse verskynsel in menigte instansies. Huidige modelleringstegnieke is nie in staat om die probleme geassosieer met gelyklopende besigheidsprosesse aan te spreek nie. Ons wys hoe model model verifikasie ("model checking") toegepas word op besigheidsprosesse om gedragseienskappe te bewys en sodoende die voorgenoemde tekortkominge aan te spreek.

'n Metode van abstraksie word benodig om besigheidsprosesmodelle, wat verifieerbaar is, te konstrueer. In hierdie verhandeling word die geskiktheid van die Logiese Denkproses ("the Logical Thinking Process") as abstraksie gereedskap aangetoon.

Ons noem die kombinasie van die Logiese Denkproses en model verifikasie Kompleksiteitsverligtingsmetodologie (CAM). Ons pas CAM op twee welbekende aanbodketting- en vervaardigingsprobleme toe en insiggewende resultate is verkry. Dit lei ons tot die gevolgtrekking dat CAM vinniger konstruering van modelle te weeg bring, sowel as probleem spesifieke en bewysbare oplossings verskaf wat nie moontlik is met simulasie of ander tegnieke nie.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to the following people and organisations:

Mma Shieling Financial Services

The Hoffmann Think-tank

Kapp Pharmaceuticals

The de Waal Graduate Centre

Geldenhuis Hazmat Consultants

DMD Lingerie

DEDICATIONS

To the man who was kind enough to lend me half his brain

CONTENTS

LIST OF FIGURES

LISTINGS

NOMENCLATURE

Glossary

CAM	Complexity Alleviation Method
CLR	Categories of Legitimate Reservation
CRT	Current Reality Tree
DBR	Drum Buffer Rope
FRT	Future Reality Tree
FSM	Finite State Machine
GUI	Graphical User Interface
JIT	Just In Time
LTL	Linear Time Logic
PROMELA	Process Meta Language
S-DBR	Simplified Drum Buffer Rope
SKU	Stock Keeping Uint
SPIN	Simple PROMELA Interpreter
TOC	Theory Of Constraints
UDE	Undesirable Effect
WIP	Work In Progress

Symbols

□	Always
◊	Eventually
!	Not
&&	And
||	Or

CHAPTER 1

INTRODUCTION

CONCURRENT and distributed business processes are becoming the norm in many organisations. This adds speed and reach in an organisation's operation but does come at a potential price. Synchronisation and fault identification are far more complicated in these circumstances when compared to traditional serial operations. This problem has, however, been addressed with a technique known as model checking, and we will show how to address the challenges faced in corporate operations, and the benefits that are derived from using this technique.

1.1 Motivation

Organisations are required to respond quickly to changes in their environment, whilst the environment itself is also in a state of constant flux. Environments consist of global markets that require global operations and reach. Products and services are required to appeal to a global market but have to satisfy local regulations and preferences whilst adhering to cost constraints. Greater integration throughout the supply chain is demanded while outsourcing services. Product turnaround times are constantly shrinking requiring more to be done in less time.

One way of dealing with change is with modelling and simulation. The use of models allows us to conduct "what if" scenarios, without the cost and implications associated with implementing changes in a real-world system. The use of models and simulation has allowed for faster, more cost effective, and reliable solutions to be produced. Business processes and protocols (the focus of this thesis) have benefited as well from the use of simulation. In the context of business process behaviour investigation, modelling and simulations have two significant drawbacks: firstly, the inability of a simulation to *prove* model properties and, secondly, the difficulty of modelling concurrent processes

accurately.

The first problem stems from how information regarding behaviour and system properties is gathered with simulation. A model is subjected to a simulation run, which is a random sequence of events generated to imitate real conditions. This run is then sampled to gain the required information. Simulation produces statistically inferred information on the behaviour of a system by taking a significant subset of all possible model executions and reporting these figures back to the user. Though sufficient to make comments in general, simulation is not capable of proving model properties [2; 3], simply because it did not observe every instance of model behaviour that is possible. In the event that not all eventualities are observed there always remains a possibility that a major anomalous event can occur. We are capable of identifying and eliminating frequently occurring problems in systems. So it is these infrequent and anomalous events that we desire to eliminate from the system; something we are not capable of doing with current methods.

Concurrent processes are difficult to prove as error free with traditional methods. This poses a serious problem to business processes, which are becoming ever more concurrent. This problem is not always fully appreciated. We deal with issues of concurrency on a daily basis, competing for resources like food and checkout clerks, negotiating traffic intersections, or competing for parking slots. There are written, and unwritten rules on how to approach these concurrent problems, but they are, however, not foolproof. For example, had the rules of driving[1] been foolproof there would not be any vehicle accidents on our roads. As business processes become more concurrent it is necessary that the rules that govern them are able to deal with this increased complexity. In addition to this, certain systems require protocols and rules that function flawlessly.

1.2 Solution framework and components

The issues with the use of simulations to prove model properties formally and address concurrency problems have been raised. A method used to solve these problems is called model checking. Instead of exploring only a subset of the model's operations space, like simulation, model checking examines all eventualities. By doing so it does not infer properties but proves them formally. In addition to this, model checking is well suited to the validation and verification of concurrent processes and protocols.

The benefits of model checking do, however, come at a price. Since all possible iterations of model execution and behaviour are evaluated, as opposed to what occurs with simulation, the verification of models with model checking are computationally expensive and memory intensive.

It is of great importance to construct models that are as abstract as possible. Normal model checking focus areas are hardware, circuits, and data communication protocols; as a result, the specification language used to describe these protocols is, as formal and as rigid as the applications it is usually

[1] Driving is a concurrent activity if more than one vehicle is involved, and this is typically the case on any road where the rules needs to be obeyed.

used for. Despite this, business processes can, and have been specified as well as verified successfully using formal languages.

However, due to its computational overhead it is generally prohibitive to model most business processes in their totality. In addition to this, organisations consist of more that just business processes. Though business processes can be specified formally, the additional components that make up an organisation may not. These components my involve entities such as emotions and company politics; issues that do not translate into formal protocols. A means is therefore needed to abstract the formal specification from an informal business process so that it can be model checked.

This abstraction is achieved with the use of the Logical Thinking Process. The Logical Thinking Process forms part of the Theory Of Constraints (TOC) tool-set and is used to identify the root cause of problems observed in systems and organisations. It takes both the formal and the informal aspects of the systems and the associated problems and reduces them into a single focal point for improvement. This output, provided by the Logical Thinking Process, is both formal and highly abstract in nature, making it ideally suited for use in conjunction with model checking.

1.3 Objectives

The aim of this work is to apply model checking to business processes in order to prove the behavioural properties of the proposed solutions. The verification and modelling of concurrent business processes are also major objectives since simulation does not address the problem sufficiently.

A method of abstraction is required to build verifiable models to accomplish the above-mentioned objectives. We will show the suitability of the Logical Thinking Process as an abstraction and analysis tool. We will also show it being able to represent intricate and complex processes in a highly abstract manner suitable for model checking. In order to be viewed as a success, the combination of model abstraction and verification must yield new insights into established and understood solutions, as well as demonstrate additional properties that are not present under current simulation methods.

1.4 Thesis outline

The rest of this document is constructed as follows. Chapter 2 describes the Logical Thinking Processes as well as components of TOC used in the abstraction of models. Chapter 3 provides a brief overview on model checking and describes concepts of the field that will be used in this thesis. Both the Logical Thinking Process and model checking are then combined into a unified methodology for problem identification, abstraction, solution generation model construction, and verification in Chapter 5. This is followed by two implementations of the methodology in Chapters 6 and 7.

CHAPTER 2

THE THEORY OF CONSTRAINTS

2.1 Short overview of characteristics

To understand the ideas presented in this thesis, a brief and selective overview is given below of the Theory of Constraints (TOC) and related tools.

TOC is a management philosophy that teaches how to improve an organisation's effectiveness in reaching its goals by understanding and eliminating the underlying causes that limit a company in achieving its objectives. Originally developed by Goldratt, it became popular through a series of "business-based" novels [4; 5; 6; 7; 8]. It has gradually moved from its original focus on production and operations management (a so-called *hard method*) to a general approach that covers all aspects of business (now a *soft method*). It has been described by Goldratt as "an overall theory for running an organisation" [9, p. 453]. Although it may appear mercantile, unscientific, or common-sensical, TOC has found a place in the academic Operations Research/Management Science (OR/MS) communication; Davies [10] and Rahman [11] survey the work around it.

2.1.1 TOC background

The aim of TOC is business improvement. It initially focused on the improvement of manufacturing, as set out in the business novel The Goal. Goldratt describes in this novel the assumptions and principles on which later TOC work is built.

Every system has an objective, and every system has constraints. To improve a system the objective or "Goal" of the system must be defined, as well as its constraints. It is important to realise that constraints that inhibit a system are *contextual* in terms of the system goal. This means that the lack of comfortable seating in Boeing 747 passenger jets is a constraint, whilst the same situation in

a Boeing 747 cargo plane is not. According to TOC, the biggest improvement in a system comes from eliminating the largest constraint, or that which inhibits the system the most from achieving its goal.

Improving a system with TOC uses the assumption that systems are subject to the laws of cause and effect. What happens in one section of a system has an impact on the other sections of the system. More importantly, the impact is —to a large extent— predictable and even quantifiable. If this assumption holds, multiple negative effects can be attributed to a common or root cause, and TOC can improve performance.

2.1.2 The five focusing steps

Improvement in TOC is viewed as a continuous process, which is formalised in an algorithm known as the "five focusing steps":

1. Identify the most serious constraint.
2. Decide how to exploit the constraint.
3. Subordinate all other processes to the decision in step 2.
4. Elevate the constraint.
5. Re-evaluate the situation by returning to Step 1. Do not let inertia become the constraint.

The steps are explained in The Goal with the example of a father taking his son's scout troop camping, and solving the problems faced during the hike. The first aim of the hike is reaching the campsite safely, and on time. From the outset some kids start to race ahead of the troop and some lag behind. This makes supervision of all the children difficult, and the pace is too slow to reach the campsite before dark.

The first step is to identify the constraint. For example, in the case of the troop the constraint is Herby, the slowest of all the kids. Exploiting the constraint means ensuring that Herby keeps walking. Every moment that Herby stops is a moment that the troop cannot make up, since he is the slowest. Subordinating all other processes, Herby is placed at the front of the troop. This ensures that all kids walk together. Elevating the constraint further, Herby's backpack is removed and the weight is distributed amongst the rest of the troop. Herby now sets the pace for the troop and when he walks faster, everybody else walks faster. All effort has focused on Herby until he is no longer the slowest kid in the troop, and now the process can start over again.

2.2 The Logical Thinking Process

In this thesis the component of TOC used most frequently is the Logical Thinking Process. In essence, it is a form of formalised common sense to graphically describe the cause-and-effect relationships within a system.

The purpose of the Logical Thinking Process is to describe different states that the system is, or could be, in (either the current, past or future states of the system), in order to identify, exploit, promote and elevate the largest constraint.

It was first introduced in It's not Luck and later further formalised in Thinking For A Change [12]. According to Dettmer [13], the entire Logical Thinking Process comprises six diagrams each, built in the same way. The construction method is explained in Section 5.2.1. The diagrams are:

1. the Intermediate Objectives Map,
2. the Current Reality Tree,
3. the Evaporating or Conflict Cloud,
4. the Future Reality Tree,
5. the Prerequisite Tree, and
6. the Transition Tree.

In this work we use only the Current Reality Tree (CRT), the Conflict Cloud, and the Future Reality Tree (FRT).

2.2.1 The Current Reality Tree

The CRT is used to answer the question "*what* to change", the Conflict Cloud and FRT "what to change *to*".

As its name suggests, the CRT diagram describes the current reality in the system that is being improved. The aim of the diagram is to locate the largest constraint in the system. This is done by taking three to five problems faced by the system and finding the relationships that exist between them.

This process relies on the assumption stated in Section 2.1.1 that multiple problems can be related to a common or root cause. It is this root cause that is the largest constraint and will be the focus of improvement.

Let us construct the CRT given in Figure 2.1 of the scout troop example. The problems faced were that it was difficult to supervise the kids, and that the campsite would not be reached before dark. These two undesirable effects (UDEs) are marked in red.

The first UDE, that the troop is hard to supervise, is due to the distance between Mike in the front, and Herby at the back. The reason for the distance is the combination of three things. Mike walks the fastest, Herby walks the slowest, and everybody is allowed to walk at their own pace. This results in a large distance between the first and the last kid, and makes the troop hard to supervise.

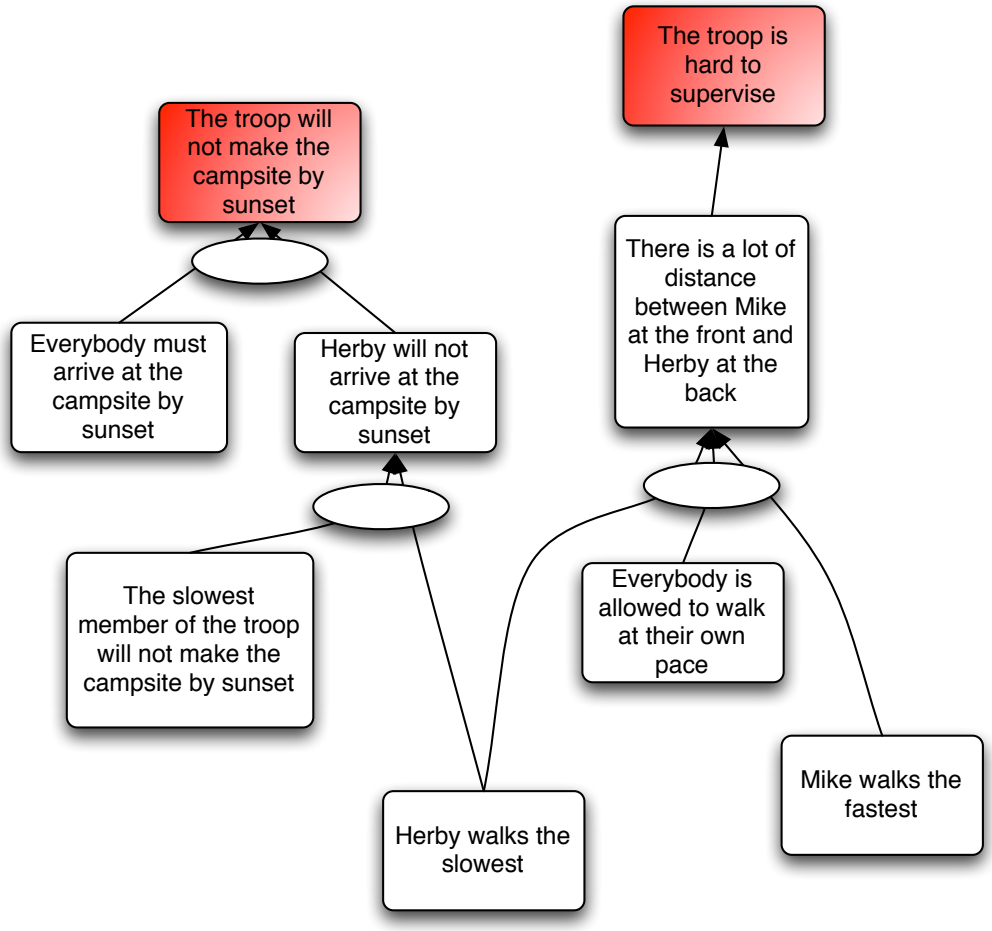


Figure 2.1: The CRT of the scout troop hike

Changing any of these three entities will impact the supervision UDE. Asking Mike to walk slower will reduce the space and make the troop easier to supervise. Slowing Herby down will, however, make the problem worse.

The second UDE is explained in the same way. It would not matter if most of the kids reach the campsite before sunset; unless all the members of the troop arrive before sunset, the whole troop has not arrived. So Herby must make the campsite before sunset.

The CRT answers "what to change" by identifying the entity that affects all UDEs. You will impact at least one UDE by touching any of the entities in the CRT, but only one (in this case) will affect them all. This is the root cause, and the focus of improvement. In the case of the scout troop example, the root cause is Herby's speed, and all efforts for improvement will centre around it.

2.2.2 The Conflict Cloud

Having identified the root cause as the speed at which Herby walks in the CRT, we now focus on "what to change *to*". A conflict arises when dealing with Herby's speed problem. This conflict is depicted in Figure 2.2.

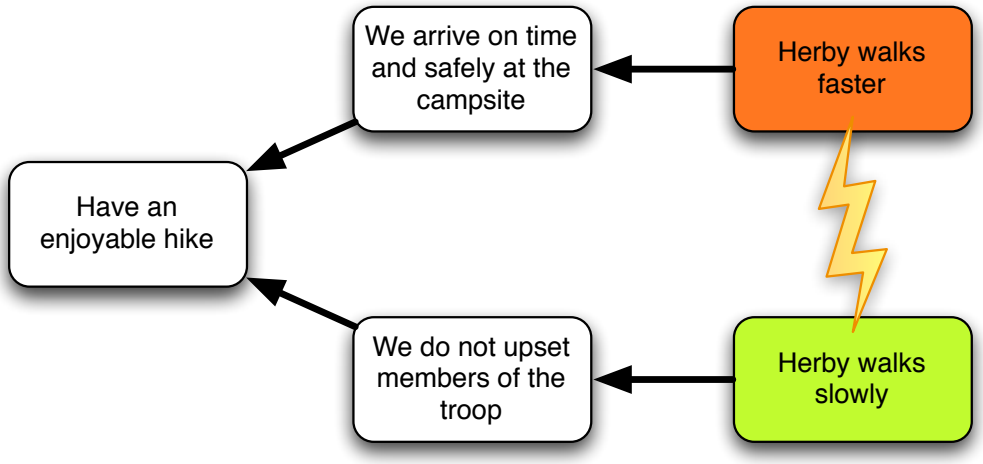


Figure 2.2: The conflict cloud of Herby's slow walking pace

The aim of the hiking trip would ultimately be to enjoy it. For that to happen the troop must arrive safely and on time at the campsite. This requires Herby to walk faster. Enjoying the hike is far easier if all of the members of the troop are happy. Placing undue pressure on Herby to walk faster, or on the other kids to walk slower can jeopardise this. So we must allow Herby to walk at his own pace.

This leads to a conflict. To enjoy the hike Herby must walk faster, and to enjoy the hike he must walk at his own pace. The aim is to find how to solve this apparent contradiction, and the key to the solution lies in the assumptions used to construct Figure 2.2. We assume that if we force Herby to walk faster he will become unhappy. This is possibly true. The question is how do we make Herby walk faster but still keep everybody happy?

Two possible actions can solve this problem. One, placing Herby at the front of the troop, and two, lightening Herby's load. This provides us with a feasible solution that will address all our UDEs identified in the CRT.

2.2.3 The Future Reality Tree

After having generated feasible solutions with the Conflict Cloud it is helpful to see what impact these solutions will have on the system.

Placing Herby in the front of the troop, as mentioned, ensures that Herby walks as fast as possible, since he sets the pace. Herby is also carrying a heavy backpack. Redistributing the weight between other members of the troop will speed Herby up automatically. Both of these actions move us closer to both our goal of having an enjoyable hike and removing the UDEs in the CRT.

We can depict this in the form of an FRT and an example of an FRT for the scout troop is given in Figure 2.3.

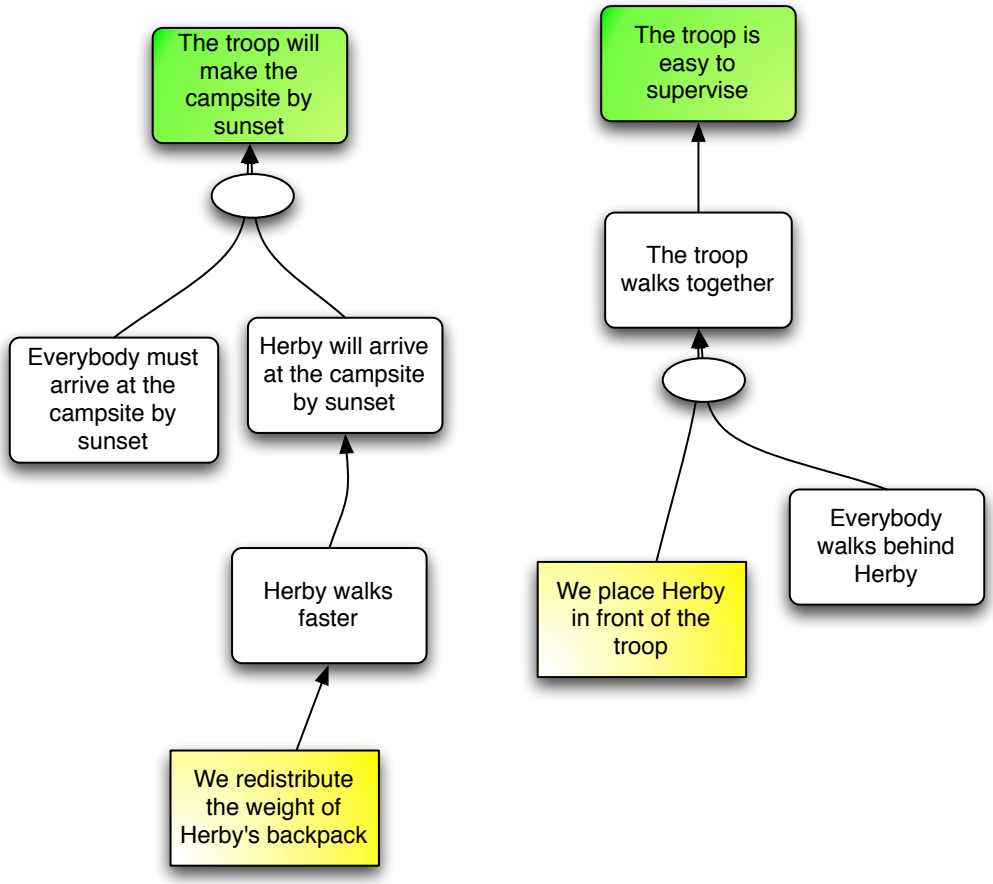


Figure 2.3: The FRT once we increased Herby's pace

Similar to the CRT, the FRT is used to describe the future state of the system. The FRT is used to assess both the positive and negative impact of a proposed solution before it is implemented. It is done to save effort during implementation and to prevent worsening of the current situation.

