Consistency-Based Diagnosis Using Dynamic Models

by

Andrew Watkins

B.Eng., University of Victoria, 1990

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF
Master of Applied Science
in
THE FACULTY OF GRADUATE STUDIES
(Department of Electrical and Computer Engineering)

We accept this thesis as conforming
to the required standard

The University of British Columbia
September 1999
© Andrew Watkins, 1999
In presenting this thesis in partial fulfilment of the requirements for an advanced degree at the University of British Columbia, I agree that the Library shall make it freely available for reference and study. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by the head of my department or by his or her representatives. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Department of Electrical & Computer Engineering
The University of British Columbia
Vancouver, Canada

Date Sept 10 1999
Abstract

As society grows ever more reliant on increasingly complex technology, so does the importance of being able to quickly detect and diagnose the inevitable failures. Thus, the relatively new field of computer-assisted diagnosis has emerged to help fill this need. To date, expert-systems form the most common paradigm for computer-assisted diagnosis but another approach, that of model-based diagnosis (MBD), is superior in many respects and offers several advantages over expert-systems. These advantages stem from the fact that MBD uses system models that capture underlying information about both the behaviour and the structure of an artifact. This representation of the artifact allows unanticipated faults to be diagnosed, the system to be simulated, reactions to unusual inputs predicted and makes possible reactive, real-time system control.

However, the most important and difficult aspect of MBD is creating useable models; this thesis presents a new approach to modeling for diagnosis that allows simple and declarative models to be written that accurately reflect the behaviour of the artifact of interest. This is done by applying the technique of consistency-based diagnosis (one form of MBD) to the logic-based languages cc and tcc. The cc language allows models to be written using concurrent logic agents and constraints. Tcc extends this by allowing temporal models and default logic to be expressed. The application of consistency-based diagnosis to cc represents an important contribution to the field of model-based diagnosis because the concurrent nature of these languages correspond well to the multiple, concurrently operating components found in most artifacts and their use of constraints allow flexible, qualitative models to be written. These benefits are extended by tcc, which allows temporal models exhibiting default behaviour to be realized. Together, cc and tcc form a new approach to model-based diagnosis that is declarative, expressive, flexible and able to diagnose temporal models with timeouts.
Contents

Abstract ii

Contents iii

List of Tables v

List of Figures vi

Acknowledgements viii

Dedication ix

1 Introduction 1
  1.1 Model-Based Diagnosis 1
  1.2 Model-Based Diagnosis Using tcc 3
    1.2.1 The cc Modeling Language 3
    1.2.2 The tcc Modeling Language 4
  1.3 Objectives 5
  1.4 Summary of Results 5
  1.5 Chapter Overview 7

2 Background 8
  2.1 Consistency-Based Diagnosis 8
  2.2 Introduction to cc and tcc 11
  2.3 Diagnosis of Dynamic Systems 15
  2.4 Other Approaches 16
List of Tables

3.1 $Clp(P)$ Entailment Rules ........................................ 23

4.1 Full-Adder Execution Trace ...................................... 34
4.2 Valid Signals and Values for the LEGO Factory ............... 45
4.3 LEGO Controller Commands ...................................... 47
4.4 Observations of Correct Behaviour for Untimed and Timed Models .................. 57
List of Figures

2.1 cc Program Syntax .......................... 13
2.2 tcc Program Syntax ......................... 14
2.3 Definition of a JK Flip-Flop Written in tcc. .......................... 14
2.4 Definitions of tcc combinators ..................... 15

3.1 AND-OR Circuit. ................................ 23
3.2 Cc Program for the AND-OR Circuit. ................. 23
3.3 Representation of $P_\Delta$. ......................... 25
3.4 Delaying Theory Completion .................... 26
3.5 The Three-Bulb Circuit. .......................... 27
3.6 Agent for the Battery Component. ................ 28

4.1 Inverter With Delay ................................ 33
4.2 Full Adder with Gate Delays ....................... 34
4.3 Relationship Between States and Tcc Iterations .............. 35
4.4 State Diagram and Digital Circuit .................. 36
4.5 Tcc Model of the State Machine .................. 37
4.6 Interaction Between $clp(S)$, tcc and the Driver .......... 40
4.7 Motor-Controlled Pneumatic Valve .................. 41
4.8 Tcc Agent For Switching Pneumatic Valve Position .......... 41
4.9 Factory Conveyor Belt ............................ 42
4.10 Tcc Agent For Controlling the Conveyor Belt ............ 42
4.11 Definitions of tcc combinators .................... 43
4.12 The LEGO Factory ............................... 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.13 Diagram of the Model Factory</td>
<td>46</td>
</tr>
<tr>
<td>4.14 Lego Factory System Description in tcc</td>
<td>50</td>
</tr>
<tr>
<td>4.15 The Six States of the Lego Factory</td>
<td>51</td>
</tr>
<tr>
<td>4.16 Simulation Output</td>
<td>53</td>
</tr>
<tr>
<td>4.17 Overview of the Diagnostic Process</td>
<td>54</td>
</tr>
<tr>
<td>4.18 Active and Reactive Models of the Arm Component</td>
<td>55</td>
</tr>
<tr>
<td>4.19 Flexible Conveyor Belt Agent</td>
<td>58</td>
</tr>
<tr>
<td>4.20 Reactive, Qualitative Factory System Description in tcc</td>
<td>61</td>
</tr>
</tbody>
</table>
Acknowledgements

I would like to express my sincere thanks to my supervisor, Dr. Greg Bond, for introducing me to this area of research, providing very substantial groundwork that really helped me to get up to speed quickly and for always giving me good advice whenever I reached what I thought was an impasse. I would similarly like to thank Dr. Peter Lawrence for his help, making it possible for me to complete my studies.

I have also received great support from my parents Dave and Lois, my sister Lori and cousin Mike Rooney. But most importantly, I would like to thank Denise for her patience and faith.

Finally, I would like to thank some of my lab mates who have really helped me: Ying Fang, Kendra Cooper, Maryam Moghaddas, Buke Otkunc and Paria Noorkami.

ANDREW WATKINS

The University of British Columbia
September 1999
To Denise.
Chapter 1

Introduction

The field of computer-aided diagnosis is an expanding discipline with great relevance to today's highly complex world. Within this field, model-based diagnosis is emerging as the most promising technique for diagnosing faults in increasingly complicated, dynamic systems. However, it has been 'emerging' for at least the past 15 years with few successful industrial applications. This lack of success is due, in part, to the difficulty in creating suitable models with current techniques.

This chapter provides a brief introduction to the concepts of model-based diagnosis. The limitations of current techniques are then outlined and new modeling languages for diagnosis are proposed. These are the languages $cc$ and $tec$. The objectives of this thesis are then outlined, followed by the results that were achieved.

1.1 Model-Based Diagnosis

Every day our society becomes more dependent on technology. With it many old industries, such as saw mills, have improved efficiency and increased profits. In other situations (computer networks, for example), the technology itself has created whole new markets and the industries to serve them. In either case, the constant pressure to gain a competitive advantage and to improve productivity means that this technology is becoming both increasingly sophisticated and increasingly relied upon.

However, along with these complex systems come the inevitable failures: machines break down, circuits fail and software bugs crop up. With such a reliance on technology, diagnosing and fixing these faults quickly has become extremely important and yet, at the same time it has become increasingly difficult. The complex nature of today's devices means that a large amount of background
knowledge is required just to understand the problems that occur and the pervasiveness of technology means that more people than ever are required to have this knowledge. As diagnosing faults has become more important, the field of computer-assisted diagnosis has emerged to help deal with these problems.