2.2.4 Necessity versus sufficiency

The Logical Thinking Process relies on the concepts of necessity and sufficiency. The difference between the concept can be described in terms of building a house. *Necessity* conditions would be steel, sand, cement and bricks; without these elements building cannot start. Their absence would be deemed as"show stoppers". Steel, sand, cement, and bricks are, however, not *sufficient* to build a home. Roofing, wiring and paint, as well as numerous other items are required before a structure is sufficient to be called a house.

The CRT and FRT are sufficiency structures; consequently we will not be focusing on the necessity issues in this thesis since their associated diagrams are not used in the scope of the work.

The Logical Thinking Process describes the process of constructing the above-mentioned diagrams in detail. The process, like the examples demonstrate, remains informal and subjective, yet reduces the system to a single specify-able focus point. It takes the informal aspects of the system into account, like the troops feelings and enjoyment of the hike, and processes them into concrete and tangible properties that can be modelled, reducing emotions and physical considerations into a single focus; in the case of this example, Herby's speed.

The creation of any model requires abstraction on the part of the modeller. As a result models are subjective representations of the perception and implementation preferences imparted to it by its creator. The representations that are formulated for the same reality will not be the same for different constructions with the use of the Logical Thinking Process. These representations will differ depending on who constructs them, and will vary from analysis to analysis depending on who performs it. Nevertheless, the guidelines provided by the Logical Thinking Process aids in the creation of faithful representations of reality in a manner that would not be possible without that use.

CHAPTER 3

MODEL CHECKING OVERVIEW

MODEL checking is used to prove that system models are error free. This chapter describes how model checking operates and how it establishes that a system is functioning properly. It follows Inggs [1] closely and describes what model checking is, what tools and languages we use in this thesis and how to apply them.

3.1 Introduction to model checking

It is important that systems function properly. A system that fails, or performs poorly, inevitably results in problems. These problems range from mild irritation to life threatening conditions, and can cause damage, cost money or lives. As a result, the effective identification of defects is important.

One of the main methods to identify defects and demonstrate the correctness of a system is by testing. Testing involves the observation of the behaviour of an active system under a specific set of conditions and inputs. Though effective under the right circumstances, testing has its limits. Testing is biased, especially towards frequently occurring problems or scenarios. To ensure that a system is defect free, all possible executions of the system must be checked under all possible conditions. Accomplishing this with an active system requires additional controls and computational overheads that make it infeasible. Testing is also difficult in concurrent systems. These systems may have subtle timing and synchronisation issues and total process control may not always be possible.

Model checking addresses these problems. It does not require a running system to operate. System design behaviour is modelled and interpreted without an implementation. It also covers all possible system execution paths, so inconspicuous and subtle defects are detectable. Model checking is especially useful for the evaluation of concurrent systems that are difficult (or even impossible) to

evaluate with more direct testing.

These properties allow model checking to be used early on in the development process. Therefore, model checking can detect even subtle defects in a specification without having a physical system implementation. This makes defects easier and cheaper to rectify.

3.2 The model checking process

In the past, model checking has been used mainly for communication protocols, concurrent processes and safety critical systems. These systems consist of interacting components that execute concurrently by exchanging data or messages. One can draw a parallel between the aforementioned systems and normal business processes. This is left for Section 3.5.

In order for a system to be verified, the model checker requires two separate components: firstly a formal specification of the system itself, defining its behaviour, structure and interactions and secondly, a formal specification of each desired or undesired system behaviour. With the model specification and behaviour requirements specified the model checker can evaluate if the specification adheres to the stated behavioural requirements. After the completion of the evaluation the model checker provides a "yes" or "no" response to whether or not the model provably adheres to these behaviour specifications. A "no" response is also complimented with a counter example on how the system failed.

It is important to note that the verification of a system for a given set of properties does not imply that the system is free from defects or potential malfunctions. All that the verification demonstrates is that the system specification provably adheres to the specified behaviour requirements. For example, verifying that a system is free from deadlock does not prove that the same system will be free from starvation. The primary aim of model checking is to detect errors that are difficult to find under normal circumstances. The verification of a system's adherence to certain properties improves confidence that the behaviour of the system will be correct.

Model checking proves the adherence of a system to a property or a set of properties by evaluating all possible executions that can take place. This is possible because such systems usually consist of a finite set of states.

Having discussed on an abstract level how model checking works, it is beneficial to take a closer look at how to define system behaviour, verification requirements and the verification thereof.

3.3 The model checker SPIN

There are numerous model checking tools available. In this work we will be using the model checker SPIN (*S*imple *P*ROMELA *In*terpreter)[14]. SPIN is an open source tool that has been successfully used to model check business processes; the final aim of this work. The combination of a free tool with

proven abilities to accomplish the intended task, in combination with locally available knowledge resulted in the adoption of SPIN as the model checker for this project.

SPIN is a verification system developed by Bell Labs in the eighties and nineties and is distributed freely on the Web.[1] It is also one of the widest used logical model checkers in the world, employed by both academia and industry alike. In 2002, ACM (the Association for Computing Machinery) awarded SPIN its *Software System Award*; its most prestigious accolade. This recognises SPIN as truly breakthrough software: systems such as UNIX, TEX, Smalltalk, Postscript, TCP/IP and Tcl/Tk, are all previous laureates of the award.

SPIN will be used as a black box solution that is able to verify a model's adherence to the specified properties. A deeper understanding of how this is accomplished is not required since the concepts presented in this work only require a high-level understanding of the tool. The inputs and outputs of the tool, however, require explanation. An authoritative overview of the tool can be found in the work done by Holzmann [14].

3.4 System behaviour specification

Often the behaviour of systems is represented by state-transition diagrams, which is a graphic representation of a finite-state machine (FSM). An FSM consists of the basic states that a system can find itself in and the actions that will result in the transition from one state to another. The state of a system is typically the combination of the states of the different subsystems, as well as the values of all global variables.

As an example of an FSM, let us examine the simple alternating bit protocol. The system is made up of two separate sub-processes; a sender and a receiver. They communicate via a single channel across which messages can be sent. This is depicted in Figure 3.1.

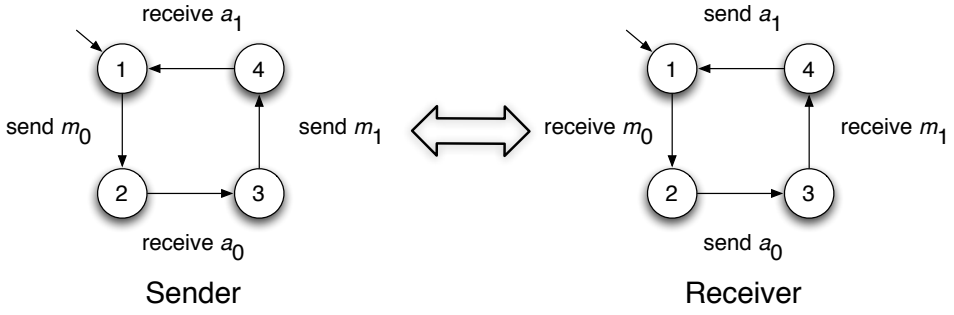


Figure 3.1: Sub-process state-transition diagrams of an alternating bit protocol [1]

[1] SPIN, as well as additional literature, can be downloaded from spinroot.com

Each process has four states, and the states are numbered. The processes are initialised and start off in state 1. From there the sender transmits m_0 (a message with the value 0) over the channel and moves to state 2. Upon receiving m_0 the receiver process moves over to state 2. Once in state 2 the receiver replies over the channel with a_0 (an acknowledgement of receiving the value 0 sent by the receiver). This in turn moves the sender to state 3, and so on. The communication channel is also "lossy" so messages that are transmitted are not always delivered.

Systems that have a finite set of states can be evaluated by SPIN. The combination of all these different states is called the global state space. Each state in the global state space contains three components: one each for states of the sender and receiver, and one for the contents of the communication channel (e in this case denotes an empty channel). The global state transition diagram is shown in Figure 3.2.

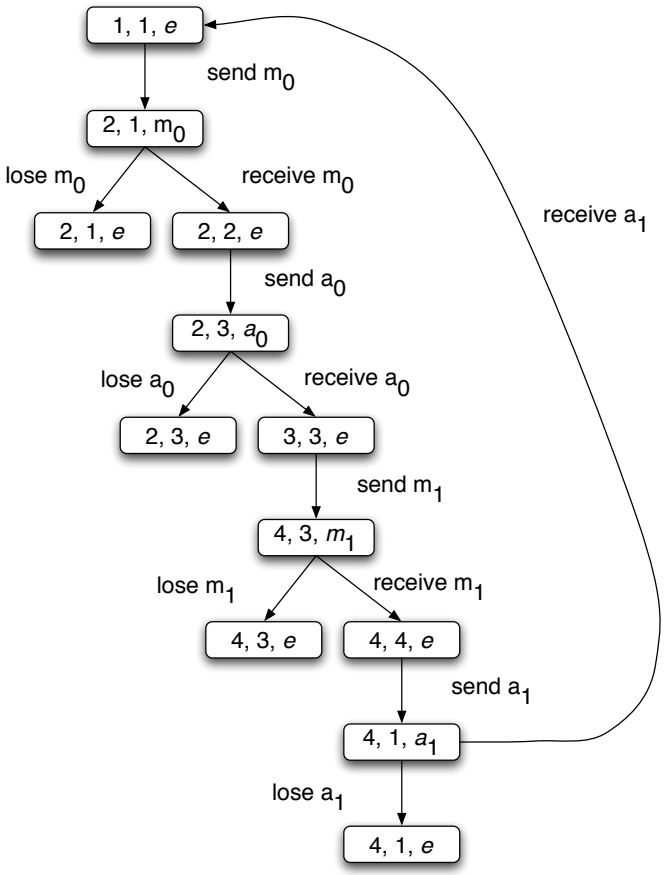


Figure 3.2: Global state-space diagram of the alternating bit protocol shown in Figure 3.1 [1]

To specify large systems accurately in FSM form would be impractical, or at least very difficult. High level formalisms are used to specify systems for model checking in a similar manner as pro-

gramming languages are used for software development. As a result, model checkers often provide high-level formalisms in order to specify model behaviour, including PROMELA, SMV, SAL, ESML and Rainbow. In this work we will use PROMELA.

3.4.1 PROMELA

PROMELA is an acronym for *Pro*cess *Me*ta-*La*nguage. The use of *meta* in the name of the specification language is significant, since it is designed to facilitate the abstraction of the system when modelling. The emphasis is placed on the modelling of process synchronisation and co-ordination, not how the computation of the process will be done. The structure of PROMELA is devoid of the concept of time or a system clock. There are also no floating point numbers and only a limited number of computational functions.

PROMELA models are made from the following components:

- asynchronous processes
- buffered and unbuffered message channels
- synchronisation statements
- structured data.

The component limitations mentioned above, and the lack of common functions and data types like floating point numbers, make an ordinary task of computing a square root impossible with PROMELA and SPIN. Modelling the behaviour of clients and servers in a network environment is, by contrast, a trivial operation. These attributes also aid in the use of PROMELA in the modelling of business processes, since it was not designed for computation, but for system specification.

3.4.2 Correctness properties

Correctness properties specify the desired and undesired aspects of a system. If a systems adheres to the desired aspects and shows itself to be free from undesired properties, it can be considered capable of meetings its design objectives. The verification of a system hinges on theses properties, and they themselves are as important as the model that is verified.

Correctness properties are separated into two main categories, namely safety and liveness properties. The basic distinction between the two is that a safety property defines that something undesirable will never happen. A liveness property on the other hand, states that something desirable will invariably happen.

SPIN's correctness properties are specified using temporal logic. There are two well-known types of temporal logic, namely branching time temporal logic and linear time temporal logic [1]. Computation Tree Logic is an example of the former and Linear Time Logic (LTL) of the latter. The examples

Table 3.1: LTL grammar[15]

f::=	**p**	
	| **true**	
	| **false**	
	| **(f)**	
	| **f binop f**	
	| **unop f**	
unop ::=	$\Box$	always
	| $\Diamond$	eventually
	| !	logical negation
binop ::=	U	strong until
	| &&	logical and
	| ||	logical or
	| $\rightarrow$	implication
	| $\leftrightarrow$	equivalence

in this document will, however, only use LTL, since it is the logic which PROMELA employs to specify correctness claims. Correctness properties are specified using LTL grammar. This grammar can be found in Table 3.1.

If one takes the alternating bit protocol depicted in Figure 3.2 as an example, a liveness property would be that in the event that the sender sends a message with the value 0, the sender will at some later point receive an acknowledgement of that message also with the value 0. This can be expressed in LTL as

$$\Box(m_0 \rightarrow \Diamond a_0). \tag{3.1}$$

A common safety property for concurrent systems is freedom from deadlock. The protocol given in Figure 3.2 does not include error detection. In the event that the channel loses a message the system is stuck in deadlock. This can be seen in the example by inspection. In the event that a message gets lost, the system will enter into one of the four states that have no outgoing edges, such as state $(2,1,e)$. If the system enters one of these states it becomes stuck, hence the term deadlock.

3.5 Model checking for business processes

The work as presented in this thesis applies model checking outside of communication protocol development and safety critical systems. Interesting application examples of model checking include work by Bäumler et al [16], subjecting medical guidelines to verification. Koehler et al [17] make a well-defined case regarding the need for model checking in business processes and the benefits that they can bring to a system in terms of reliability and flexibility.

Business processes contain communication protocols between departments, and often many of

the components of a business system is safety critical. Model checking offers benefits to both these issues and has therefore been applied to verify business processes in the past. The bulk of the work in the field was done by the Telematics Institute during the Testbed Project [18]. The aim of the project was to devise a method that would make model checking accessible to "managers"— or non-technical users —an audience that would otherwise be deprived of the benefits that model checking has to offer.

The Testbed Project resulted in the development of the BIZZDesigner software package, allowing a user-friendly GUI interface for business modelling using model checking. Eertink[19] and Janssen [18; 20] documents the challenges of design abstraction and graphical modelling of business processes for verification.

The approach to address the challenge of abstraction in business processes was done by means of the specification of a formal graphical language called Amber. Amber is a stricter specification than PROMELA; this is done to ensure that models remain of verifiable complexity. This does reduce the flexibility and type of models that can be verified, since they firstly have to adhere to Amber. In addition to this, BIZZDesigner does not allow for the specification of correctness claims directly in LTL. It has a specification builder built in that allows users to verify models only for one property at a time. These techniques make the benefits of model checking available to business process modelling, but do so in a limited manner.

3.6 Business process modelling and meta-modelling

There are numerous approaches to business process modelling and a wealth of work has been done in the field. Lu [21] presents a comparative survey on the field and divides the approaches of the field into two categories, graph-based, and rule-based modelling. The graph-based approach derives its roots from graph theory. The rule-based approach is based on formal logic. Model checking as it is used in this work comprises both these elements, and can therefore not be grouped with either of these approaches exclusively. It is in this duplicity that the potential of model checking resides when compared to other techniques.

Within the field of business process modelling numerous keywords are shared with model checking. Concepts like verification and validation do imply the exact same concepts in both domains. In this work we will use the concepts from the model checking domain.

Validation Validation is the process of ensuring that the model which is constructed is a true reflection of reality. Validation ensures that the specification drawn up for a product or project is a true reflection of what is desired. In the event that a customer ordered a jet-propelled aircraft, validation of the project would ensure that he/she does not receive a propeller variant.

Verification Verification is the process of ensuring that the constructed model is a true representation of what was intended to be designed. Verification ensures that the project is completed correctly. In a sense it debugs the validated specification by ensuring that there are no logical inconsistencies and that the project would indeed achieve its objectives. Verification ensures that once the process of developing a jet aircraft for a client has started, the client will receive a working jet.

In short, validation answers the question “are we building the right product?”, while verification answers the question “are we building the product right?”

CHAPTER 4

RESEARCH PROBLEM

THE introduction of changes into a business process can lead to unpredictable results. The results of changes in systems that are complex, concurrent, distributed, non-sequential and interdependent are even less predictable. To mitigate this uncertainty, models are built that describe these changes, and predict the results they will have. The construction of these models and using them to predict effects are challenging as well.

To represent a system as a model, abstraction is needed. An incorrect abstraction of a system to a model renders the model potentially useless; there is no point predicting effects off of a model if the model does not accurately reflect the system. An incorrect level of abstraction is also problematic. Too much abstraction leaves models that are not capable of answering the required questions; too little abstraction results in large models that cannot be analysed, constructed, or require too much time or effort to construct.

A means is required to abstract models from a business process that are an accurate reflection of the system; capable of answering questions posed to it, and yet be small enough to construct and interpret easily. The interpretation of the model itself poses problems once abstraction is completed.

Simulation is used to anticipate model behaviour. Though capable of evaluating model behaviour, simulation is bound by certain limitations. It relies on unknown or estimated data for its operation, and simulation only provides feedback on general model behaviour. Simulation does not evaluate all possible executions of the model, and can therefore not prove model properties or behaviours.

The evaluation of models without simulation is even less robust. It relies on intuition and testing to find faults in the process. Testing can also not prove properties or evaluate all possible executions of the system.

An approach is therefore needed that can abstract, construct and evaluate models easily. This

method must be able to evaluate all possible executions, answer questions on model behaviour with certainty, and be capable of formally proving model properties.

4.1 Research argument

Model checking is capable of addressing the needs as described above. It evaluates all possible model behaviours and answers questions regarding behaviour. The aforementioned answers provided the by model checking are the formally proven behaviour properties of the model.

In order to apply model checking, formal specifications of the process and behaviour properties are required. So for business processes to be model checked requires them to be specified in a formal language. Models must also be small enough to be verified.

A business process consists of a definable and finite set of rules that structure related tasks and activities together to produce a specific service or product. These rules and their behaviour requirements can be specified in a formal language like PROMELA. Therefore, in general, business processes can be verified in the same way as a communication's protocol with model checking. BIZZDesigner uses this this property to convert business processes into PROMELA code for verification with SPIN.

One objection to the formalisation of business processes is that the human component makes it unsuitable for model checking[22]. Human behaviour is viewed as irrational or non-deterministic and not verifiable using model checking. This objection is, however, not valid. Though a single computer may be deterministic in its operation, the communication between a network of computers is not. Model checking is used to develop procedures and protocols that operate reliably in non-deterministic environments.

Model checking is used to address the unpredictability of distributed and networked computing. It allows for the development of procedures that can reliably function in this non-deterministic[1] environment. The unpredictability of humans in a system is not a problem for model checking, rather it can help to reduce the effects that such behaviour has on the system.

Using model checking on humans and business resources can build predictable and reliable systems from unpredictable and unreliable components, in the same way that we have reliable and predictable communications over unreliable and unpredictable network infrastructure.

4.1.1 Model checking in an informal environment

The challenge in using model checking in business processes is not the unpredictability of the components from which it is made, but rather its lack of formality. There are too many variables to consider and the processes are not always concrete enough. The problem is the abstraction of the process to a level where model checking can be applied to it.

[1] Network activity and timing issues result in the non-deterministic behaviour of network computing.

There are two obstacles to the verification of business processes. Firstly the transformation of both the formal and informal elements of a business process into a PROMELA model. Secondly this model must be of verifiable complexity. A method is needed to process the "soft", "fuzzy" or informal elements of humans and organisations into models and requirements that can be model checked.

4.1.2 Formality distillation with the Logical Thinking Process

The formalised common sense of the Logical Thinking Process uses both the formal and informal components to analyse the system. It takes the formal parts and scrutinises them with the intuition and feelings of the people who are involved in the system. Though not a formal method in itself, it uses the informal to abstract a model that describes the essence of the problem in the system.

Although the inputs and processes of the Logical Thinking Process are not formal, the outputs are. They provide concrete verification requirements, as well as solutions that can be model checked. The process also abstracts the solution from the system by only focusing on the trouble spots. This reduces the complexity of the models, and makes them easier to construct and verify.

4.2 Solution outline

In the next chapter we will take the Logical Thinking Process and use it to construct abstract models of the reality that is currently operating in the system. In doing so, the non-formal components are removed and abstract solutions constructed that are formal enough to model check. We will then take these solutions and transform them into models that are then model checked using standard model checking methods.

CHAPTER 5

THE COMPLEXITY ALLEVIATION METHOD

IN this chapter we introduce the Complexity Alleviation Method (CAM) we have developed to formalise business protocols from their informal environment and to present them in a form suitable for model checking. A graphical outline of the method is given in Figure 5.1.

CAM transforms business processes and their non-formal components with the Logical Thinking Processes into system specifications and verification requirements. These specifications and requirements are then translated into PROMELA and verified using SPIN.

5.1 Methodology overview

A business process that is model checked is described by three different paradigms when CAM is used. This is the normal, or reality paradigm description of the process, the TOC paradigm description, and the model checking paradigm description. The TOC paradigm is used to identify the principal constraint of the system and develop solutions that would address this constraint, as can be seen in Figure 5.1. The model checking paradigm then takes theses solutions and verifies if they in fact possess the properties that they are supposed to have, thus leaving a validated plan.

The transition between these paradigms is, to a large extent, a jargon transformation. In the following section we map the terminology used in the different paradigms and discuss how they relate to each other.

5.1.1 Reality to TOC terminology transformation

When changing a system, two design considerations are present; *concerns* that need to be addressed and *objectives* that need to be met. Modelling a solution should address both of these areas.

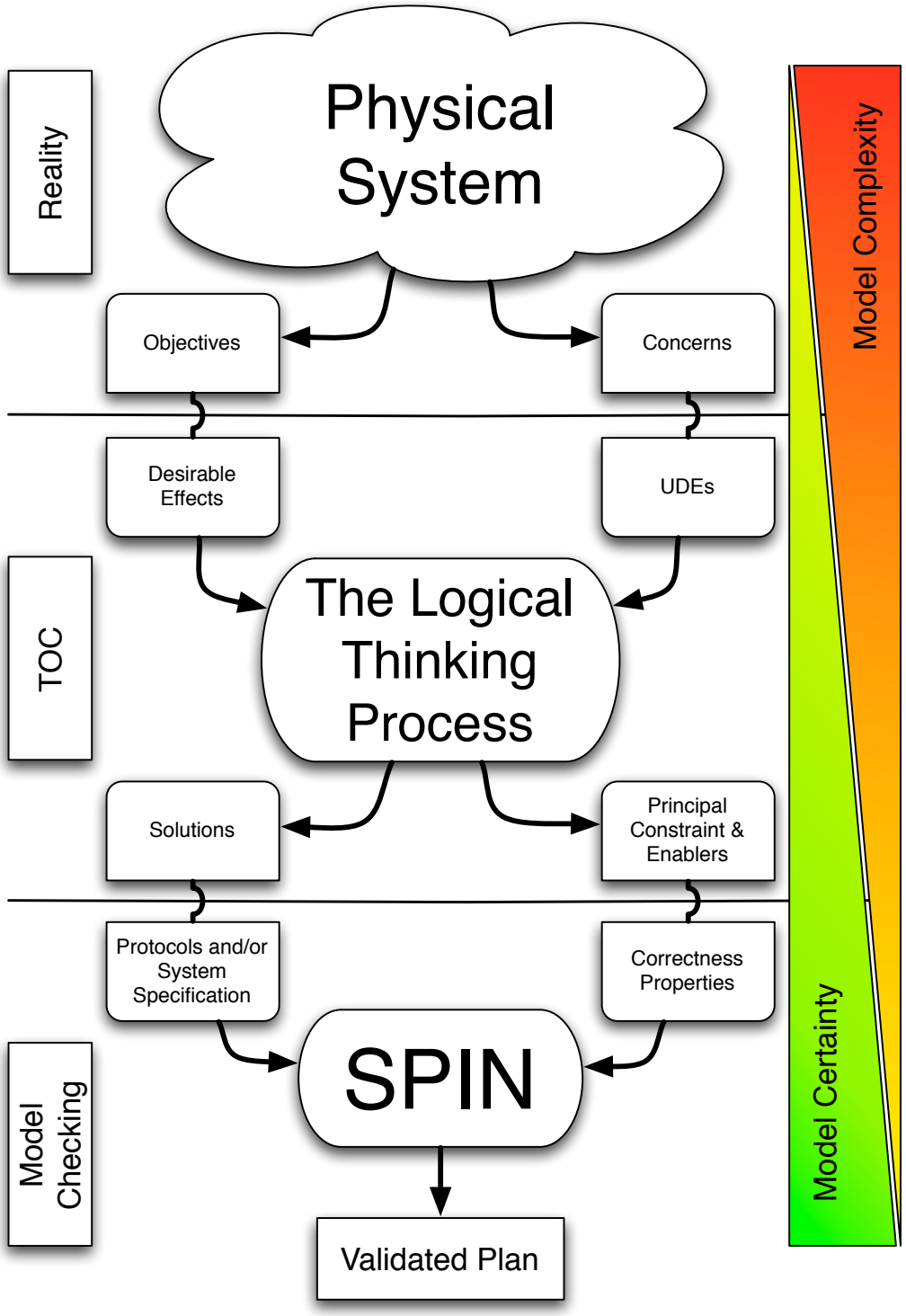


Figure 5.1: Diagrammatic overview of CAM

Concerns about the system are known in TOC as Undesirable Effects (UDEs) and are used to construct the CRT as illustrated in Section 2.2.1. Objectives or ideas about how the system should behave are called Desirable Effects. These are used in the construction of the FRT.

5.1.2 TOC to model checking terminology transformation

The CRT maps the process and determines the root cause of its problems or principal constraint. This principal constraint is then translated into safety properties. We wish to eliminate the principal constraint or root cause from our system, in other words, verify its non-existence in the system by specifying it as a safety property. Likewise, the FRT determines the principal enablers, entities which we desire to occur infinitely often; these enablers are translated into liveness properties. The Conflict Cloud produces solutions that can be specified in PROMELA and are abstract enough to be verified by SPIN.

This provides SPIN with all the components required for verification; formally specified correctness properties, and an abstracted process model.

5.2 The application of the Logical Thinking Process

In this section we will describe how to transform business processes into inputs for SPIN using the Logical Thinking Process. The process of transformation is depicted in Figure 5.2.

Objectives and concerns in the reality domain must be transformed into protocols and verification requirements in order for proposed business processes to be model checked. This transformation is done in three parts: determining the principal constraint, developing solutions, and establishing the principal enabler.

5.2.1 Formulating safety properties from the CRT

We will start by discussing how to determine the principal constraint from the concerns or UDEs using the CRT, and how to translate this into safety properties. This section focuses on the TOC subprocess as highlighted in Figure 5.3.

The aim of TOC is systems improvement, by identifying, exploiting, elevating and eventually eliminating the constraint as described in Section 2.1.2. A business process that is model checked must at the very least address the principal constraint. In other words, any solution that is produced by our approach must, under all circumstances, be free of the principal constraint. This is the exact definition of a safety property. So the principal constraint is verified as a safety property of the new business process.

Assuming that a system is interrelated, we ascribe multiple concerns UDEs to a single cause. This single cause to most problems is the principal constraint safety property. The process to determine the principal constraint starts with identifying and formulating the UDEs.

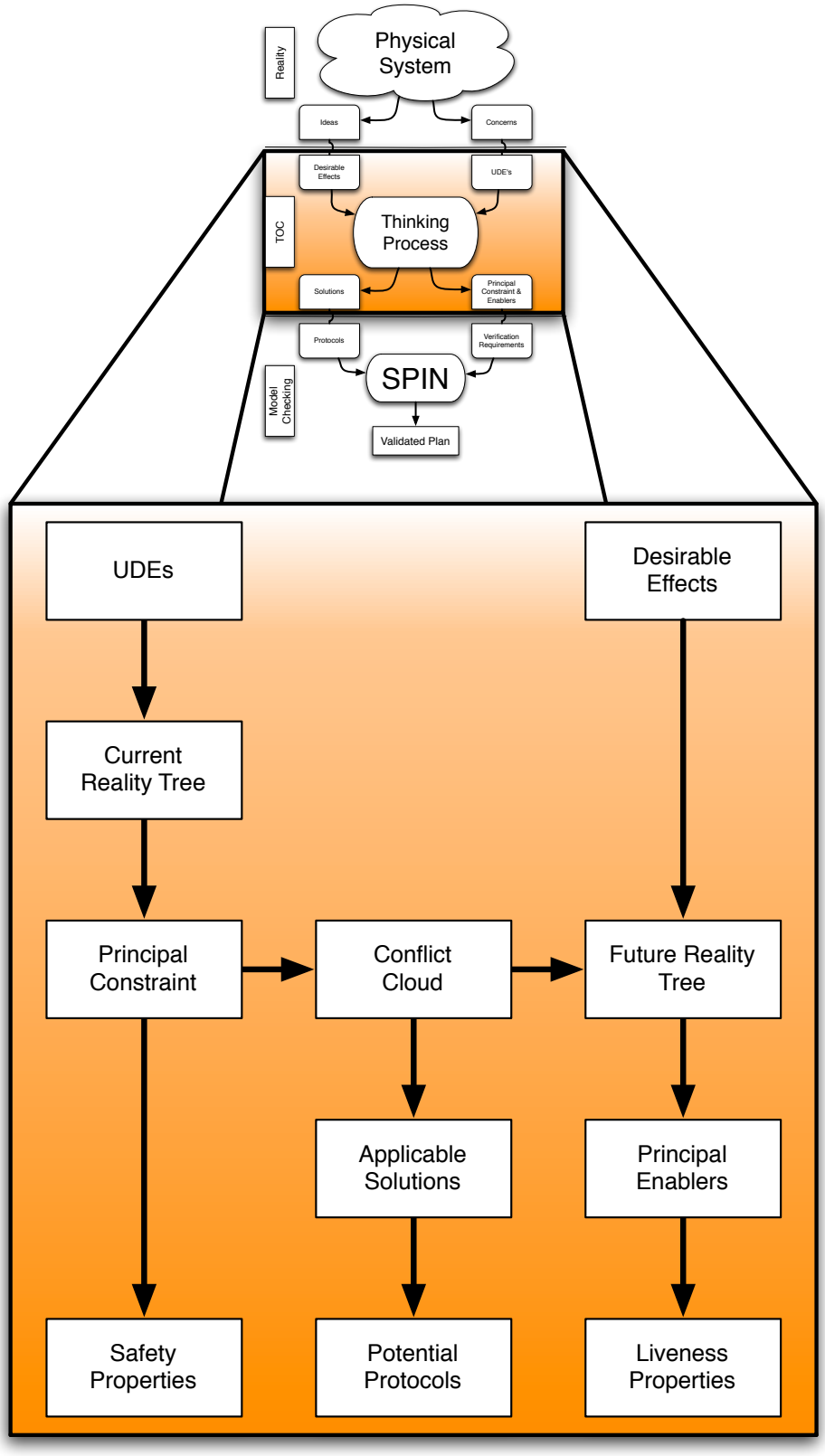


Figure 5.2: Expanded transitional diagram of the TOC sub-processes

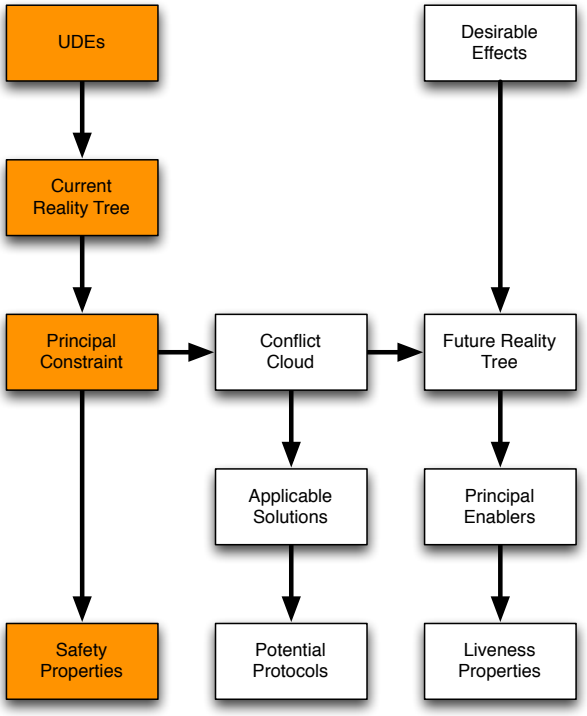


Figure 5.3: The formulation of safety properties as part of the TOC sub-processes

At any point in time there are numerous UDEs in a system. These UDEs are subjective and depend on the person that identifies them, and as discussed in Section 2.1.1, they are also context dependant. UDEs that are used in the CRT have to meet additional criteria, over and above just being a concern. Everyone must agree that it is a relevant UDE in the system. Only the three to five most important and agreed upon UDEs are used to construct a CRT.

For demonstration, four generic UDEs A, B, C and D are shown in Figure 5.4. These UDEs form the start of the CRT that will be built. They have also been coloured red for identification.

Wanting to understand the relationship between the four UDEs, we describe their cause or effect. The relationships of the UDEs in Figure 5.4 are read as follows: E causes C, or C exists because of E. If B exists then F happens. *If D and G* exist, *then A* will be present in the system.

We continue to expand the cause and effect relationships until all the UDEs are connected in a single diagram as shown in Figure 5.5.

We want to find the principal constraint. We described the cause and effect relationships between the UDEs until all of them were related. The constraint in the system is then the entity that is the root cause of most, if not all of the UDEs. In Figure 5.5 the principal constraint is entity E.

E is the principal constraint since none of the UDEs will exist if E does not exist. We make this statement due to the causal relationship we described between entity E and the UDEs. In the event

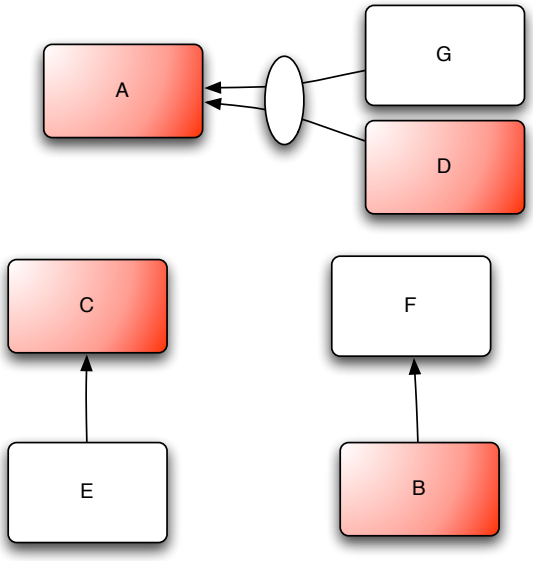


Figure 5.4: Cause and effect of UDEs *A*, *B*, *C* and *D*

that we remove *E*, *C* will no longer exist. Entity *I* exists because of UDEs *A* and *C* and is therefore directly dependent on *E* as well. The UDE *B* has a similar dependence on *E*.

E is the common denominator for the existence of all UDEs in the system and is, as a result, the principal constraint. If we remove *E* then none of the UDEs are caused based on our understanding of the relationships that exits in the system. Removing *E* therefore solves all the UDEs in our system.

The non-existence of *E* is the safety property of the new process. This is because *E* has been identified as the principal constraint and is the root cause of all of the UDEs in the system. If it can be shown that *E* does not exist in the new business process then the new process will address all the UDEs of our current system. A new process must at the very least address the principal constraint for it to be effective.

It has been mentioned that a safety property is something that is never to be violated. Therefore if a solution has been verified for a specific safety property it is assured that the undesired situation will never occur.

So if *E* is the principal constraint in the system then it is necessary to demonstrate in the new solution that *E* will never occur. This can be expressed in LTL as Always Not *E*, or equivalently

$$\Box\,!\,E. \tag{5.1}$$

Note that equation (5.1) is a valid LTL statement which can be used by a model checker to validate that a model adheres to the above-mentioned requirements.

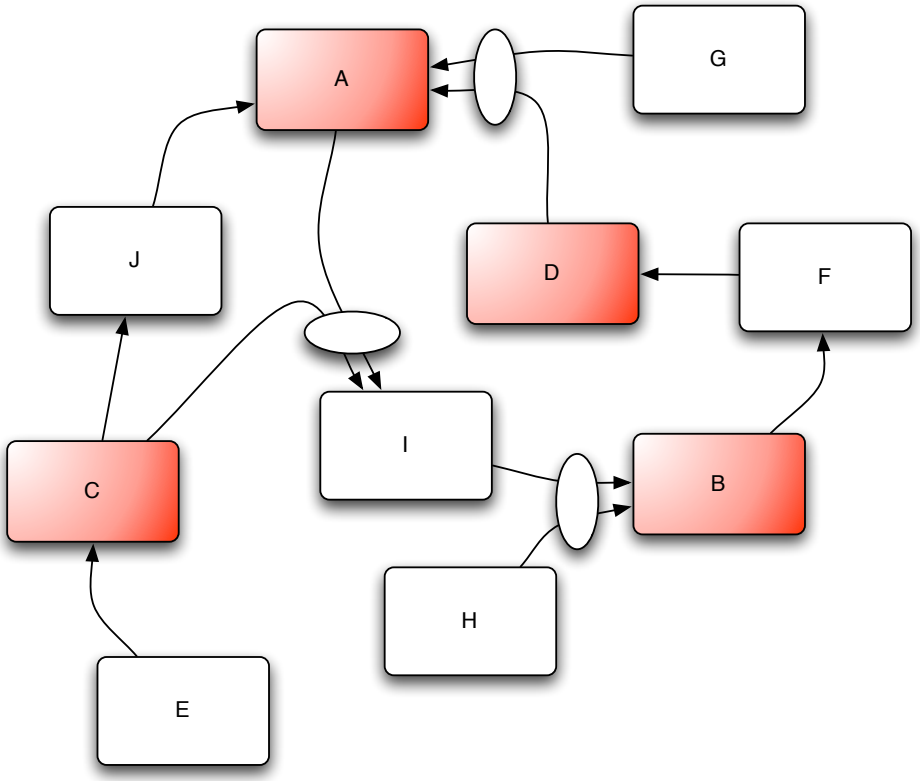


Figure 5.5: Completed Reality Tree

5.2.2 Example

As a more concrete example, let us take a financial service provider that has determined that the principal constraint in their current business process is that clients do not have 24-hour access to their finances. We must then demonstrate using SPIN that clients always have access to their finances. As an LTL safety property this is given by

$$\Box \, Access\ To\ Finances. \tag{5.2}$$

The preceding section demonstrated how to take the concerns that are present in the physical system and transform them with the Logical Thinking Process into correctness properties. The principal constraint derived from the CRT is the safety property required by SPIN to verify potential solutions.

To construct a model that has to describe the behaviour of a single principal constraint property instead of multiple UDEs reduces model complexity. The implication of this is that elaborate solutions describing numerous concerns are no longer required. The result is smaller models with less variables that are easier to construct and verify.

5.2.3 Solution selection and creation

Section 5.2.1 demonstrated how to determine the principal constraint and define safety properties. In this section we discuss the construction of the Conflict Cloud from the principal constraint, and how to develop solutions from it as introduced in Section 2.2.2 p. 8. Figure 5.6 depicts the progress of CAM.

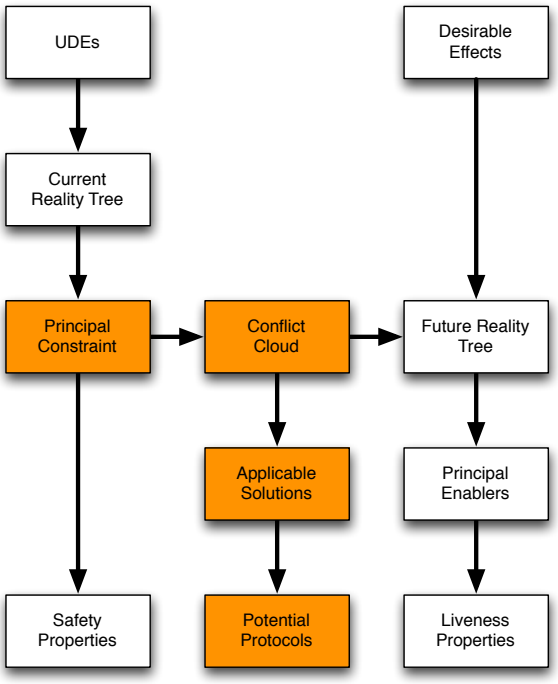


Figure 5.6: Solution selection and creation process

In the previous section we identified that the current system has a principal constraint E. The new system should have the safety property (5.1) but still retain all the functionality of the old system. This poses a problem.

If we look back we see that E was not identified as a UDE, and by implication, is not an obvious concern in the system. Otherwise it would have been selected as a UDE. So the existence of E has some reason for its presence or performs some function other than causing the UDEs. So removing E will have an impact other than the removal of UDEs from the system. This leads to a conflict. Keep E and retain current system functionality, or remove E and the UDEs but lose system functionality.

This is where Conflict Cloud comes in. It is designed to help devise solutions that will retain all the functionality of having E but with none of the UDE impact.

The generic construction of the clouds used in CAM starts with the principal constraint identified in the CRT. The principal constraint in the generic Conflict Cloud is represented by entity D. The objective is to construct a system that is free from the principal constraint and would imply the existence of an entity that is the opposite, or inverse of the principal constraint, $!D$. This inverse principal constraint is represented by entity D' in the cloud shown in Figure 5.7. All principal constraints possess an inverse; it would not be possible to construct a Conflict Cloud without them.