Expert systems were the first and are still the most widely used foundation for computer-assisted diagnosis. Expert systems attempt to capture the diagnostic knowledge of a human expert by relating the symptoms of failures to the underlying faults and then storing this knowledge in a 'rulebase'. This knowledge can then be used in the future by presenting the expert system with observed symptoms and letting it use its rules to determine the most likely cause of the failure from its internal list of possible faults. However, expert systems have some inherent disadvantages. One is that capturing this diagnostic knowledge is not easy, as most human experts have difficulty translating their accumulated 'wisdom' into expert system rules. Another, more fundamental disadvantage, is that that expert systems generally contain knowledge only about how a system fails. This leaves important information about how the system normally behaves out of the expert system's knowledge base. Expert systems generally cannot deal with unanticipated faults because they have no understanding of the underlying structure or behaviour of the system.

In the early 1980's a new approach to diagnosis was developed through the work of de Kleer and Brown [9], Genesereth [13] and Davis [7]. This is the model-based approach which uses a representation, or model, of the correct behaviour of the system of interest. Therefore, while expert systems model how systems fail, the model-based approach captures knowledge of how systems work by modeling their structure and how they normally behave. This so called "deep knowledge" [25] is then used to predict the behaviour of the system. In the case of a failure, there will be a discrepancy between the normal behaviour predicted by the model and abnormal behaviour observed in the actual system. It is this discrepancy between the predicted behaviour and the observations that is used to form the diagnosis.

These ideas were clarified by Reiter [25] who detailed consistency-based diagnosis, which is one specific form of model-based diagnosis. With consistency-based diagnosis each component in the model is labeled as being either normal or abnormal. The discrepancy between the observed and expected behaviours is used to guide this labeling and if, by labeling certain components as abnormal the discrepancy can be explained, then these components form a diagnosis.

These researchers were all members of the artificial intelligence (AI) community, as diagnosis has always been an important area of interest in AI. In developing this model-based approach to
diagnosis, AI researchers built their models using logic, a tool they were very familiar with. Using logic as the modeling framework has many advantages: logic based models are precise and declarative and their underlying properties are well known.

Most applications of consistency-based diagnosis, however, use propositional logic to construct their models. Propositional logic has no variables and its simple form makes it easier to manipulate than higher-order forms of logic. However, propositional logic can be inflexible and lacks the expressiveness of higher-order logics such as predicate logic, which supports variables. For example, propositional logic can only model an inverter by enumerating all input/output combinations, such as

\[(\text{in}.0 \rightarrow \text{out}.1) \land (\text{in}.1 \rightarrow \text{out}.0)\].

This approach quickly breaks down when modeling a more complex system, such as a multiplier. However, predicate logic can easily describe a multiplier as

\[\forall X \forall Y \exists Z (\text{mult}(X, Y, Z) \leftarrow Z = X \ast Y)\].

Simple propositional logic also produces brittle models that are unable to reflect a range of possible behaviours and it has no intrinsic representation of time, needed to create temporal models.

Model-based diagnostic reasoning is only as good as the models used [8] and in order for consistency-based diagnosis to be successful, a flexible and expressive modeling framework is of primary importance.

1.2 Model-Based Diagnosis Using \textit{tcc}

1.2.1 The \textit{cc} Modeling Language

In the early 1990's a new approach to logic-based modeling appeared in the form of \textit{concurrent constraint} programming. The concurrent constraint (or more simply, \textit{cc}) approach is due to Saraswat [27] and represents the confluence of two ideas that, up until that point, had been separate areas of research in logic programming. These are the areas of concurrent logic programming and constraint logic programming.

Concurrent logic programming originated in the early 1980's [3] as an application of parallel execution of processes to logic programming. This new approach to logic programming however, was hindered by problems with communication and synchronization between the processes and by
inefficient execution of the goals [26]. Constraint logic programming, on the other hand, deals with using logic programming techniques to evaluate constraints [19, 20, 22]. Constraints are pieces of partial information, such as \( x < 5 \) or \( V = I \times R \), that restrict the values that variables can hold without necessarily providing specific values for those variables.