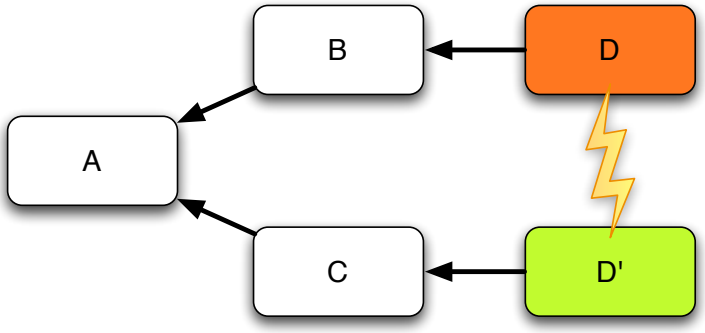


Figure 5.7: Conflict Cloud of the principal constraint

The second step would be to find B and C, and they are determined by answering the following questions:

- *Why would the existence of D/D' be desirable?*
- *What would fail to happen should D/D' not exist?*
- *What would be achievable if D/D' is present?*

Entity A represents the common objective of both B and C. Two perceived mutually exclusive aims are competing against one another for the same objective, D, and D', this giving rise to the conflict, and as a result this common goal must be determined. The formulation of the common objective lays the foundation for solution development by exposing the assumptions that are inhibiting system performance.

The Conflict Cloud gives rise to a logical contradiction, since both D and D' strive to obtain the same goal, yet D and D' are deemed mutually exclusive. Therefore one of the assumptions used to construct the cloud must be flawed. The Conflict Cloud finds the assumption that causes the principal constraint in the same way the CRT finds the principal constraint from the UDEs. This incorrect and inhibiting assumption is the key to the development of a solution that eliminates the principal constraint from the system.

Let us explain the identification of this assumption by means of an example.

We start with the principal constraint of the system. In Section 5.2.1 we stated the bank identified their principal constraint as "clients do not have 24-hour access to their funds". This is because clients need to transact outside of normal office hours. We now label entity D in the Conflict Cloud *"We maintain normal business hours"*. The inverse of the principal constraint D is placed in D', *"Provide 24-hour service"*.

So, why would the existence of the principal constraint D be desirable? The answer is that normal operating hours keeps overheads in check. This is placed in B. Extending business hours would create more transactions for the bank and therefore more turnover. So C is labelled *"Generate more transactions"*.

To complete the Conflict Cloud a common objective between B and C is found. This would be the long-term profitability of the bank. This common objective is placed in A and completes the Conflict Cloud. The cloud is provided in Figure 5.8.

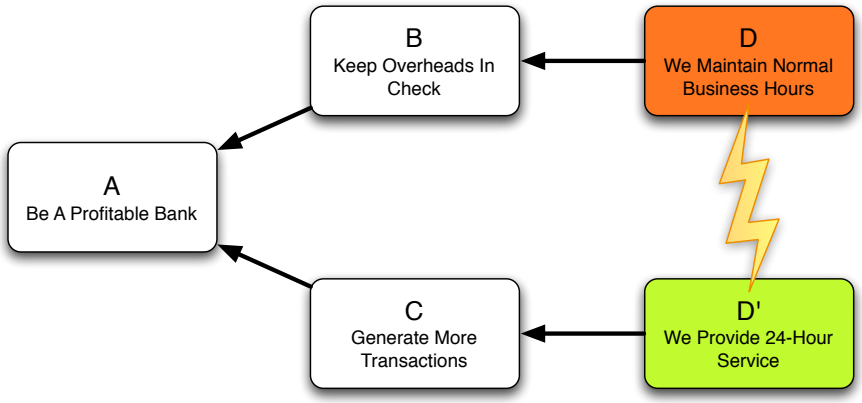


Figure 5.8: Bank example of a Conflict Cloud

In order to solve the Conflict Cloud we need to identify the invalid assumption that gives rise to the contradiction in the Conflict Cloud.

The example states that is necessary to provide clients 24/7 access to their finances in order to improve turnover and profitability. At the same time it is not possible to extend business hours. Hiring extra staff for the extended business hours will raise overheads considerably. There exists a conflict as well as a contradiction in the Conflict Cloud. So, in order to increase profitability, it would appear that profitability needs to be reduced.

What makes the Conflict Cloud so powerful is that it visibly presents the associated assumptions of the conflict. These assumptions are located under the arrows and the lightning bolt. Therefore a closer look is taken at each of these assumptions and from amongst the five. one should appear suspect. In the case of the Conflict Cloud in Figure 5.8 the assumption that is first to merit a closer

look would be the lightning bolt between D and D'. The assumption that underpins the lightning bolt would be that in order to provide 24-hour service to clients normal business hours need to be increased, and this is not necessarily true.

The aim would be to find an instance where the aforementioned assumption would no longer necessarily be true. Introducing an ATM into the bank's business process would accomplish just that, since the bank would not need to extend their normal business hours but would still be able to transact with their clients 24/7.

Therefore it would appear that the solution to the principal constraint as described here would be the introduction of an ATM in the bank's business process. This conjecture forms the departure point from which solutions are constructed.

Solutions are now developed around this conjecture. They focus on the incorporation of new ideas into the current system. Some questions need to be answered: How will we accommodate an ATM in the current business process model? What will change when an ATM system is introduced? How will this change affect clients?

It is the incorporation of the conjecture into the current system that gives rise to solutions. The conjecture is derived from the principal constraint, so the solutions are centred around it as well. There may also be more than one way to incorporate the conjecture into the current system. Even so, all these solutions have to address the same safety property, and flow from the same conjecture. If only one safety property is addressed and verified by all solutions, instead of multiple solutions addressing different properties, then solution models are small, abstract, quicker to construct and comparable with each other.

The addition of the conjecture changes the underlying protocol on which an organisation operates. It is these changes which we would like to model and verify. The development of the new business protocol is done by working out how the process will change or look when the new conjecture operates in the system. As has been said, there may be more than one way this can be done, and all these should be developed and explored.

The protocol describes how the process will operate under the new incorporated conjecture. The aim of the protocol is to eliminate the principal constraint, and we therefore specify the non-existence of the principal constraint as a safety property. Had we not used the Logical Thinking Process, it is not guaranteed that a common safety property would be shared by multiple solutions. This common principal constraint is used to focus the model abstraction and enables multiple solutions to be compared with each other. The informality of business processes that is mentioned in Section 4.1.1 does not lend itself to this process without the help of the Logical Thinking Process.

Instead of modelling the entire operating system of the bank, addressing numerous variables, all that the protocol must show is that the client will always have access to his finances. The solution for the bank takes shape as follows.

Prior to the introduction of the ATM the customer process would have been described as in either of one of three states. Idle, going to the bank or waiting for the bank. If a customer is in the idle

position then the client is not in need of money. In the event that the need for money arises one of two things can happen. The client can move to the state "going to the bank" in the event that the bank is open, or "wait" until the bank opens up the next day. The introduction of an ATM into the system then allows for another possibility should the customer desire money and the bank is closed. Now a customer is no longer forced to enter into the "wait" state outside of business hours.

The FSM representation of these protocols can be seen in Section 5.3.1 in Figure 5.13.

5.2.4 The principal constraint as a common performance measurement unit and how to distinguish between multiple solutions

At this stage of CAM we have generated protocols and verification requirements, and this is what is needed for a verification run. If SPIN verifies that the safety property holds then it has been demonstrated that the model will address the principal constraint and therefore bring benefit to the system. If there is only one solution and it addresses the principal constraint the analysis can stop. In the event that multiple solutions have been developed and are capable of addressing the same principal constraint, it would be useful to distinguish between them based on some form of merit.

This is where having the same safety property between different solutions becomes useful. Were this not the case it would be difficult to compare different solutions with each other. It is not possible to objectively measure the suitability of solutions if they do not address the same problem. If solutions were developed for the respective UDEs, then how would you compare the solutions to each other? How do you weigh the importance of the different UDEs, and what does it mean if solutions are found that do not address all of the UDEs? Reducing the multiple UDEs to a single root cause provides a common unit of measurement and solves these problems.

Having established now that more than one solution can address the principal constraint, it would be informative to know what additional benefits the different methods can provide. The following section determines these benefits in the same way as the identification of the principal constraint.

5.2.5 Formulation of liveness properties

This section of the methodology is highlighted in Figure 5.9. The following section enables SPIN to demonstrate the differences between multiple solutions that verifiably solves the principal constraint.

It has been stated that the aim is systems improvement, and that the greatest improvement is gained when the principal constraint is eliminated. This is verified as a safety property in the previous section and as a proven capability of any solutions at this point. Having brought benefit to the system it is possible to further distinguish between solutions with the use of liveness properties.

Holzmann describes liveness and safety properties as follows. Safety, then, defines the *bad* things that should be avoided, and liveness defines the *good* things that capture the required functionality of the system.

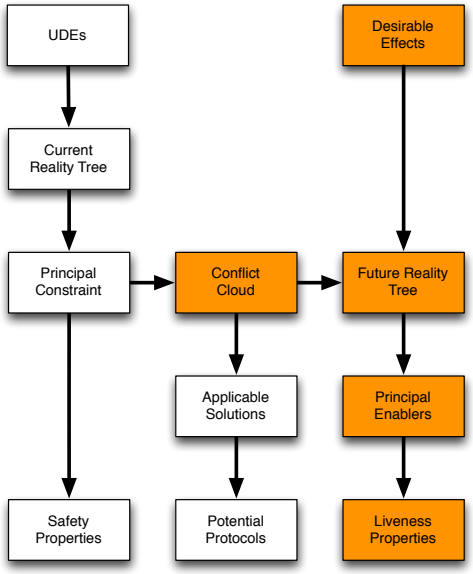


Figure 5.9: Expanded transitional diagram of the TOC sub-process generating liveness properties

Paraphrasing Holzmann in our context means the elimination of the identified UDEs (*bad*) as safety properties and the description of additional *good* features about the system as liveness properties. These liveness properties are identified in the FRT.

The FRT is built on the same common sense as the CRT as described in Section 2.2.3, p.8. Where the CRT starts with the UDEs the FRT takes the conjecture from the Conflict Cloud, and expands it until it is linked with all the desirable effects to be seen in the system. Liveness properties are desirable situations which should occur infinitely often in a system. If this is the case, then the TOC equivalent of liveness priorities are desirable effects. Like the CRT, the FRT takes multiple causes and conjectures[1] and determines the relationships between them.

These relationships take the injections which solves the Conflict Cloud and expands them until they reach the desired effects which are to be seen in the system. There are only a limited number of these desirable effects, thus reducing the complexity model of the solution.

The desirable effect to model for is the effect which stems from the greatest number of assumptions. The process is similar to the determination of the principal constraint only the direction is inverted. Where the safety properties were found at the bottom of the CRT, liveness properties are found at the top of the FRT.

The beauty of using trees is that they intrinsically reduce complexity. Once again, complex and interrelated realites as well as the multiple impacts of proposed solutions are reduced to a limited number of focus points.

[1] Also known in TOC as injections.

Liveness properties are formulated in LTL in the same way as safety properties in Section 5.2.1.

5.3 Model checking domain

This section discusses the model checking and final section of CAM depicted in Figure 5.10. It shows how the verification of a solution generated by the Logical Thinking Process is done.

5.3.1 Input transformation

The solutions and verification properties generated in Section 5.1.2 are transformed to make them suitable for SPIN and verification. This process is shown in Figure 5.11.

We have already formulated the safety and liveness properties with LTL in the previous section. Safety properties are the negation of the principal constraint and liveness properties the desired effects. This holds true for the majority of the cases. It may, however, be necessary in some cases to verify properties other than the simple negation of the principal constraint. This is left to the discretion of the modeller.

SPIN does not accept LTL formulae in their raw form. It is necessary to reformulate the formulae in PROMELA code. This conversion is done by a separate module in SPIN and all that is required is a valid LTL statement. The conversion of the "Always Not Wait" safety property identified as the bank's principal constraint is done by the SPIN command: spin -f '[]!Wait'. The output of this command is listed below.

```
$ spin -f '[]!Wait'

never {    /* []!Wait */
accept_init:
T0_init:
    if
    :: (! ((Wait))) -> goto T0_init
    fi;
}
```

Listing 5.1: A never claim in PROMELA code as generated by SPIN

The never claim generated by SPIN is then inserted into the PROMELA model and is used to verify the behaviour properties.

Having converted the correctness properties into PROMELA code we now move on to the conversion of the solutions. The intermediate step to writing the solution in PROMELA is to draw it as an FSM. This is similar to writing the principal enablers and constraints as LTL statements before generating PROMELA code for it.

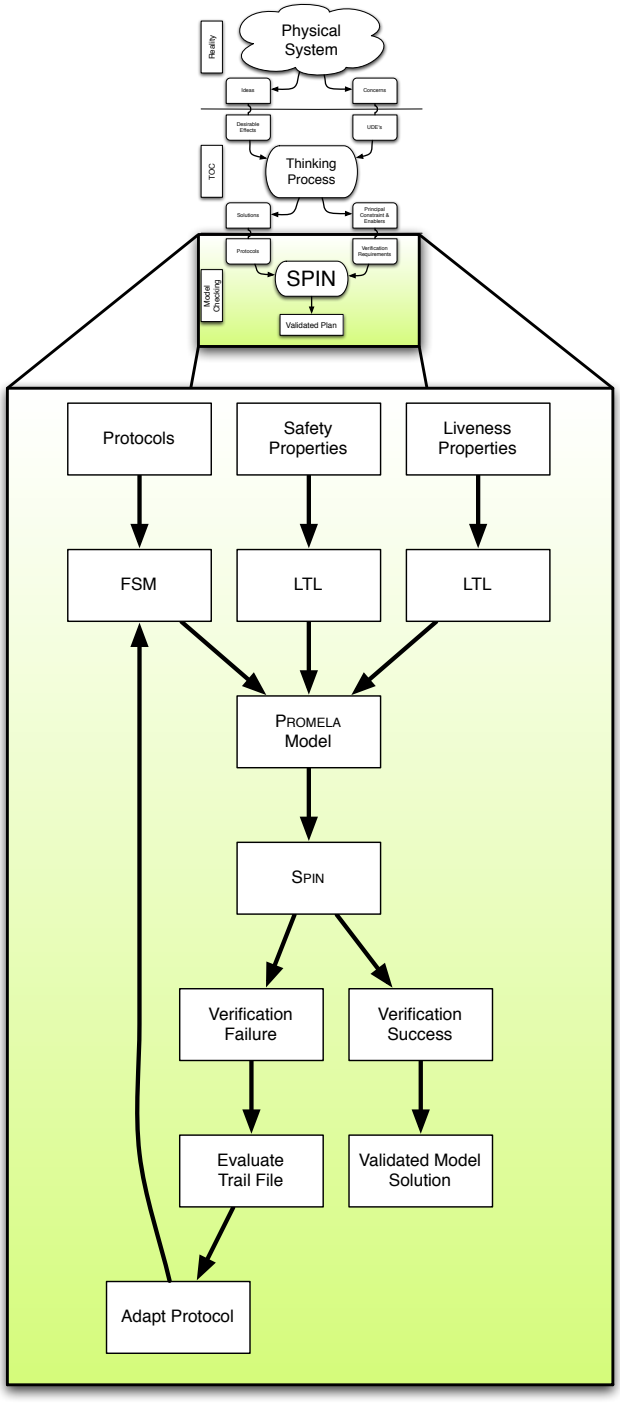


Figure 5.10: Diagrammatic method representation of the model checking domain

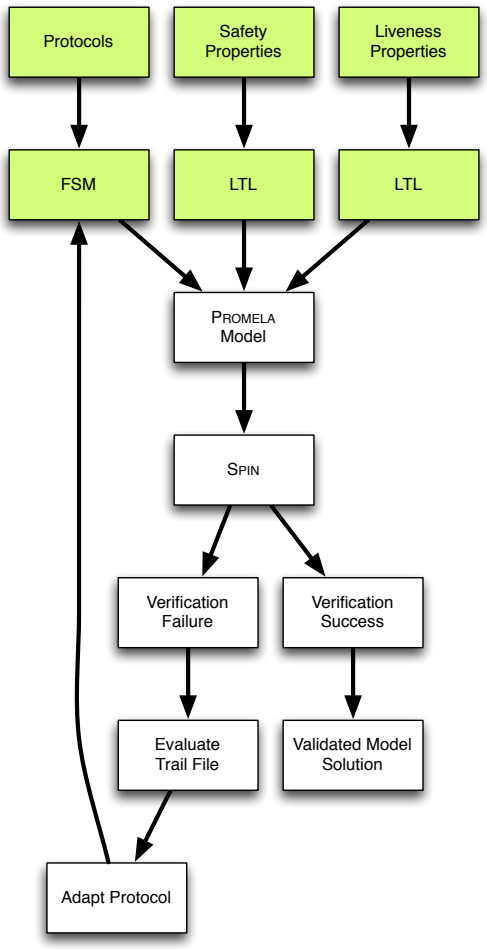


Figure 5.11: The homogenisation process within the model checking domain

The reason for this conversion into an FSM is twofold. Firstly it is easier to visualise the solution as an FSM. Secondly it is a simple conversion to write PROMELA code from an FSM. It is possible to write the PROMELA model directly from the solution, but the construction of the FSM as an intermediate step helps in the transformation of the solution from TOC to the model checking paradigm. To demonstrate this conversion let us take the ATM example given in Section 5.2.3 as an FSM.

The initial business process of the bank without an ATM consists of 3 states: `Idle`, `Wait` and `Bank`. `Idle` is the state that the customer remains in until he needs money. In the event that a need arises and the bank is available to meet that need, the state of the customer changes to `Bank`. After the customer has been serviced he returns to the `Idle` state. A customer enters the `Wait` state in the event that the need for bank services occurs outside of normal business hours. The FSM of this process is given in Figure 5.12.

The introduction of the ATM into the system changes the FSM by adding an additional state where

a customer can be served by an ATM over and above the services provided by the bank itself. As with the `Bank` state, once the client's needs have been serviced by the ATM he returns to the `Idle` state. The `Wait` state is still defined as the interim state when neither the bank nor the ATM can be accessed and the customer has the need to access his or her funds. The FSM of this figure can be seen in Figure 5.13.

Once the proposed protocol has been converted into an FSM (as described above) the construction of the PROMELA model can begin.

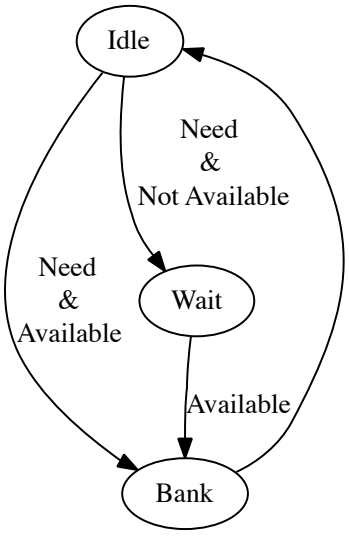


Figure 5.12: An FSM of the original process

5.3.2 PROMELA model construction and verification using SPIN

The next step in CAM, as shown in Figure 5.14, is the generation of the PROMELA model. The conversion from LTL to PROMELA was already shown in Listing 5.1. We will now describe how the translation form FSM to PROMELA is done.

We continue with the ATM example and construct a model of the FSM depicted in Figure 5.13. The model is written in pseudo code for the sake of clarity. The reader is referred to [14] for an exhaustive description on how a full PROMELA model is constructed.

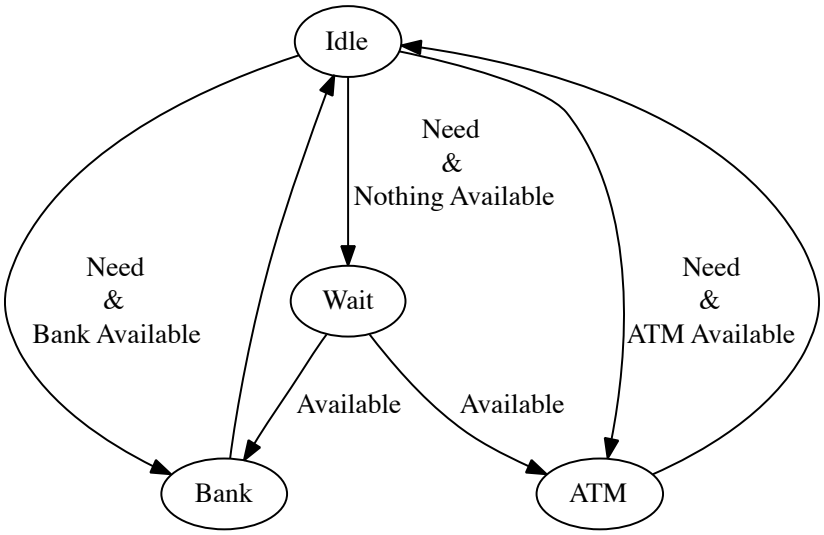


Figure 5.13: An FSM of the proposed business process that includes the introduction of an ATM

The construction of a PROMELA model from an FSM is done by the conversion of three of the elements in the FSM into their equivalent counterparts in PROMELA code. These elements and their PROMELA equivalents are listed below.

- Transitions between states are transitions (->) with guards in PROMELA.
- States in the FSM are rewritten as labels in PROMELA.
- The transition requirements are the guard statements before the Go-To's.

Labels Each state of the FSM in Figure 5.13 represents a label in the control structure of the PROMELA code. In the case of the ATM bank FSM instance there will be four separate labels:

- Idle
- ATM
- Bank
- Wait

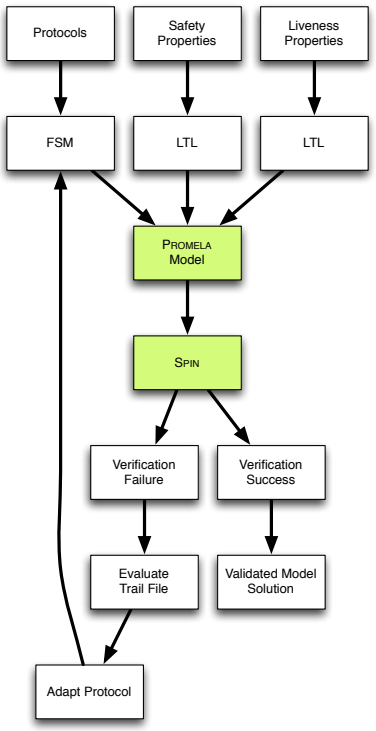


Figure 5.14: Model construction and verification process within the model checking domain

Guards The transitions that occur between the states depend on certain conditions prior to their execution. The client can only move from `Idle` to `Bank` if there is both a need and the bank is available; the same applies to the transition from `Idle` to `ATM`. Moving from `Idle` to `Wait` implies a need but no availability from either bank or ATM. The customer waits until either bank or ATM becomes available.

PROMELA Pseudo Code When all of these separate elements are combined it forms the PROMELA model in Listing 5.2. A working PROMELA model of both FSMs in Figures 5.12 and 5.13 is provided in Appendix A.

```
never {    /* []!Wait */
accept_init:
T0_init:
    if
    :: (! (Wait)) -> goto T0_init
    fi;
}

Idle:
```

```
    if
    ::(Need && Bank Available) -> goto Bank
    ::(Need && ATM Available) -> goto ATM
    ::(Need) -> goto Wait
    fi;

Bank:
    do
    :: goto Idle
    od;

ATM:
    do
    :: goto Idle
    od;

Wait:
     if
     :: (Bank Available) -> goto Bank
     :: (ATM Available) -> goto ATM
     fi;
```

Listing 5.2: PROMELA code of FSM in Figure 5.13

Verification By Using SPIN Once both the verification requirements as well as the model have been constructed the completed model is handed to SPIN for verification. The aim of the verification is simply to receive a "yes" or a "no" answer to the adherence of the solution to the behaviour specifications. In the event that SPIN returns a "yes" then the verification is complete. If "no" is returned we are interested in how the model failed, so that we can reevaluate the solution. Of all the output generated by SPIN only the "yes" "no" answer is important.

A verification for both bank models will be done, demonstrating the differences between the successful verification of a model and a verification failure. Let us then verify the FSM of the bank process with an ATM in Figure 5.13 with SPIN to ensure that the customer will never be placed in the "wait" state again. Listing 5.3 is the output of a SPIN for this verification run.

```
1 (Spin Version 5.2.2 -- 7 September 2009)
2   + Partial Order Reduction
3
4 Full statespace search for:
5   never claim           +
6   assertion violations  + (if within scope of claim)
7   acceptance   cycles   - (not selected)
8   invalid end states  - (disabled by never claim)
```

```
9
10 State-vector 28 byte, depth reached 27, errors: 0
11        17 states, stored
12        50 states, matched
13        67 transitions (= stored+matched)
14         0 atomic steps
15 hash conflicts:         0 (resolved)
16
17     4.653 memory usage (Mbyte)
18
19 unreached in proctype BankATM
20   line 42, state 17, "p = 1"
21   line 55, state 35, "-end-"
22   (2 of 35 states)
23
24 pan: elapsed time 0 seconds
```

Listing 5.3: SPIN output for the verification of the ATM bank process

The "yes" or "no" answer to the verification run is located on line 10 of Listing 5.3. The error value of 0 denotes a "yes" and means that the business process for the bank with an ATM does not place a customer in the wait condition under any circumstances, and implies a successful verification.

5.3.3 Verification results

The outcome of the verification determines which section of highlighted CAM is executed in Figure 5.15.

Verification Success A verification is considered a success in the event that SPIN returns an error value of 0, as does the example given in Listing 5.3. This means that there is no instance that where the solution fails to meet the verification requirements. This means that, by implication, the solution provably meets its behaviour objectives under all circumstances and can be implemented with confidence. The successful verification of the ATM bank model in Listing 5.3 implies that the bank can implement the solution with confidence and that CAM can terminate.

Verification Failure The FSM of the original bank processes in Figure 5.12 is expected to fail a verification with the same correctness properties. We subject the model to the same verification requirements as we did for the ATM model and analyse the output. The SPIN output for this verification is given in Listing 5.4.

```
1 pan: claim violated! (at depth 25)
2 pan: wrote NoATMModel.trail
```

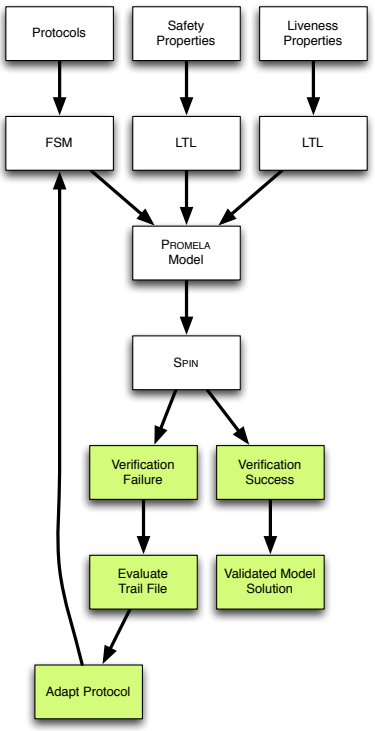


Figure 5.15: The verification results process within the model checking domain

```
3
4  (Spin Version 5.2.2 -- 7 September 2009)
5  Warning: Search not completed
6    + Partial Order Reduction
7
8  Full statespace search for:
9    never claim           +
10   assertion violations  + (if within scope of claim)
11   acceptance   cycles   - (not selected)
12   invalid end states  - (disabled by never claim)
13
14 State-vector 28 byte, depth reached 25, errors: 1
15        15 states, stored
16         8 states, matched
17        23 transitions (= stored+matched)
18         0 atomic steps
19 hash conflicts:         0 (resolved)
20
21     4.653 memory usage (Mbyte)
22
23
```

```
24 pan: elapsed time 0 seconds
```

Listing 5.4: SPIN output of the bank process without an ATM

In line 14 an error value of 1 has been returned. This means that, as expected, the model failed to meet the verification requirements. Line 2 refers to a .trail file that was written by SPIN. This file contains the information needed by SPIN to reproduce the failure run that shows how the model failed to meet the verification requirements. The simulated trail file for the failed verification of the bank business process is given in Listing 5.5.

```
1  Thomas:jock$ spin -t -p ATMModel
2  Starting BankATM with pid 0
3  Starting :never: with pid 1
4  Never claim moves to line 17  [else]
5    2:  proc  0 (BankATM) line  28 "ATMModel" (state 1) [p = 0]
6    4:  proc  0 (BankATM) line  49 "ATMModel" (state 28)  [Need = 1]
7    6:  proc  0 (BankATM) line  33 "ATMModel" (state 2)
8        [((State==1))]
9    8:  proc  0 (BankATM) line  35 "ATMModel" (state 3)
10       [((Need&&BankAvailable))]
11   8:  proc  0 (BankATM) line  35 "ATMModel" (state 4) [State = 2]
12  10:  proc  0 (BankATM) line  50 "ATMModel" (state 29)  [Need = 0]
13  12:  proc  0 (BankATM) line  52 "ATMModel" (state 31)
14     [BankAvailable = 0]
15  14:  proc  0 (BankATM) line  40 "ATMModel" (state 12)
16     [((State==2))]
17  14:  proc  0 (BankATM) line  40 "ATMModel" (state 13)  [State = 1]
18  16:  proc  0 (BankATM) line  49 "ATMModel" (state 28)  [Need = 1]
19  18:  proc  0 (BankATM) line  33 "ATMModel" (state 2)
20     [((State==1))]
21  20:  proc  0 (BankATM) line  37 "ATMModel" (state 7)
22     [(((Need&&!(BankAvailable)))]
23  20:  proc  0 (BankATM) line  37 "ATMModel" (state 8) [State = 4]
24  22:  proc  0 (BankATM) line  42 "ATMModel" (state 16)
25     [((State==4))]
26  24:  proc  0 (BankATM) line  42 "ATMModel" (state 17)  [p = 1]
27 Never claim moves to line 16  [(p)]
28 spin: trail ends after 25 steps
29 #processes: 1
30     p = 1
31  25:  proc  0 (BankATM) line  55 "ATMModel" (state 35)
32     <valid end state>
33  25:  proc  - (:never:) line  19 "ATMModel" (state 7)
34     <valid end state>
35 1 processes created
```

Listing 5.5: A SPIN simulation of the bank business process .trail file

Listing 5.5 is interpreted in conjunction with the PROMELA model in Listing A.2. The process is initialised with the bank being available and the customer not in a state of waiting. Line 6 shows that a customer registers a need to transact with the bank. Line 8 shows that the need arises during the idle state and Line 10 states that the bank is available to meet the need. The need disappears (line 12) while the customer is being serviced (line 16), but so does the bank availability (line 14). This can be interpreted as a client being helped in the bank while it closes. The need for service arises again (line 16) while the client is leaving (line 17 and 20) but the bank is now closed (line 14 and 22). This forces the client into the waiting state (line 23 and 25) that triggers the failure of the never claim (line 26 and 27).

This is a valid scenario of how the bank process without an ATM would require a customer to wait for it to open before it can transact. SPIN is capable, as shown above, of describing such failure in great and sequential detail. The combination is a step-by-step guide on how a solution failed. Though the example given here can be solved by inspection, it demonstrates the detail and the accuracy of the feedback provided by SPIN when model failures happen. This feedback can then be used in fixing the solution or to establish if a different approach is needed altogether. Once a replacement solution is developed it is again given to SPIN for verification.

CHAPTER 6

CAM EXAMPLE

WITH CAM developed in the preceding chapter let us demonstrate its operation by means of an example. CAM is applied to the bullwhip effect, a common supply chain problem, to explain how to apply it to a practical problem; the solution process works to abstract protocols from solutions, and implement and verify a model. It also demonstrates how a large and involved system can be represented by a small, abstracted, and verifiable model.

6.1 The bullwhip effect

The bullwhip effect is the term used to describe the phenomenon when only minor changes in demand for a product result in major fluctuations further up in the supply chain. This phenomenon has been documented as far back as 1961 [23] but its impact and effect are still prevalent in numerous supply chains today.

The phenomenon is well-studied and documented [24; 25; 26] with numerous case studies and reports on the phenomenon. The quintessential example of the bullwhip effect is the case where Procter & Gamble [27] noticed a massive fluctuation in the manufacturing demand for their Pampers diapers. This fluctuation was odd since the consumer demand for diapers remains stable. The fluctuations due to the amplification of normal variation in demand to abnormal levels further up the supply chain. This is the result of the policies, practises and procedures as well as physical and operational conditions that exist in the supply chain. The result of these factors is that the further away from the consumer you move in the supply chain the greater the amplification effect becomes.

The bullwhip effect compels manufacturing to invest in excess sprint capacity to fulfil orders. Capacity that is not only expensive, but also unnecessary, since there is no real fluctuation in demand.

One response is to push up inventory levels to deal with the fluctuations, which implies additional holding and obsolescence costs. Despite all the investment in inventory and excess capacity a supply chain subject to the bullwhip effect will also suffer frequent stock outs. Since there is no actual fluctuation in demand these additional expenses only add costs to the product without bringing any real benefit to the customer.

In this chapter we model business processes that are prone to suffer from the bullwhip effect. This shows how CAM models business processes and hoe it is used to analyse and improve systems. It also demonstrates how CAM is used to develop solutions and business processes that combat current operational problems, in this case the bullwhip effect.

6.2 TOC subsection of CAM

The first step in CAM is to identify concerns in the system. These concerns are the drivers for the construction of the CRT. The Logical Thinking Process requires the identification of 3 to 5 main concerns or UDEs for the construction of the CRT. In this analysis the UDEs identified in a supply chain suffering from bullwhip are that:

- resources are not used efficiently,
- there is not enough capacity to fulfil orders in time,
- there is too much inventory of unneeded stock keeping units (SKUs),
- and there are stock outs of needed SKUs.

We combine these UDEs into a CRT in order to find the constraint that is the common cause to them all. The CRT for the supply chain suffering from bullwhip is given in Figure 6.1.

From the CRT we can see that the root cause of all the UDEs stems from the fact that orders placed in the system are for both anticipated and actual demand. This is the reason for the amplification further up the supply chain. The amplification is due, in part, to the negative feedback loop, shown in Figure 6.1, created by the stock outs. This feedback loop reinforces the client's behaviour to anticipate stock outs and order quantities other than the actual demand. This further degrades the ability of forecasting methods to predict sales, which in turn leads to incorrect production and more stock outs before the whole process reinforces itself. The incorrect production schedule for the inaccurate forecast is the cause for the other UDEs.

Now that we understand why orders are placed in this way the question remains as to how this behaviour can be eliminated from the system in the future. The elimination of the principal constraint/behaviour will solve the current UDEs and bring benefit to the system, as we stated in the development of CAM. The way to eliminate this root cause from the system is to resolve the conflict

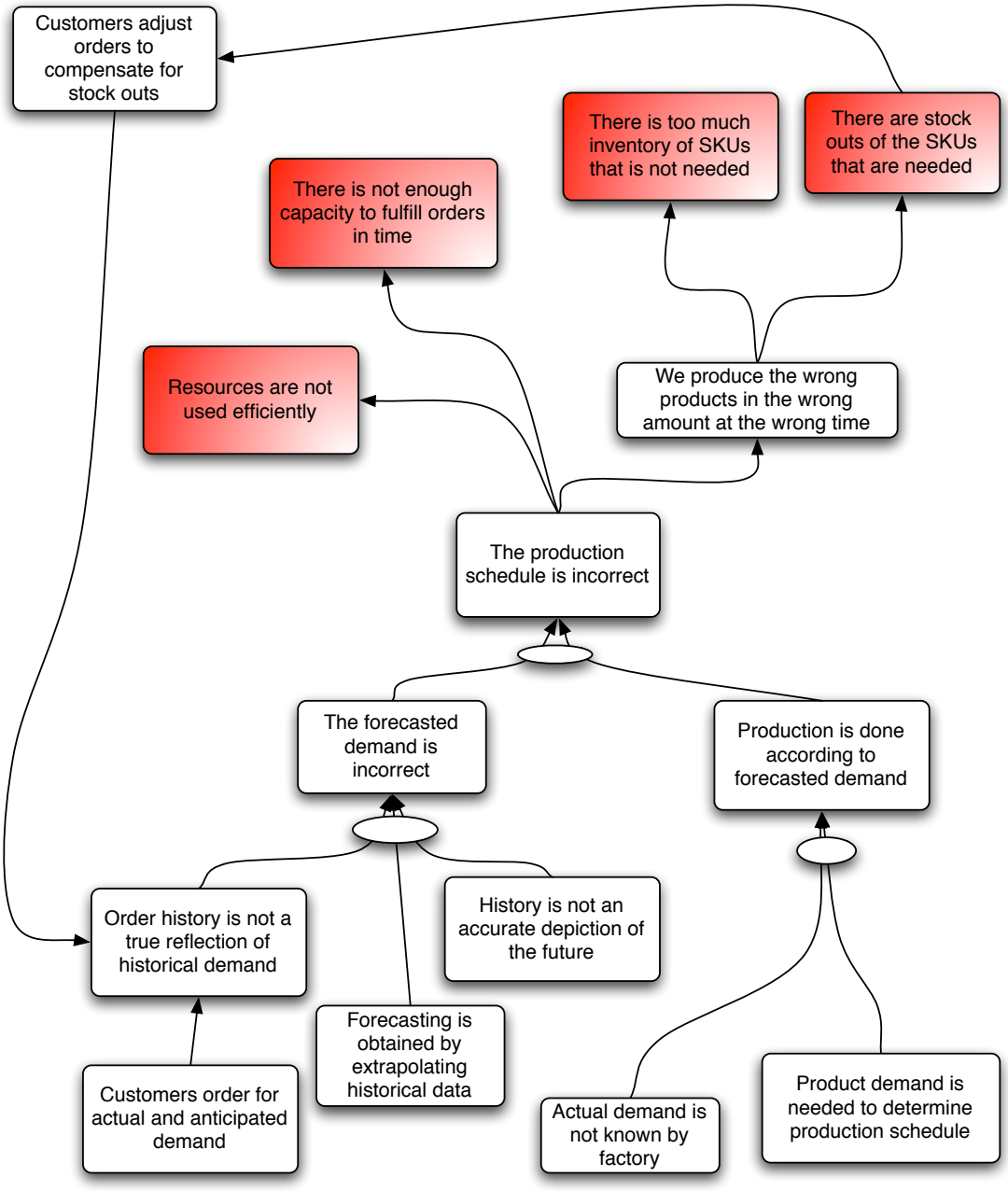


Figure 6.1: The CRT of the bullwhip supply chain

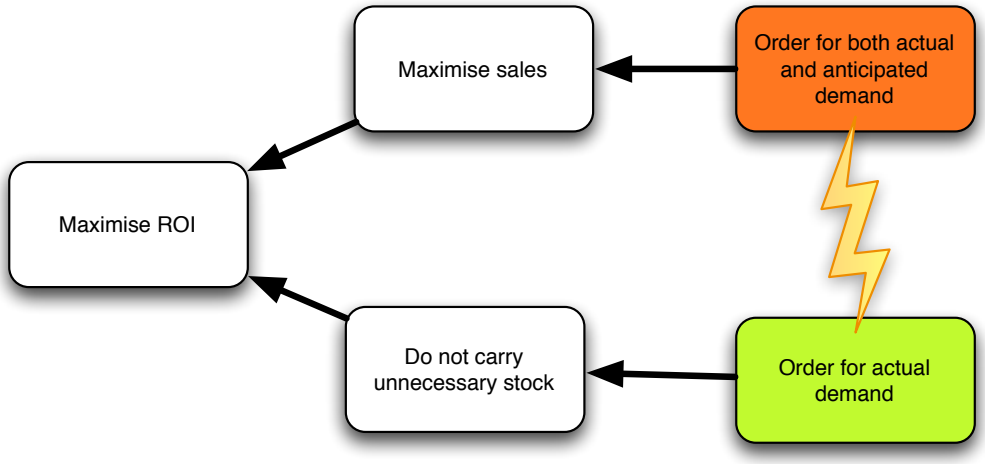


Figure 6.2: The Conflict Cloud depicting the cause of the bullwhip effect

which gives rise to its existence. This, as mentioned, is done by the construction of a Conflict Cloud. The Conflict Could for the bullwhip effect is provided in Figure 6.2.

The Conflict Cloud explains the problem faced by the supply chain as follows. The aim of the system is to maximise the return on investment (ROI) for each of the independent parts of the supply chain. To do this it is necessary that we maximise sales whilst not carring unnecessary stock. If we want to maximise sales we must ensure that no stock outs occur and therefore we order for actual sales as well as for anticipated demand. At the same time, if we carry unnecessary stock we incur carrying, holding, and obsolescence costs. These costs must be minimised to maximise ROI. This implies that we order only what is sold. This leads to a compromise between two conflicting actions that are intended to achieve the same objective.

To solve this conflict the assumptions that lead to it must be identified. The assumption that the use of a forecast to maximise sales is suspect. Forecasted sales information is inaccurate. Its reliance on historical data, and numerous assumptions based on client needs and preferences provides various degrees of error. The forecast is used to protect the supply chain against stock outs, yet the information we use to protect the system is inaccurate. If the forecast is based on historical data, this data may not be for sales only but for safety stock also, further degrading forecasting accuracy. The question is how do we maximise sales by not losing any to stock outs without the use of forecasted information. The aim is to replace forecasting with a more effective means of combating stock outs.

If we shorten the time it takes between the placement of an order and the actual delivery thereof, i.e. “lead times”, the risk of stock outs are reduced. If stock outs are addressed there is no need for customers to anticipate demand. If so, then clients only need to place orders that reflects actual demand. If we only need to respond to actual demand, which varies only slightly overall, instead of perceived

demand (which varies drastically) this reduces variability in the system. Less variability enables the system to respond even quicker to changes, thus eliminating the root causes of the bullwhip effect.

With the reduction in lead times it is possible to implement new order procedures in the supply chain. These procedures are to order only what is sold.