These concepts were combined in the concurrent constraint programming paradigm using a few simple ideas, one of which was to use a global store as a repository for storing all constraints. Multiple, concurrently executing logic programs, called agents, are then used to interact with this store, adding new constraints and checking if other constraints are satisfied by information already in the store. These two ideas turned out to be complementary. The parallelism of the concurrent agents and the flexibility of the constraints greatly increase the expressive power of the language while the global constraint store solves the problems of communication and synchronization among agents.

This new paradigm shows good potential for use as a logic-based modeling language for diagnosis. \( Cc \) agents implement first-order predicate logic, allowing greater expressiveness than is possible with simpler propositional logic and \( cc \)’s use of constraints allows flexible, qualitative models to be written. Also, the parallel nature of agents mirrors the parallel nature of components that make up a complex system and, finally, models written in \( cc \) are compositional, meaning that components can be mixed into the model just by adding new agents.

1.2.2 The \( tcc \) Modeling Language

\( Cc \) actually forms the basis for a family of languages, one of which is Timed Default \( cc \), or \( tcc \). In \( tcc \) the basic concepts of \( cc \) have been extended in two ways. The first way is with the addition of Reiter’s defaults [24]. Defaults specify an action to be taken when certain conditions are not met, similar to the \texttt{else} statement in standard procedural programming languages. Defaults are important because they allow the detection of missing information, something not possible with basic \( cc \). The second way in which \( tcc \) extends \( cc \) is with the addition of the notion of time to the language. This is because a \( tcc \) program is evaluated as a series of \( cc \) executions, each one representing a snapshot in time.

\( Tcc \) greatly improves the modeling potential of \( cc \). The addition of time allows temporal models to be expressed. This is important because most real-world systems change over time and, to be useful, a modeling language must be able to capture this. Defaults too extend the expressiveness of the language. In combination with the temporal properties of \( tcc \) they allow timeouts to be modeled.
by expressing what should happen if an event does not occur. The ability to model timeouts is important because most real-time systems use timeouts to detect error conditions and the ability to model this aspect of system behaviour means that the causes of these timeout conditions can be diagnosed.

Tec seems well suited to creating the models needed for consistency-based diagnosis. The parallel nature of its predicate-logic agents produces models that are declarative and expressive and the ability to easily add in new agents makes the models composable. Furthermore, the use of constraints allows these models much flexibility in the range of behaviours they represent and the incorporation of time and defaults allows temporal models to be created and timeouts diagnosed.

1.3 Objectives

The main objective of this thesis is to investigate the use of tec as the language used in creating the models employed by a consistency-based diagnosis procedure and to discover any resulting ramifications.

Although the importance of consistency-based diagnosis has grown steadily, it has been pursued mainly as an area of pure research and has not matured into an engineering discipline. This lack of success is due, in part, to the restrictiveness of current modeling techniques. Tec offers much promise toward improving the applicability of consistency-based diagnosis because it allows models to be expressed that are not possible with standard propositional logic. Tec’s use of constraints allow more flexible, qualitative models to be written that are consistent with a range of observations making them less brittle in the face of variations of correct behaviour. Also, tec’s use of default logic allows missing behaviour to be modeled. The parallel nature of tec’s agents make models inherently declarative because the structure of the model closely corresponds to the structure of the artifact. In short, tec allows logic based models to be written that are expressive, declarative, composable, flexible and temporal, and, if these attributes can be carried over to a consistency-based diagnosis procedure significant gains in the field of model-based diagnosis could be realized.

1.4 Summary of Results

This research shows that cc and tec can be successfully applied to consistency-based diagnosis. This is an important result because it directly addresses one of the biggest drawbacks of model-based
diagnosis, the difficulty in creating useful models. In conducting this research, interpreters for \textit{cc} and \textit{tcc} were written and then modified to incorporate the consistency-based diagnosis procedure, creating a generic framework for diagnosing \textit{cc} or \textit{tcc} programs or for models written in these languages.