6.3 Model checking section of the CAM example

In the previous section we identified the principal constraint that is the cause of the bullwhip effect, i.e. orders are placed for both actual and forecasted demand. We must verify the safety property that orders are always equal to actual demand. This can be expressed in terms of LTL as

$$\Box\,(\mathit{Orders} \leftrightarrow \mathit{Demand}). \tag{6.1}$$

All proposed solutions to the bullwhip problem must satisfy this safety property.

The next step is to specify the proposed solutions as FSMs before they are transformed into PROMELA code for verification. Solutions should only order stock to replenish that which has been sold. The proposed implementation of this can be done in two different ways. The first option would be to have a set schedule or rules that would order stock from the supplier at defined intervals. A system functioning on this procedure would run on the FSM depicted in Figure 6.3. We denote this proposed procedure as S_a.

The alternative to S_a is to place an order after every single transaction. A solution that uses this approach would have an FSM that is depicted in Figure 6.4. Let us call this S_b. By placing an order after every order it receives itself, S_b feeds live consumption information to suppliers, and the feed creates transparency throughout the supply chain.

To verify these solutions against the principal constraint they are converted into PROMELA code and handed to SPIN. The code used for the verification runs in Section 6.4 are provided in Appendix B and C, respectively. The result of the verification of S_a and S_b against the principal constraint was successful. These results show that both solutions will provably address the principal constraint and bring benefit to the system and can therefore be implemented.

6.4 Determination of liveness properties

In the event that the Logical Thinking Process creates two separate solutions that both address the principal constraint, CAM is also used to distinguish between the solutions. As stated in Section 5.2.5, the way to accomplish this is with the use of liveness properties. These properties are derived from the FRT in the same way that safety properties are drawn from the CRT and its principal constraints. Liveness properties are the Desired Effects found in FRTs.

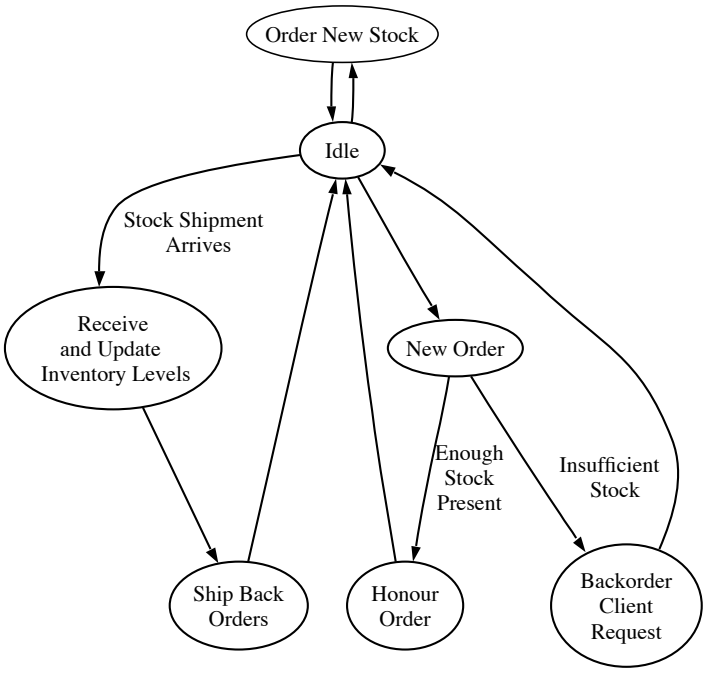


Figure 6.3: Proposed procedure S_a

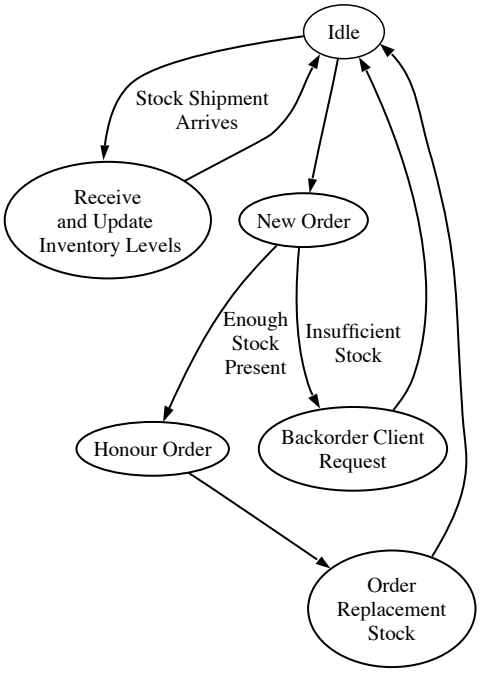


Figure 6.4: Proposed procedure S_b

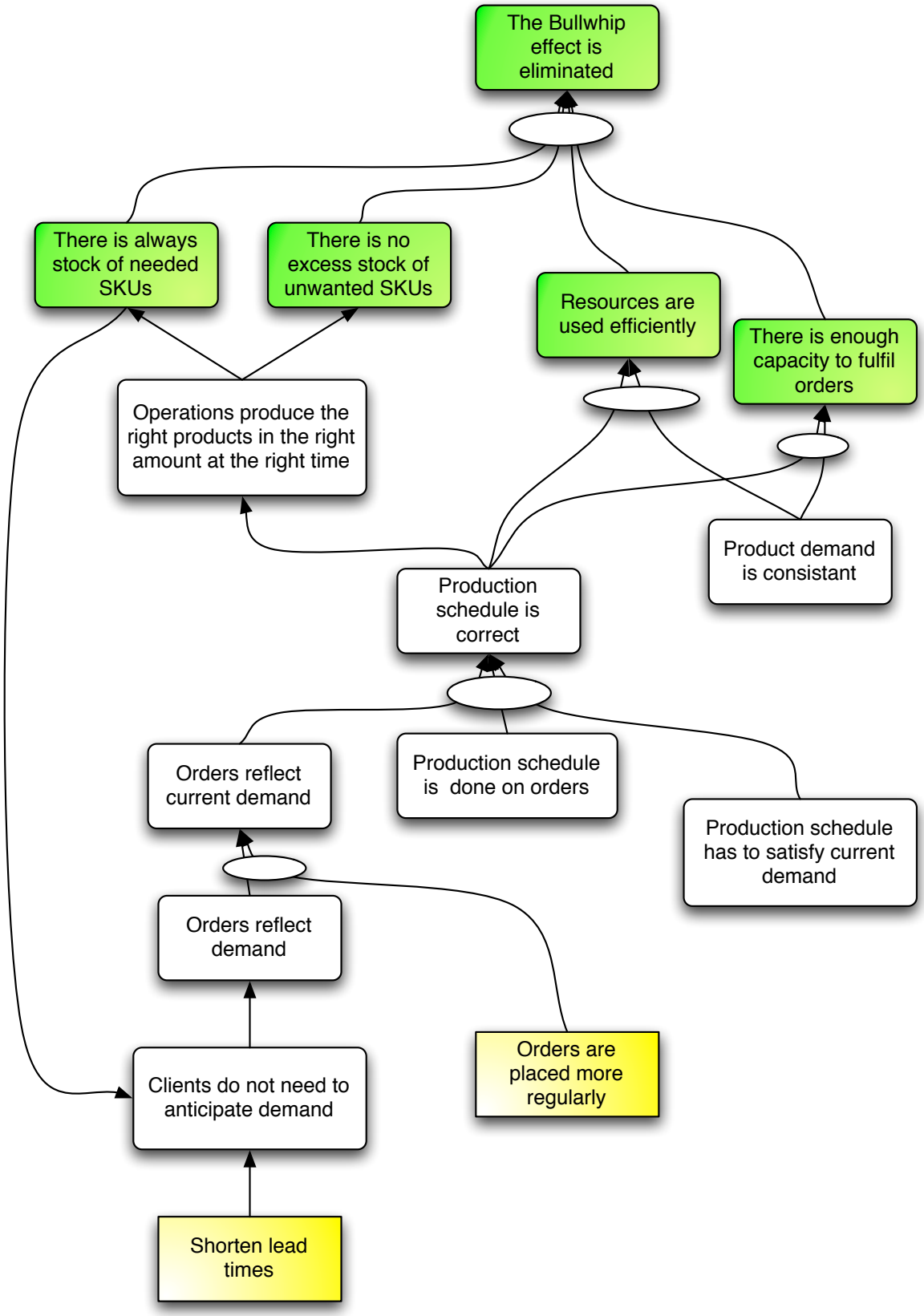


Figure 6.5: The FRT of the supply chain after the implementation of solutions

Table 6.1: Verification results

Solution	Safety Property Verification $\Box$ (*Orders* $\leftrightarrow$ *Demand*)	Liveness Property Verification $\Box$ [max (*PD*) $\leq$ max ($k \times CD$)] , $k = 1$
A	✓	✗
B	✓	✓

The FRT of the solution is given in Figure 6.5. In the FRT we see that one of the predicted desirable effects is that the bullwhip effect will disappear from the system. This implies that demand on production will level out. The demand will not be considerably more than the actual maximum of product consumed per time period. One of the trademark characteristics of a supply chain suffering from bullwhip is that demand on the factory is not correlated to actual consumption demand. So, in addressing the principal constraint we will not only eliminate the identified UDEs in Figure 6.1 but also resolve the bullwhip effect. We therefore define the measure of the bullwhip effect in the system as a relative measure of the maximum demand from clients versus the maximum demand placed on production.

At this stage we know that we have shown that we do not order more than is actually consumed by clients. What we want to verify is that the demand placed on the factory is in line with the demand of customers. So the liveness property that we verify for is that the maximum demand at the factory is always less than the maximum client demand multiplied by a factor k. This is expressed in LTL as

$$\Box\ [\max\ (PD) \leq \max\ (k \times CD)], \tag{6.2}$$

with $k \geq 1$, $PD = Production\ Demand$, and $CD = Client\ Demand$. The value of k expresses the degree of fluctuation allowed between these maximums. If verified with $k = 1$ the maximum demand at the factory will never be more than the maximum consumed by customers.

If verified against this liveness property in Equation 6.2 described above, S_a fails and B passes. A summary of the verification results is given in Table 6.1. The results are for a value of $k = 1$, and a check mark (✓) denotes verification success.

The verification from these solutions provides the following insights. Firstly we are capable of addressing the principal constraint of the problem that is faced. We are also not only able to address this principal constraint but do so verifiably. This means that both of these solutions are in a position to bring benefit to the system. With the safety properties verified, we are now able to measure S_a against S_b in terms of their verification on the liveness property expressed in Equation 6.2. The performance of the solutions in terms of their ability to curtail the bullwhip effect do, however, differ. With the strict measure of $k = 1$ only S_b verifies. With less stringent requirements on k, S_a verifies as well.

Further verification demonstrates that there exists a direct correlation between the chosen value of k, the lead time l in the system and the ability of a solution to pass verification. During the previous verification runs the value of l was kept constant; if it is altered, the outcome of the verification

Table 6.2: Verification results matrix for Equation (6.2) with l and k as variables.

Verification for: $\square$ [max (PD) ≤ max ($k \times CD$)]				
Solution	Value of k	l	$2l$	$3l$
A	1	✗	✗	✗
	2	✓	✗	✗
	3	✓	✓	✗
	4	✓	✓	✓
B	1	✓	✓	✓
	2	✓	✓	✓
	3	✓	✓	✓
	4	✓	✓	✓

is affected, and shows the correlation between the magnitude[1] of the bullwhip and lead time. The results are provided in Table 6.2.

6.5 Verification implications and conclusions

The Logical Thinking Process analysis of the problem identified the principal constraint that orders consist of both sold stock as well as forecasted demand. Both proposed solutions rectify this problem provably. The use of CAM enabled an entire supply chain to be abstracted to a level that it can be model checked. The reduction in complexity did not impede the ability of the model to provide valuable feedback, and decisions can be made on solution behaviour and suitability despite their abstract nature.

The failure of the S_a to verify for the given liveness property established the correlation between lead times and the bullwhip effect. It also shows that if complete transparency is obtained, as with S_b, then the bullwhip effect can be negated, even with long lead times. It also showed that if lead times are shortened then the bullwhip effect is also reduced for the value of $k = 1$. It is therefore possible to customise a solution so that increased costs of greater transparency and lead time reduction can be balanced to maximise the ROI of the system. These results echo the findings in the literature [24; 25; 26; 27] that reduction in lead times and the increase in supply chain transparency reduces the bullwhip effect. This was demonstrated by CAM and done with generic and lower complexity models when compared to simulation models presented in the literature.

[1] Of which k is the indicator; a larger k implies a larger variation.

CHAPTER 7

JIT VERSUS S-DBR COMPARISON

THE idea behind the of use CAM is to gain insight into the core problem faced by a system, to apply this insight to find a solution, and then finally verify if the proposed solution does in fact address the core problem. Folklore states that there are many ways to skin a cat, and in this sense all solutions can be considered equal, yet in practise some solutions are more equal than others[1], and the aim is to find the better solution out of a group of apparently equal options.

In this chapter we apply CAM to a common problem scenario faced by the manufacturing environment. In the process it will determine the principal constraint and formulate solutions to the problem, as was done in the previous chapters. The solutions verified in this chapter, JIT and S-DBR, are some of the main improvement and management techniques used in manufacturing today [28]. The aim is to determine which of these accepted and widely implemented solutions are best suited to address the problems faced by the hypothetical shop floor that is described in this chapter.

7.1 Production floor conditions

The first step in CAM is to identify the current UDEs in the system. The main concerns that are evident on the shop floor are as follows:

1. Too much overtime.
2. Frequent expediting of orders.
3. Large amounts of Work In Process (WIP).

[1] Unlike the pigs in George Orwell's, Animal Farm.

4. Missed shipments.

Once again the CRTs are marked in red and the completed CRT is given in Figure 7.1. From the CRT it is clear that there are two principal constraints. The first is the fact that the processes are interdependent, and the second that there is variability in the system. It is from these two principal constraints that the observed UDEs originate. To address these UDEs one of these principal constraints must be resolved. At this point in time the choice as to which of these two issues to address is left up to the modeller, since resolving either will bring benefit to the system, as explained in Chapter 2.

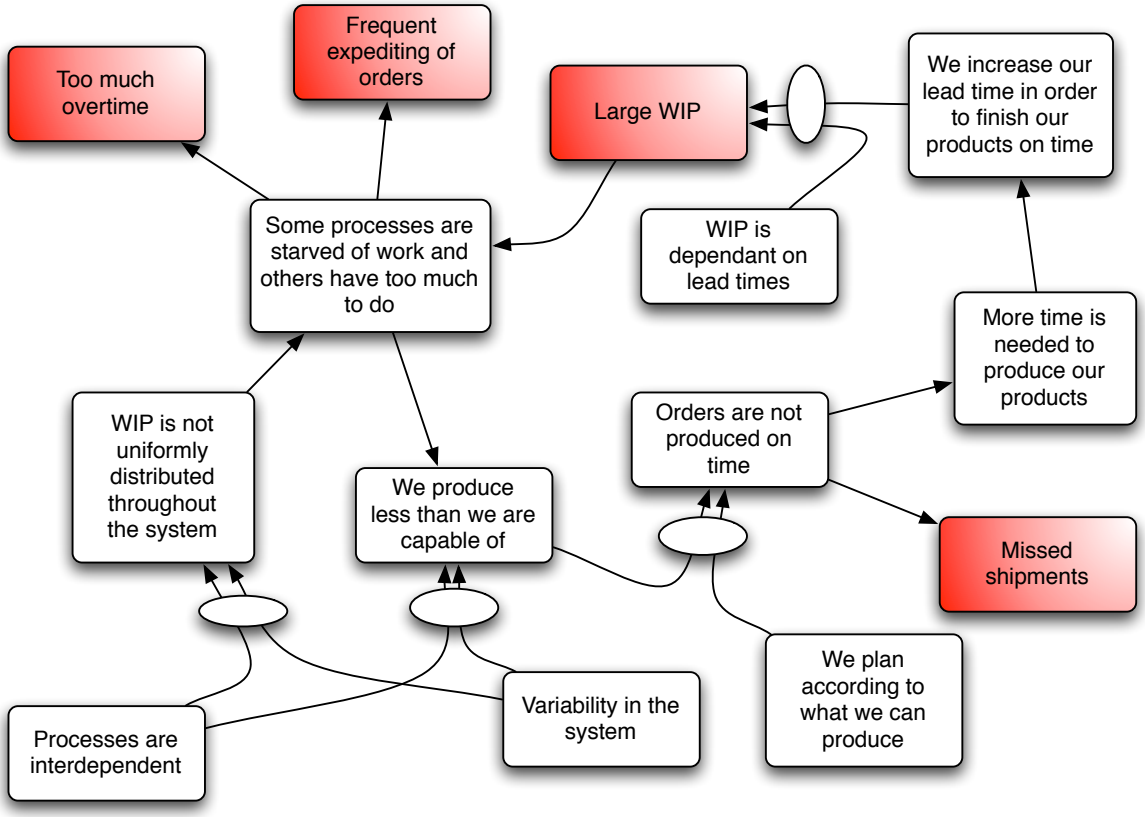


Figure 7.1: CRT of production prior to implementing of a solution

In this evaluation the principal constraint that will be focused on is the variability in the system. If the variation can be eliminated from the system then all the UDEs can be solved. So the criteria that any solution should meet is the reduction of variability in the system.

The issue with the removal of variability in the system is that it is needed in some cases. For example, if a client wishes to change an order or propose some sort of alteration, accommodating their

request requires variability in the process. To better depict the conflict, the problem is formulated in a Conflict Cloud provided in Figure 7.2.

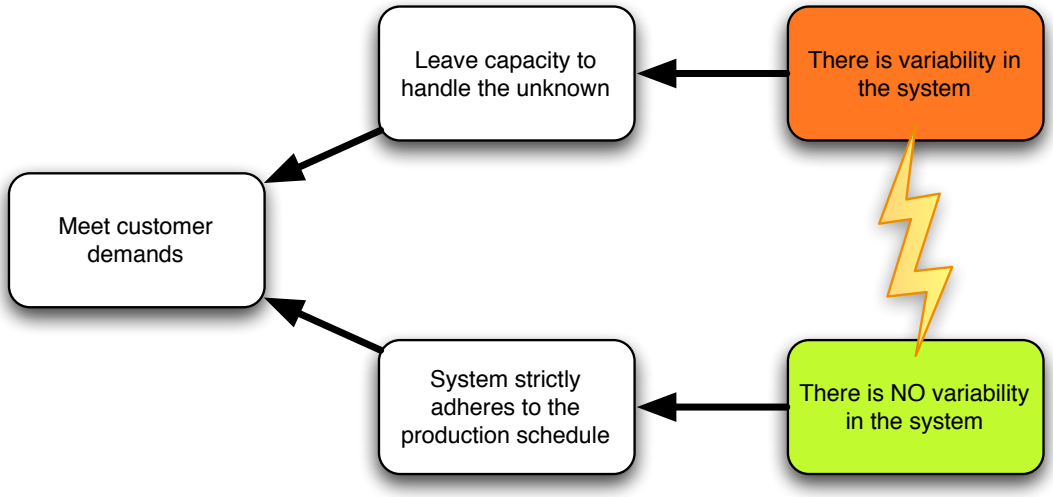


Figure 7.2: Conflict Cloud representing variability as principal constraint

The Conflict Cloud states that the main objective of the system is to meet customer demand. In order to meet this demand capacity is needed to handle the unknown, which in turn implies variability in the system. On the other hand, if there is no variability in the system then schedules can strictly be adhered to, and in doing so customer demand is met.

What injection would break the conflict in the system? The case has been made that not all variability is bad, though most variability usually is. One has to ask oneself, does a customer really care how often the product was late during assembly or construction, if it is delivered on time? So a solution to the problem must address, or preferably eliminate the variation in deliveries to customers.

7.2 Potential solutions

There exist improvement methods and frameworks that address the problem of variability on the shop floor. We will evaluate two such solutions with CAM in the following pages. Firstly the Just In Time (JIT) method which forms part of the Toyota Production System [29, p.405], and secondly the Simplified Drum Buffer Rope (S-DBR).

7.2.1 JIT

JIT addresses variability in a system by decoupling dependant processes from each other by means of the Kanban system [30]. A Kanban is a measure of work that for a specific station, be it a sub-assembly or a number of parts that are grouped as a unit of work. A station is responsible for keeping the buffer of Kanbans full for its immediate downstream process. The protocol that a workstation of a shop floor running a JIT implementation should progress as follows.

1. If there is an open Kanban downstream, start production to fill it.
2. Continue producing until there is no more open downstream Kanbans.
3. If there are no open Kanbans do nothing.

This protocol is represented in FSM form in Figure 7.3.