Consistency-based diagnosis is first applied to the simpler \textit{cc} language. \textit{Cc} offers many advantages as a modeling language and these beneficial properties extend to diagnosis. Most of the standard examples commonly cited in the diagnosis literature are reproduced with models that are short, clear and declarative. In applying consistency-based diagnosis to \textit{cc}, a process of \textit{theory completion} is identified, along with its ability to provide sample input and output values for failed components; these values aid in 'explaining' the diagnoses. Furthermore, it is shown that fault models can be incorporated and that this framework can be used to perform another form of model-based diagnosis: abductive diagnosis.

This diagnostic approach is then extended to \textit{tcc}, allowing temporal models of artifacts exhibiting much more complex behaviour to be written for both simulation and diagnosis. This is important because this diagnostic approach with \textit{tcc} is the first we are aware of that is able to use a first-order temporal modeling language, whereas existing comparable approaches use propositional temporal modeling languages. Additionally, \textit{tcc}'s usefulness as a real-time control language is demonstrated by using it to control a model of a small factory that was built as a testbench and which could be modeled and diagnosed. This too is a useful result because not only does it demonstrate \textit{tcc}'s applicability in filling very different roles, it also means that faults in both the control program and in the model can be diagnosed with the same procedure. With this testbench, \textit{tcc}'s ability to model and diagnose systems that can be represented as a series of discrete states is demonstrated.

Finally, it is shown that the inequality constraints of \textit{tcc} allow flexible, qualitative models to be constructed that are able to reflect a range of correct behaviours while still conflicting with observations of faulty behaviour. Furthermore, the temporal and default logic properties of \textit{tcc} allow timeouts to be detected and their causes diagnosed. Timeouts are very commonly used to detect errors in industrial systems and, to the best of our knowledge, this is the first approach able to diagnose possible causes for these failures.
1.5 Chapter Overview

In chapter one, this chapter, an introduction to the concept of model-based diagnosis is presented. The languages $cc$ and $tcc$ are also briefly outlined and their superior modeling properties discussed. The most important part of model-based diagnosis is the model and $tcc$'s superior expressiveness, flexibility and declarative nature appear to make it ideal. Therefore, this chapter outlines the objective of using $tcc$ as the modeling language for model-based diagnosis.

Chapter two provides further background into consistency-based diagnosis and the $cc$ family of languages. It then outlines two existing implementations of diagnosis with dynamic models. These examples are used to underscore the desirable qualities of a diagnosis of dynamic systems.

Chapter three describes applying the simpler $cc$ language to diagnosis. It is found that a custom constraint system is required to model component ports and that a process of theory completion is needed to check potential diagnoses for consistency with the observations.

Chapter four extends the ideas of chapter three. First, creating temporal models and models that use defaults are discussed. Then, the physical artifact that was used as a testbase is described. The most interesting part of this testbase is that $tcc$ was used to control the system. Diagnosing faults in this system are described. Finally, creating flexible models and diagnosing timeouts is discussed.

Chapter five outlines the conclusions arrived and and outlines future recommendations.
Chapter 2

Background

This chapter provides a brief introduction to the different concepts which form the foundations of this thesis. At its heart, this research examines the combination of two very different areas of research, those of consistency-based diagnosis and concurrent constraint programming. Together they form a new approach to diagnosis that is both powerful and flexible. This combination has been refined and extended beyond static systems diagnosis to allow dynamic systems to be modeled and diagnosed.

This chapter also discusses some of the difficulties associated with diagnosing dynamic systems and examines two existing implementations. While each of these are successful demonstrations of diagnosing dynamic systems, they still have many limitations. It is these limitations that serve to illustrate what characteristics a general, flexible and powerful approach to model-based diagnosis should possess.

2.1 Consistency-Based Diagnosis

Consistency-based diagnosis (CBD) is a logical approach to diagnosing faults in systems. This technique relies on models, written in logic, to predict the behaviour of the system that is being diagnosed (also known as the artifact) and is unique in that only the correct behaviour of the artifact is modeled. If the model has been written correctly and the artifact is working properly, then the observations taken from the artifact should be consistent with what the model predicts. However, if the observations are not consistent with the model, then a fault is detected. Consistency-based
CHAPTER 2. BACKGROUND

diagnosis uses this conflict between the expected and observed behaviours to implicate one or more components of the system which could explain the fault. Each of these combinations of components is a potential diagnosis.

This approach contrasts with that of expert systems, which are currently the most widely implemented paradigm for diagnosis[11]. Expert systems are so named because they attempt to capture the knowledge of a human expert using rules of the form if Effect then Cause. When applied to diagnosis, these rules relate symptoms to failures and attempt to capture the knowledge of an experienced diagnostician who can diagnose many different faults based on the observed symptoms. An expert system uses these rules, in conjunction with the observed symptoms, to deduce possible failures. So, while expert systems attempt to capture knowledge of how a system fails and what are the associated symptoms, the consistency-based diagnosis approach models how the system behaves correctly. The consistency-based diagnosis approach has been said to have a 'deep knowledge' of the system and this permits it to do things that expert systems cannot. The biggest benefit of the consistency-based diagnosis approach is that unanticipated faults can be diagnosed. However, because consistency-based diagnosis is model-based, other important activities such as simulation and planning are possible.

As stated above, consistency-based diagnosis relies on logical models of the artifacts being diagnosed. These models are constructed as a set of interconnected components, their interconnections mirroring the structure of the artifact. The components themselves are modeled as a set of constraints which define the relationships between inputs and outputs. In the case of an observed fault, it is these constraints which create the inconsistency between the observed and expected behaviours. If, however, these behavioural constraints are lifted, then the conflict will go away. This then is the essence of consistency-based diagnosis: those components in the system model whose behavioural constraints, when lifted, restore the consistency between the model and the observed behaviour form a diagnosis for the observed fault.

Obviously, if the behavioural constraints of every component in the model are lifted, there will never be an inconsistency. Therefore, the goal of the diagnostic procedure is to find the smallest sets of components forming diagnoses, these are the minimal diagnoses.

In consistency-based diagnosis the system being modeled is generally formalized as a pair $(SD, COMPS)$ where $COMPS$ is a finite set of constants comprising the components of the system and $SD$ is the system description written in a set of first-order logic sentences which describe the components' behaviours and interconnections. The observations from the artifact are denoted by
OBS and a special (distinguished) predicate \( ab(c) \) (where \( c \in COMPS \)) is used to indicate that \( c \) is behaving abnormally while \( \neg ab(c) \) means that the component \( c \) is operating correctly. In other words \( ab(c) \) means that component \( c \) is abnormal, while \( \neg ab(c) \) indicates that component \( c \) is not abnormal (i.e. is normal). The system description is written such that each component's normal behaviour is required in the model of the system only if the component has not been labeled as abnormal, therefore SD has the form:

\[
SD = \{ B_c \vee ab(c) \mid c \in COMPS \}
\]

where \( B_c \) describes the normal behaviour of the component. A labeling of \( \neg ab(c) \) means that the normal behaviour model for the component is used, while if a component is labeled \( ab(c) \), then \((B_c \vee ab(c)) \land \neg ab(c) \) reduces to \( ab(c) \) and all behavioural constraints disappear. A diagnosis is then a labeling of all components \( c \in COMPS \) as being either abnormal \( (ab(c)) \) or normal \( (\neg ab(c)) \), such that consistency with the observations is restored.