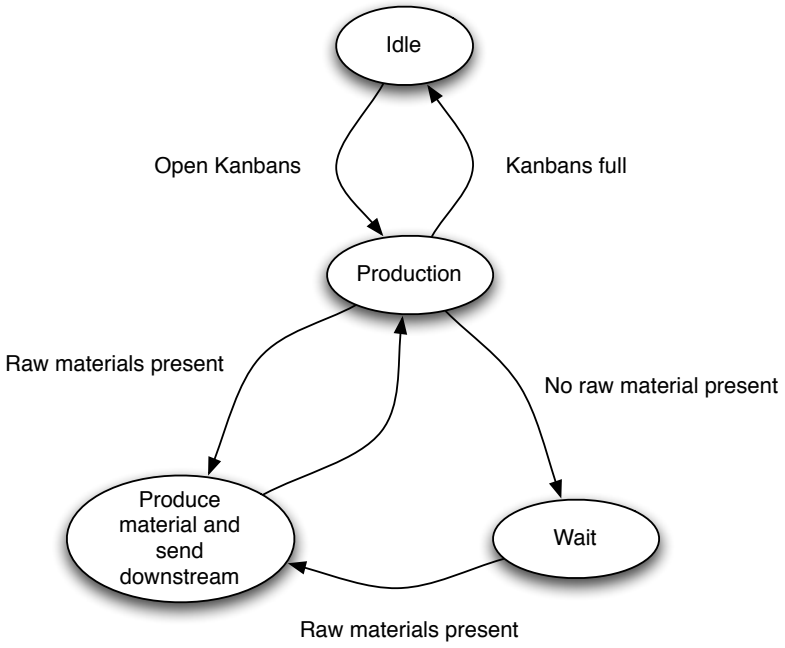


Figure 7.3: The FSM representation of the JIT shop floor protocol

JIT addresses the issue variability by decoupling the interdependent processes with the Kanban system. This places protective buffering that reduces the variability experienced by each of the individual processes.

7.2.2 S-DBR

The S-DBR solution does not address the variability of the process as a whole but rather focuses on the final shipping variability. Everything in the system is done so that no orders are shipped late. The solution consists of the following protocol [31, p. 149]:

1. If there is no work, do not go and look for work.
2. If work arrives start with it immediately.
3. If busy and work arrives with an earlier delivery date, then start with the job with the earliest delivery date.
4. If there is more than one job in the queue then start on the job of greatest importance first.

The protocol is transformed into an FSM and is given in Figure 7.4.

7.3 Safety property formulation and solution verification

From the CRT we conclude that the variability in the system has to be addressed. Both JIT and S-DBR appears to be in a position to accomplish this. From the Conflict Cloud we draw the conclusion that not all variability needs to be addressed in order for the system to achieve its objective of meeting client demand. Any implemented solutions must address variability of on time delivery to customers, and preferably eliminate it. We define this safety property in LTL as

$$\Box \textit{ Deliver To Client On Time.} \tag{7.1}$$

JIT and S-DBR were subjected to verification under the safety property in Equation 7.1. The results were that JIT failed and S-DBR passed. A summary of the verification results are provided in Table 7.1.

The verification results are interpreted as follows. Even though JIT potentially possessed the means to address variability in the system, it is not capable of doing so consistently in the area where it is needed most, on time delivery to clients. S-DBR, in contrast, is verifiably capable of meeting this need. Therefore, given the set of UDEs which we originally wanted to eliminate from the system the solution best suited to accomplish this would be S-DBR.

In environments where lead times are cut to the extent that even minor fluctuations would result in an order being late the verification results do change. JIT still fails but so does S-DBR. The interesting thing is that even though the solutions fail to verify against the safety property in Equation 7.1, S-DBR does verify for

$$\Box\Diamond \textit{ Deliver To Client On Time.} \tag{7.2}$$

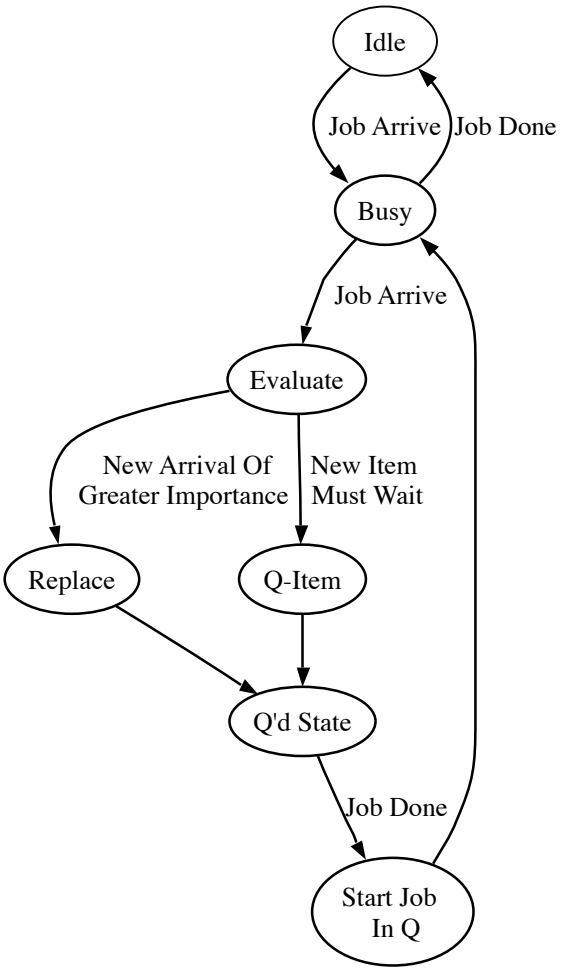


Figure 7.4: The FSM of the S-DBR protocol

Equation 7.2 is the logical extension of 7.1 based on our aim to reduce variability in on time delivery to clients. If we are not in a position to eliminate all variability in on time delivery of products to clients as described by the verification requirement in 7.1, then the second most restrictive LTL formula to describe the property will then be 7.2.

7.4 Results discussion

The system in question faced numerous problems. These problems were all relayed back to a common root cause. In this case it was the fact that variability exists in a system. In order to solve these problems this variability has to be addressed. Two potential solutions were put forward, JIT and S-DBR, both holding the potential to reduce this variability.

The implementation of both of these solutions, respectively, have produced impressive results in

Table 7.1: JIT and S-DBR verification results

Safety Property	JIT	S-DBR
□ *Deliver To Client On Time*	✗	✓
□◊ *Deliver To Client On Time*	✓	✓
With aggressive scheduling		
□ *Deliver To Client On Time*	✗	✗
□◊ *Deliver To Client On Time*	✓	✓
Minimised inventory		
□ *Deliver To Client On Time*	✗	✓
□◊ *Deliver To Client On Time*	✓	✓
With aggressive scheduling and minimised inventory		
□ *Deliver To Client On Time*	✗	✗
□◊ *Deliver To Client On Time*	✓	✓

practise, and are therefore capable of doing so. The aim of CAM is not to form an implementation methodology, but rather to match solutions to problems. This example once again demonstrated this. For the problems described in this scenario S-DBR will be more suited to address the needs in the system.

7.4.1 S-DBR recovery properties

S-DBR provably meets the critical objectives under a greater number and stricter operational requirements and JIT does not. We therefore, based on the verification of the solution ability to address the principal constraint, recommend that S-DBR be implemented over JIT.

S-DBR provably addresses the principal constraint under normal circumstances, as well as extreme conditions. The verification of S-DBR for Equation 7.2 provides additional insight into the behaviour of the model under extreme conditions. S-DBR is proved by Equation 7.2 to be capable of healing itself in all of the operational environments verified for in Table 7.1. In the event that the system starts to fall behind on orders, S-DBR will, by its very nature, bring the system on time without any additional external input.

This property has not been studied or documented in literature. This can possibly be attributed to the novelty of using model checking to verify business properties. Using verification instead of simulation in the analysis of protocols provides a different perspective and as a result insightful and otherwise overlooked provable properties.

7.4.2 The Lean paradox

The verification of JIT also brought insight not available with other techniques. From the verification runs it is evident that there exists a paradox in JIT, and more specifically the Lean[2] approach to manufacturing. Lean strives to eliminate all waste [32, p.205] from the system. One of the main categories of waste is inventory. From the verification results it is evident that this results in the failure of the manufacturing process to run smoothly under all conceivable circumstances. Though capable of recovering itself (like S-DBR) from backlog, it is still not capable of producing consistently on time.

The models constructed for model checking are highly abstract. The implementation of JIT in PROMELA describes the logical flow of the protocol. It is the simplest implementation model possible of the process; it is independent of the underlying process times and variability. This pure conceptual model analyses the *logic* of the protocol, and it is found to be self-contradictory. Only inventory should be kept that is needed to satisfy consumer demand, according to the elimination of waste philosophy. Seeing that the simplest form of demand then would be the existence of a single customer with only demand for a single product, inventory should be kept to meet client demands, and in this case inventory should be kept for a single customer. Under these conditions the process does not meet the always on time delivery objective. Increasing inventory to triple the required amount still does not provide the desired performance, whilst S-DBR provides this performance even at low inventory levels.

There are numerous Lean implementation success stories: this contradiction does not invalidate this. What it does point out is that there are mismatched assumptions in the approach. From the verification results it is deduced that the performance of the protocol is independent from the level of inventory. So regardless of the amount of surplus capacity, stock as well as constant demand, the process will always move out of bounds, but will always return back to normal. This has undoubtedly been compensated for in implementation but does not mean that the anomaly should be ignored. We use both quantum mechanics and the theory of relativity (though mutually exclusive) contradictory to great effect, yet the pursuit to unify them has never stopped. This paradox points to a fundamental problem, or incomplete understanding of the method and how its performance is related to inventory levels. From this suggestion it would appear that the level of inventory in a system is related to the protection of the system going out of bounds and is some form of an axiom. If this discrepancy can be understood and its effects explained there is room for improvement given the additional understanding gained from the resolution of this conflict.

This contradiction is not documented in literature. Given the prevalent modelling and analysis methods it does not come as a surprise. There exists room for expansion on this problem and its implications alone. This thesis does, however, not provide the platform to explore this avenue to the extent that does justice to the problem. It does, however, demonstrate that model checking is capable of providing insights into well-known and widely implemented solutions. One of the reasons

[2] JIT is one of the fundamental pillars upon which the Lean Production philosophy is built.

for the use of model checking was to expose subtle and inconspicuous flaws in processes. In this simple example, a serious[3] defect was exposed in aone of the most widely implemented production protocols globally.

7.4.3 Conclusion

CAM showed that it is capable of delving into the heart of complex problems, reducing them to simple focus points: the identification of potential solutions, and the verification of these solutions. It also showed how it is capable of a high level of abstraction so that these solutions can be verified. The abstraction is done with a common set of safety and liveness properties that makes different solutions directly comparable to each other. This comparison is based on criteria that are causally related back to all the main concerns in the system. In doing so it places the power of model checking in the hands of business process modellers. It is also capable of exposing logical conflicts and encompasses both policy philosophies and actual implementation. This provides a platform to investigate held assumptions regarding business processes that would otherwise be left unchecked.

[3] From a logical point of view JIT is not capable of delivering on time. This indicates that greater on time deliveries can be achieved with S-DBR when compared to JIT; this at lower inventory levels. both primary objectives of JIT.

CHAPTER 8

DISCUSSION

We now ask ourselves, what would CAM allow us to do tomorrow that we cannot do with business process modelling methodologies today? The answer to this question encapsulates the reason for the use of CAM and the benefit it will bring to a system.

8.1 Speed

Models built with CAM are highly abstract, and do not contain computational or operational details. It only concerns itself with the process logic that the system uses to execute. As a result, the models are small, and even models of large processes can be constructed in a minimal amount of time[1].

No time studies are needed to determine model parameters, since only logic governs the operation. This further reduces the modelling period because no process performance data is required for the construction of the model. In addition to this, many components in a business processes functions on the same logic. Therefore the logic is generic; it can be reused, instead of having to establish the performance parameters of every individual process, as one would have to do in simulation.

8.2 Accuracy

The use CAM at the outset of an improvement or change initiative tests the logic of the solution. In as much as it does not rely on statistical data to draw conclusions, it is solely focused on the suitability of the solution.

[1] Once familiar with model construction techniques the creation time for some of the models depicted in this thesis were completed in a matter of hours.

Parameter tweaking can be used to skew model behaviour in favour of certain solutions, or this can be merely due to an incorrect interpretation of the statistical significance. Taking the S-DBR versus JIT example, it would be possible to construct models that would show that JIT is faster than S-DBR in terms of execution, and vice versa. The use of CAM resolves this problem. It shows, independently from sub-process performance, which solutions have a superior ability to address the problem at hand. In the case of JIT and S-DBR, both are capable of healing themselves, but only S-DBR is capable of always delivering on time, form a logical point of view at least.

Armed with the proof of the logical soundness and superiority of a solution provided by verification, it is now possible to simulate the solution and tweak the variables so that performance of the final implementation is maximised. The result is therefore an optimised and verified solution, which compares favourably with normal unverified simulation solutions.

8.3 Direct solution comparison

Arguably there are many methods [33] that can be used to compare proposed solutions with each other. The benefit of CAM over other methods is that it ensures a common measurement for all proposed solutions. The measurement is the common problem all solutions should address if implemented. This problem is used as a measurement, and all models must be able to address this problem provably. If this was not possible, model checking could not be used to compare solutions with each other. This common measurement is inherent to the method and not optional, as is would be under normal modelling situations. A modeller will not be put in the position that would force him to choose between two models, each with its own unique set of advantages and disadvantages, and base his choice on a perceived, unproven, or a correlated measure of benefit.

8.4 CAM benefits

To summarise and answer what CAM allows us to do tomorrow what we could not do today:

CAM *allows for the quicker modelling of business processes, as well as providing problem specific and proven solutions in a manner not possible with simulation or other techniques. A combination that is not available in other methodologies today.*

CHAPTER 9

CONCLUSIONS

At the outset of this thesis the aim was to verify business processes with model checking. The reason was to prove the behavioural properties of proposed or implemented protocols to gain a better understanding of their operational dynamics. Especially dynamics left unobserved if evaluated with normal testing and simulation methods. A point of particular interest was the behaviour and testing of concurrent solutions, a field in which more and more business processes now operate in, and an area left sufficiently under-addressed by mainstream simulation.

Two obstacles in the way of the application of model checking to business processes are the lack of formality in business processes and the difficulty of abstracting verifiable models from complex and informal systems. This problem was addressed with the creation of CAM, which combines the Logical Thinking Process as an abstraction tool-set with the power and capabilities of model checking to meet the original objective.

In the examples of Chapters 6 and 7 we showed that the Logical Thinking Process can transform complex and heterogeneous systems into homogeneously formal and verifiable models. The use of the Logical Thinking Process not only provided abstract models of verifiable complexity, but also supplied the requirements necessary for the verification of the models. These requirements provided a common measurement matrix that is used for multiple potential solutions to the same problem, and as a result renders a means of direct comparison between solution alternatives.

An example of this is the common measurement used to compare the S-DBR protocol against JIT. Both methods have different backgrounds, approaches and operational philosophies giving rise to disjointed measurement[1] objectives. The Logical Thinking Process provided a common measure-

[1] S-DBR focuses on buffer and bottleneck performance, whilst JIT strives to eliminate waste like inventory, setup times and overproduction.

ment matrix for both solutions from a common set of objectives, allowing them to be compared directly with each other with the same measurements.

Providing model checking with these abstract and verifiable models allowed the business processes to be verified and have their behavioural properties formally proven. This process verified the known properties of these implemented and studied solutions, as well as demonstrated new properties not possible with other methods[2].

These properties were shown in Chapters 6 and 7 by means of the following listed examples. The demonstrable modelling and prediction of the bullwhip phenomenon in a supply chain. This model also showed the connection between lead times and order frequency with regard to the magnitude of the bullwhip effect. Chapter 7 demonstrated and verified the self-healing properties of S-DBR and JIT; attributes proven with the use of model checking for the first time. In the process we also showed that the performance of the JIT protocol is independent from the amount of WIP in the system from a temporal logic point of view. This raises questions regarding the governing principals and assumptions that are used in Lean manufacturing and TPS[3]. These assumptions and nuances are not necessarily detectable using other techniques.

The examples demonstrated that new insights can be obtained into the well-known and studied methods with the use of model checking. These insights are obtainable even with the use of highly abstract models. This shows that nuances and intricacies of procedures are detectable with model checking in a way that has not been possible with the current business process analysis methods.

9.1 Future work

The Logical Thinking Process forms part of a greater improvement and implementation strategy within TOC. In this work only a subsection of the complete process was used. It would therefore be possible to expand CAM and integrate model checking further with the Logical Thinking Process allowing for the verification of the both necessary as well as the sufficiency assumptions.

Given the results that have been obtained in the verification of the S-DBR, JIT and the bullwhip effect, it should provide new insights into the behaviour of standard and current solutions. Furthermore, it would be interesting to explore the apparent anomaly of the independence of WIP and process performance of the JIT protocol in more detail.

Having a usable tool for the analysis and verification of concurrent business processes, it would provide grounds for the investigation on how the alteration of current process and design techniques should change given the availability of model checking tools. Additionally, this could be employed to investigate how these tools can be used in the development and implementation of new processes, as well as to measure the benefits that they bring in tangible and quantifiable terms.

[2] Methods such as simulation and testing.
[3] The Toyota Production System.

LIST OF REFERENCES

[1] Inggs, C.P.: Model checking. Class Handouts—RW744 Concurrent Programming 2, University of Stellenbosch, February 2007.

[2] Popper, K.R.: *The logic of scientific discovery.* 2nd edn. Hutchinson, 1959.

[3] Vickers, J.: *The Problem of Induction.* Fall 2009 edn. Stanford University, 2009.
Available at: `http://plato.stanford.edu/entries/induction-problem`

[4] Goldratt, E.M.: *It's Not Luck.* North River Press, Jan 1994.
Available at: `http://books.google.com/books?id=c3cLAAAACAAJ&printsec=frontcover`

[5] Goldratt, E.M.: *Critical Chain.* North River Press, Jan 1997.
Available at: `http://books.google.com/books?id=FKimAILd3NsC&printsec=frontcover`

[6] Goldratt, E.M. and Cox, J.: *The Goal: An Ongoing Improvement Process.* Gower, Aldershot, 1984.

[7] Goldratt, E.M. and Cox, J.: *The Race.* North River Press, New York, 1986.

[8] Goldratt, E.M., Schragenheim, E. and Ptak, C.A.: *Necessary But Not Sufficient: a theory of constraints business novel.* North River Press, Jan 2000.
Available at: `http://books.google.com/books?id=3CrlAgAACAAJ&printsec=frontcover`

[9] Goldratt, E.M.: Computerized shop floor scheduling. *Intertional Journal of Production Research*, vol. 26, no. 3, pp. 443–455, 1988.

[10] Davies, J., Mabin, V.J. and Balderstone, S.J.: The theory of constraints: a methodology apart? a comparison with selected or/ms methodologies. *Omega, The International Journal of Management Science*, vol. 33, no. 6, pp. 506–524, 2005.
Available at: `http://www.sciencedirect.com/science?_ob=ArticleURL&_udi=B6VC4-4DFBS00-1&_user=10&_rdoc=1&_fmt=&_orig=search&_sort=`

d&view=c&_acct=C000050221&_version=1&_urlVersion=0&_userid=10&md5=e88ed6534a1a2bc433470405ea759f22

[11] Rahman, S.: Theory of constraints: a review of the philosophy and its applications. *International Journal of Operations & Production Management*, vol. 18, no. 4, pp. 336–355, 1998.

[12] Scheinkopf, L.J.: *Thinking for a Change: Putting the TOC Thinking Processes to Use*. CRC Press, Jan 1999.
Available at: http://books.google.com/books?id=r16Hzl4eJagC&printsec=frontcover

[13] Dettmer, H.W.: *The Logical Thinking Process: A Systems Approach to Complex Problem Solving*. ASQ Quality Press, Jan 2007.
Available at: http://books.google.com/books?id=hVcZvZawBzEC&printsec=frontcover

[14] Holzmann, G.J.: *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Jan 2004.
Available at: http://books.google.com/books?id=NpBdZKmOHO8C&printsec=frontcover

[15] Holzmann, G.: The model checker spin. *IEEE Transactions On Software Engineering SE*, vol. 23, no. 5, pp. 279–295, May 1997.
Available at: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=588521

[16] Baumler, S., Balser, M., Dunets, A., Reif, W. and Schmitt, J.: Verification of medical guidelines by model checking–a case study. *Proceedings of 13th International SPIN Workshop*, vol. 3925/2006, pp. 219–233, 2006.
Available at: http://www.springerlink.com/index/03u1422628280542.pdf

[17] Koehler, J., Tirenni, G. and Kumaran, S.: From business process model to consistent implementation: A case for formal verification methods. *Enterprise Distributed Object Computing Conference, Lausanne, Switzerland*, p. 96, Mar 2003.

[18] Janssen, W., Mateescu, R., Mauw, S., Fennema, P. and van der Stappen, P.: Model checking for managers. *Theoretical and Practical Aspects of SPIN Model Checking*, pp. 92–107, Jan 1999.
Available at: http://books.google.com/books?hl=en&lr=&ie=UTF-8&id=69qkrb8SZCQC&oi=fnd&pg=PA92&dq=rDpMGQiYfEAJ:scholar.google.com/&ots=8z6Nyc8Fan&sig=wB645Raj6Nsu2XPa5EGJ3ecwwBA

[19] Eertink, H., Janssen, W., Luttighuis, P., Teeuw, W. and et al.: A business process design language. *Proceedings of the 1st World Congress on Formal Methods*, vol. 1708/1999, p. 708, Jan 1999.
Available at: http://www.springerlink.com/index/cvpjfyq43ku2ktu4.pdf

[20] Janssen, W., Mateescu, R., Mauw, S. and Springintveld, J.: Verifying business processes using spin. *Proceedings of the 4th International SPIN Workshop*, Nov 1998.
Available at: ftp://ftp.inrialpes.fr/pub/vasy/presentations/Mateescu-SPIN-98.pdf

[21] Lu, R. and Sadiq, S.: *A Survey of Comparative Business Process Modeling Approaches*, vol. 4439/2007 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 2007.

[22] Dimitrov, D.: Discussion held with study leader regarding the suitability of model checking as a modelling tool. 2007.

[23] Forrester, J.W.: *Industrial Dynamics*. MIT Press, New York, 1964.

[24] Chen, F., Drezner, Z., Ryan, J. and Simchi-Levi, D.: Quantifying the bullwhip effect in a simple supply chain: The impact of forecasting and lead times. *Management Science*, vol. 46, no. 3, pp. 436–443, Mar 2000.
Available at: http://links.jstor.org/sici?sici=0025-1909(200003)46%253A3%253C436%253AQTBEIA%253E2.0.CO%253B2-M

[25] Lee, H., Padmanabhan, V. and Whang, S.: The bullwhip effect in supply chains. *Sloan Management Review*, Jan 1997.
Available at: http://www.uta.edu/faculty/swafford/opma3308/BullWhipEffect.pdf

[26] Lee, H., Padmanabhan, V. and Whang, S.: Information distortion in a supply chain: The bullwhip effect. *Management Science*, Jan 1997.
Available at: http://links.jstor.org/sici?sici=0025-1909(199704)43%253A4%253C546%253AIDIASC%253E2.0.CO%253B2-6

[27] Torres, O.A.C. and Morán, F.A.V.: *The Bullwhip Effect in Supply Chains: A Review of Methods, Components and Cases*. Palgrave Macmillan, Jan 2006.
Available at: http://books.google.com/books?id=syFOHQAACAAJ&printsec=frontcover

[28] Gupta, M.C. and Boyd, L.H.: Theory of constraints: a theory for operations management. *International Journal of Operations & Production Management*, vol. 28, no. 10, pp. 991–1012, 2008.

[29] Jacobs, F.R., Chase, R.B. and Aquilano, N.J.: *Operations & Supply Management*. Twelfth edition edn. McGraw-Hill, 2009.

[30] Gross, J.M. and McInnis, K.R.: *Kanban Made Simple: Demystifying and Applying Toyota's Legendary Manufacturing Process*. AMACOM, New York, NY, USA, 2003.

[31] Schragenheim, E. and Dettmer, H.W.: *Manufacturing at Warp Speed: Optimizing Supply Chain Financial Performance*. CRC Press, Jan 2000.
Available at: `http://books.google.com/books?id=44sqyByWIqQC&printsec=frontcover`

[32] Cachon, G. and Terwiesch, C.: *Matching Supply With Demand – An Introduction to Operations Management*. International edition edn. McGraw-Hill, 2009.

[33] Kettinger, W.J., Teng, J.T.C. and Guha, S.: Business process change: A study of methodologies, techniques, and tools. *Management Information Systems Quarterly*, vol. 21, no. 1, pp. 55–80, 1997.

Appendices

APPENDIX A

PROMELA CODE FOR BANK EXAMPLE IN CHAPTER 5

```
1
/* The Model of the Bank process without an ATM
3     Created 9/09/2009
      Jock Odendaal
5 */

7 /* Definition of state variables */

9 #define Idle 1
  #define Bank 2
11 #define ATM 3
  #define Wait 4
13
  bool p
15


17 never {
     do
19   ::p -> break
     ::else
21   od
  }
23


25 active proctype BankATM(){

27 bool Need = 0;
```

```
bool BankAvailable = 1;
byte State = Idle;
p = 0;


do
::     if
       ::(State==Idle)->
              if
              ::(Need && BankAvailable) -> State = Bank;
              ::(Need && !BankAvailable) -> State = Wait;
              ::else
              fi;
       ::(State==Bank) -> State = Idle;
       ::(State==Wait) -> p=1;
              if
              :: (BankAvailable) -> State = Bank;
              :: else
              fi;
        fi
::Need = 1
::Need = 0
::BankAvailable = 1
::BankAvailable = 0
od

}
```

Listing A.1: PROMELA code for the FSM in Figure 5.12 of the bank example

```

/* The Model of the Bank process with an ATM
    Created 9/09/2009
    Jock Odendaal
*/

/* Definition of state variables */

#define Idle 1
#define Bank 2
#define ATM 3
#define Wait 4

bool p

```

```
never {
  do
  ::p -> break
  ::else
  od
}


active proctype BankATM(){

bool Need = 0;
bool BankAvailable = 1;
bool ATMAvailable = 1;
byte State = Idle;
p = 0;


do
::     if
       ::(State==Idle)->
              if
              ::(Need && BankAvailable) -> State = Bank;
              ::(Need && ATMAvailable) ->  State = ATM;
              ::(Need && !ATMAvailable && !BankAvailable)
                     -> State = Wait;
              ::else
              fi;
       ::(State==Bank) -> State = Idle;
       ::(State==ATM) -> State = Idle;
       ::(State==Wait) -> p=1;
              if
              :: (BankAvailable) -> State = Bank;
              :: (ATMAvailable) -> State = ATM;
              :: else
              fi;
  fi
::Need = 1
::Need = 0
::BankAvailable = 1
::BankAvailable = 0
od

}
```

Listing A.2: PROMELA code for the FSM in Figure 5.13 of the bank example

APPENDIX B

PROMELA CODE FOR S_a

```
 /* This module is to demonstrate the basic consept of bullwhip
  within an ordenary supply chain
  Created by Jock Odendaal
    11 November 2007
*/

/* Definision of the state variables that will be used in the
if Statements*/
#define RecieveOrder 1
#define RecieveStock 2
#define OrderInventory 3
#define Deliver 4
#define Idle 5
#define StockLevel 5
#define p (JobNo <= 15)

/* Definition of personal varialbes*/
mtype = { order, delivery };
byte JobNo;

never {    /* ![]p */
T0_init:
  if
  :: (! ((p))) -> goto accept_all
  :: (1) -> goto T0_init
  fi;
accept_all:
```

```
28    skip
    }
30
32
    proctype node(chan cl, sup)
34  {
      byte State;
36    byte Current;
      short Stock;
38    byte Temp;
      byte Ship;
40    short StockOrderAmount;
      State = Idle;
42    Stock = 5;
      do
44    ::  if
        :: (State == Idle) -> /* Idle state if
46                  there is no work */
          atomic{
48          if
            ::cl?order (Current)->
50            d_step{
                State = RecieveOrder;
52              printf("Recived Order of:
                    %d\n" ,Current );
54            }
            ::sup?order (Current)->
56            d_step{
                State = RecieveStock;
58              printf("Stock is %d\n",Stock )
              }
60          ::true;
              d_step{
62            State = OrderInventory;
            }
64          fi;
          }
66      :: (State == RecieveStock)->
          d_step{
68          Stock = Stock + Current;
            State = Idle;
70        }
        :: (State == RecieveOrder)->
72        atomic{
          if
```

```
74          :: (Current > Stock)->
              d_step{
76              if
                ::(Stock <= 0)->
78                Stock = Stock-Current;
                  Ship = 0;
80              ::(Stock > 0)->
                  Ship = Stock;
82                Stock = Stock -Current;
                fi;
84              printf("Stock is: %d\n",Stock);
                State = Idle;
86              printf("Shipped to client: %d\n", Ship);
              }
88            cl!order, Ship;
            :: (Current <= Stock) ->
90            d_step{
                  Stock = Stock - Current;
92                State = Idle;
                  printf("We Shiped: %d\n",Current);
94              }
              cl!order, Current;
96          fi;
            }
98        :: (State == OrderInventory)->
            atomic{
100         d_step{
              StockOrderAmount = StockLevel - Stock;
102           State = Idle;
              printf("Ordering Inventory to the value of:
104               %d\n",StockOrderAmount)
            }
106         sup!order, StockOrderAmount;
          }
108       fi
      od
110   }

112   proctype manufacture(chan manu)
      {
114     do
        ::if
116       ::manu?order (JobNo)->manu!order, JobNo;
          fi;
118     od;
```

```
120  }

122  proctype client(chan customer)
     {
124    byte Temp;
       do
126    :: customer!order, (6);
       :: customer!order, (5);
128    :: customer!order, (4);
       :: customer!order, (3);
130    ::if
         ::customer?order(Temp)
132      fi;
       od;
134  }

136  init
     {
138    chan num[10] = [0] of {mtype, byte}
       atomic
140    {
         run client(num[0]);
142      run manufacture(num[5]);
         run node(num[0], num[1]);
144      run node(num[1], num[5]);

146    }
     }
```

Listing B.1: S_a PROMELA code

APPENDIX C

PROMELA CODE FOR S_b

```
/* Definition of the state variables that will
be used in the If Statements*/
#define RecieveOrder 1
#define RecieveStock 2
#define OutOfStock 3
#define Deliver 4
#define Idle 5
#define p (JobNo <= 6)

/* Definition of personal variables*/
mtype = { order, delivery };
byte JobNo;

never {    /* ![]p */
T0_init:
  if
  :: (! ((p))) -> goto accept_all
  :: (1) -> goto T0_init
  fi;
accept_all:
  skip
}

proctype node(chan c1, sup)
{
  byte State;
  byte Current;
```

```
  short Stock;
29 byte Temp;
  State = Idle;
31 Stock = 6;
  do
33 ::  if
    :: (State == Idle) -> /* Idle state if there is no work */
35    if
      ::cl?order (Current)->
37      d_step{
          State = RecieveOrder;
39        /*  printf("Recieved Order of: %d\n" ,Current );
          */}
41    ::sup?order (Current)->
        d_step{
43        Stock = Stock + Current;
          printf("Stock is %d\n",Stock )
45      }
      fi;
47  :: (State == RecieveOrder)->
      if
49    :: (Current > Stock)->

51      d_step{
          Temp = Current - Stock;
53        State = Idle;
          printf("Out of stock ordered: %d\n",Temp, Stock)
55      }
        sup!order, Temp;
57    :: (Current <= Stock) ->
        d_step{
59        Stock = Stock - Current;
          State = Idle;
61        /*printf("Ordered Replacement stock: %d\n",Current)
          */}
63      sup!order, Current;
        cl!order, Current;
65    fi
    fi
67 od
}
69

proctype manufacture(chan manu)
71 {
  do
73 ::if
```

```
    ::manu?order (JobNo)->manu!order, JobNo;
75    printf("manufactured : %d\n",JobNo)
    fi;
77  od;

79 }

81 proctype client(chan customer)
   {
83   byte Scratch;
     do
85   :: customer!order, 6;   printf("Ordered 6\n")
     :: customer!order, 5; printf("Ordered 5\n")
87   :: customer!order, 4; printf("Ordered 4\n")
     :: customer!order, 3; printf("Ordered 3\n")
89   :: customer!order, 2; printf("Ordered 2\n")
     :: customer!order, 1; printf("Ordered 1\n")
91   ::if
       ::customer?order(Scratch) -> printf(" Received order of")
93     fi;
     od;
95 }

97 init
   {
99   chan num[10] = [0] of {mtype, byte}
     atomic
101  {
       run client(num[0]);
103    run manufacture(num[5]);
   /*    run node(num[0], num[5]);
105 */    run node(num[0], num[1]);
       run node(num[1], num[2]);
107    run node(num[2], num[3]);
       run node(num[3], num[5]);
109  }
   }
```

Listing C.1: S_b PROMELA code

APPENDIX D

SPIN RESULTS USED IN CHAPTER 6

This appendix contains the output of the various verification runs described in Chapter 6.

```
(Spin Version 5.1.6 -- 9 May 2008)
  + Partial Order Reduction

Full statespace search for:
  never claim           +
  assertion violations  + (if within scope of claim)
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  - (disabled by never claim)

State-vector 128 byte, depth reached 8290, errors: 0
   437923 states, stored
    73811 states, matched
   511734 transitions (= stored+matched)
        5 atomic steps
hash conflicts:     39851 (resolved)

   61.094 memory usage (Mbyte)

unreached in proctype node
  line 75, state 33, "-end-"
  (1 of 33 states)
unreached in proctype manufacture
  line 86, state 9, "-end-"
  (1 of 9 states)
unreached in proctype client
```

```
  line 102, state 20, "-end-"
  (1 of 20 states)
unreached in proctype :init:
  (0 of 8 states)

pan: elapsed time 1.14 seconds
pan: rate 384142.98 states/second
Theodore:BullwhipSpin jock$ cc -DSAFETY -o  pan pan.c
Theodore:BullwhipSpin jock$ ./pan
pan: invalid end state (at depth 51)
pan: wrote BullSupply2.trail

(Spin Version 5.1.6 -- 9 May 2008)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           - (none specified)
  assertion violations  +
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  +

State-vector 100 byte, depth reached 53, errors: 1
       42 states, stored
       11 states, matched
       53 transitions (= stored+matched)
        3 atomic steps
hash conflicts:         0 (resolved)

    2.501 memory usage (Mbyte)


pan: elapsed time 0 seconds
```

Listing D.1: SPIN output for S_a

```
(Spin Version 5.1.6 -- 9 May 2008)
  + Partial Order Reduction

Full statespace search for:
  never claim           +
  assertion violations  + (if within scope of claim)
  cycle checks        - (disabled by -DSAFETY)
  invalid end states  - (disabled by never claim)

State-vector 128 byte, depth reached 8290, errors: 0
```

```
   437923 states, stored
    73811 states, matched
   511734 transitions (= stored+matched)
        5 atomic steps
hash conflicts:     39851 (resolved)

   61.094 memory usage (Mbyte)

unreached in proctype node
  line 75, state 33, "-end-"
  (1 of 33 states)
unreached in proctype manufacture
  line 86, state 9, "-end-"
  (1 of 9 states)
unreached in proctype client
  line 102, state 20, "-end-"
  (1 of 20 states)
unreached in proctype :init:
  (0 of 8 states)

pan: elapsed time 1.15 seconds
pan: rate 380802.61 states/second
```

Listing D.2: SPIN output for S_b numbers

APPENDIX E

PROMELA CODE FOR S-DBR

```
/* This is the spin model of the buffer of that is used in TOC
  Created  24/04/2007
  Jock Odendaal
*/

/* Definition of the state variables that will be used in
 the if Statements*/
#define Idle 1
#define Busy 2
#define Evaluate 3
#define Replace 4
#define QState 5
#define StartJobInQ 6
#define p (JobNo <= 1)

/* Definition of personal variables*/
mtype = { job, drum };
byte JobNo;

/*chan tp = [0] of {mtype, byte};
chan fp = [0] of {mtype, byte};
chan num[10] = [0] of {mtype, byte};*/


never {    /* <>[]!p */
T0_init:
        if
```

```
28          :: (! ((p))) -> goto accept_S4
            :: (1) -> goto T0_init
30          fi;
accept_S4:
32          if
            :: (! ((p))) -> goto accept_S4
34          fi;
}
36

38 /* This process is the sole purpose of version. This is the
     fundamental building-block of the model.It models the basic
40   protocol for each workstation in both DBR and SDBR and had to
     be built before the rest of the model
42 */
proctype process(chan tp, fp)
44 {
     byte State;
46   byte Temp;
     byte current;
48   byte Qed;
     State = Idle;
50   do
     ::  if
52     :: (State == Idle) -> /* Idle state if there is no work
                  to be done*/
54       tp?job (current);
         d_step{
56         State = Busy;
           current--;
58         printf("Recived Job: %d\n" ,current );
         }
60     :: (State == Busy) -> /* Only one job is at the work
                  station and it is processed*/
62       if
         ::fp!job (current) -> State = Idle;
64       ::tp?job (Qed) -> State = Evaluate;
         fi;
66
       :: (State == Evaluate) -> /* If a new job arrives, it
68                    has to be decided if it more
                      important than the current
70                    one*/
         if
72       :: (current > Qed) ->
           d_step{
```

```
74            current--;
              Qed--;
76            State = Replace;
              printf("Recived Job: %d\n",Qed);
78            printf("Have Job %d\n",current);
            }
80        /* <= only used during v1 for simulation of how a
            process works*/
82        :: (current <= Qed) ->
            d_step{
84            State = QState;
              Qed--;
86            printf("Recived Job: %d\n",Qed);
              printf("Have Job %d\n",current);
88          }
        fi;
90
      :: (State == Replace) ->
92      d_step{
          Temp = current;
94        current = Qed;
          Qed = Temp;
96      }
        State = QState;
98      printf("Swop jobs\n")

100   :: (State == QState) ->
        if
102     ::fp!job, current->
          State = StartJobInQ;
104     fi;

106   :: (State == StartJobInQ) ->
        d_step{
108       current = Qed;
          current--;
110       State = Busy;
        }
112   fi;
    od;
114 }


116


118 proctype seed(chan tp)
    {
```

```
120   do
      :: tp!job, 6;
122   :: tp!job, 5;
      od;
124 }

126 proctype sink(chan fp)
    {
128
      do
130   :: fp?job, JobNo;
        printf("Jobdone %d\n", JobNo);
132     printf(" Value of P %d\n", p)
      od;
134
    }
136
    init
138 {
      chan num[10] = [0] of {mtype, byte}
140   atomic
      {
142     run seed(num[7]);
        run sink(num[0]);
144     run process(num[7], num[2]);
        run process(num[2], num[3]);
146     run process(num[3], num[0]);

148   }
    }
```

Listing E.1: S-DBR PROMELA code

```
1 (Spin Version 5.1.6 -- 9 May 2008)
    + Partial Order Reduction
3
  Full statespace search for:
5   never claim           +
    assertion violations  + (if within scope of claim)
7   cycle checks        - (disabled by -DSAFETY)
    invalid end states  - (disabled by never claim)
9
  State-vector 100 byte, depth reached 9999, errors: 0
11  2743594 states, stored
    2900371 states, matched
13  5643965 transitions (= stored+matched)
```

```
        3 atomic steps
hash conflicts:   3282938 (resolved)

  303.274 memory usage (Mbyte)

unreached in proctype process
  line 106, state 53, "-end-"
  (1 of 53 states)
unreached in proctype seed
  line 118, state 8, "-end-"
  (1 of 8 states)
unreached in proctype sink
  line 129, state 7, "-end-"
  (1 of 7 states)
unreached in proctype :init:
  (0 of 6 states)

pan: elapsed time 8.35 seconds
pan: rate 328574.13 states/second
```

Listing E.2: SPIN output for the S-DBR example

APPENDIX F

PROMELA MODELS AND VERIFICATION RESULTS FOR JIT

```
/* The Model of JIT Manufacturing protocol 13 January 2009
*/

/* Definition of state variables */

#define Idle 1
#define Production 2
#define Produce 3
#define Wait 4

bool p

never {    /* []<>p */
T0_init:
  if
  :: ((p)) -> goto accept_S9
  :: (1) -> goto T0_init
  fi;
accept_S9:
  if
  :: (1) -> goto T0_init
  fi;
}
```

```
26
proctype JIT(chan in, out)
28 {
bool Work = 0;
30 byte State = Idle;

32 do
::if
34   ::(State==Idle)->
       if
36     ::atomic{nfull(out) ->  State = Production};
       ::else
38     fi;
   ::(State==Production) ->
40     if
       :: (empty(in) && nfull(out)) -> State = Wait;
42     :: (nempty(in) && nfull(out))-> State = Produce;
       :: full(out) -> State = Idle
44     fi;
   ::(State==Produce) ->
46     in?Work;
       out!Work;
48     State = Production;
   ::(State==Wait) -> p=1;
50     if
       :: atomic{nempty(in) -> State = Produce};
52     :: else->
       fi;
54 fi;
od
56

}
58

proctype raw(chan out)
60 {
   do
62 :: nfull(out) -> out!1;
   od
64 }

66 proctype consume(chan in)
{
68   do
   ::  atomic{nempty(in) -> in?_};
70 ::  atomic{empty(in) -> p=1};
   ::  else;
```

```
  od
}

init
{
  chan num[10] = [3] of {bool};
  p=0;
  atomic
  {
    run raw(num[0]);
    num[0]!1;
    num[0]!1;
    num[0]!1;
    num[1]!1;
    num[1]!1;
    num[1]!1;
    num[2]!1;
    num[2]!1;
    num[2]!1;
    num[3]!1;
    num[3]!1;
    num[3]!1;
    run consume(num[3]);
    run JIT(num[0], num[1]);
    run JIT(num[1], num[2]);
    run JIT(num[2], num[3]);
  }
}
```

Listing F.1: JIT PROMELA code

```
hint: this search is more efficient if pan.c is compiled -DSAFETY
warning: for p.o. reduction to be valid the never claim
must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)
pan: claim violated! (at depth 561)
pan: wrote JITMK1.trail

(Spin Version 5.2.2 -- 7 September 2009)
Warning: Search not completed
  + Partial Order Reduction

Full statespace search for:
  never claim           +
  assertion violations  + (if within scope of claim)
  acceptance   cycles   - (not selected)
```

```
  invalid end states  - (disabled by never claim)

State-vector 160 byte, depth reached 561, errors: 1
      271 states, stored
      174 states, matched
      445 transitions (= stored+matched)
       20 atomic steps
hash conflicts:         0 (resolved)

    4.653 memory usage (Mbyte)


pan: elapsed time 0.01 seconds
```

Listing F.2: SPIN output for the JIT model