**Definition 2.1.1 (Diagnosis)** If \( COMPS \) is partitioned into two sets \( \Delta \) and \( COMPS - \Delta \) where \( \bigwedge_{c \in \Delta} ab(c) \) and \( \bigwedge_{c \in (COMPS - \Delta)} \neg ab(c) \) then \( D\{\Delta, COMPS - \Delta\} \) is a diagnosis for \( (SD, COMPS) \) and \( OBS \) iff \( SD \cup OBS \cup \{D(\Delta, COMPS - \Delta)\} \) is consistent. Furthermore, a diagnosis is a minimal diagnosis if there is no subset \( \Delta' \) of \( \Delta \) such that \( D(\Delta', COMPS - \Delta') \) is also a diagnosis. ¹

Although the above definition determines whether a given subset of \( COMPS \) forms a diagnosis, it does not describe how to form these hypotheses. A naive approach would be to generate and then test all subsets of \( COMPS \), however, the number of such subsets is exponential in the number of components, making this approach impractical. Fortunately, more efficient techniques exist which, on average, greatly reduce the number of hypotheses which must be checked. This efficiency is gained from the observation that, when only correct behaviour is modeled, all supersets of a diagnosis are also diagnoses. Initially, all components faulty \( (\Delta = COMPS) \) is tested as a hypothesis before successively smaller subsets of \( \Delta \) are checked. Using this approach minimal diagnoses can be found, one at a time, in polynomial time [2].

Consistency-based diagnosis is not the only method to use logic to diagnose systems. A second approach, called abductive diagnosis [23], uses logic models of how the components can fail. This is similar to the expert system approach except that the cause and effect relationship in the rules is reversed. Abductive fault models have the (arguably more natural [30]) form \( \text{if Cause then Effect} \),

¹This notation is due to Reiter [25]
modeling how a fault will cause a symptom. With rules of this form, abductive diagnosis involves hypothesizing minimal faults which imply the observed symptoms.

In an abductive diagnosis, the diagnoses entail the observed faults, they are really explanations of the observed behaviour [30] as there is a causal link from the diagnosis to the observations. This is in contrast to CBD where the diagnoses are merely consistent with the observations. However, unless one has a complete model of the failure modes of the artifact, the CBD approach is preferable [17].

In reality, consistency-based diagnosis and abductive diagnosis form two extremes of a range of approaches to logic-based diagnosis. CBD techniques may include fault models and abductive techniques often include models of normal behaviour in addition to fault models. This thesis examines the addition of fault models to CBD.

2.2 Introduction to cc and tcc

In the previous section, models were said to be constructed using “behavioural constraints” but no explanation was given as to what form these constraints take and how they are evaluated. This section discusses cc and tcc, two closely related concurrent constraint programming languages used for modeling with constraints.

A constraint is simply a piece of partial information. Constraints can take forms such as ranges of values like $0 \leq X \leq 10$, relations between variables such as $V = I \times R$, or they can be combinations of these forms. Constraints are defined over different domains, such as real numbers or booleans, and these domains define what values variables can take and what are the allowed functions and relations. Constraints become interesting when they are grouped together because this allows complex problems to be described. A constraint solver is used to check if all constraints in the group are satisfiable and, if so, what value or values will satisfy them.

Concurrent constraint programming [27] (CCP) is a programming paradigm built on top of two different concepts: constraint programming and concurrent logic programming. The foundation of concurrent logic programming provides a framework whereby agents, each of which is a small logic program, can be executed concurrently; constraint programming provides the means by which agents interact. Thus, these two concepts complement one another, providing a simple but powerful mechanism for scheduling the evaluation of constraints.

Agents are the basic construct of a CCP program and they communicate through a global constraint store. This constraint store accumulates constraints as they are evaluated by agents and
also provides memory for agents much like conventional programs use variables to store specific values and alter the flow of control. However, it is important to note that the use of constraints make the CCP approach fundamentally different from that of the conventional von Neumann concept of computing. In the latter, memory is used to store specific values, whereas with CCP, memory “stores a collection of pieces of partial information about values that variables can take.” [28]

The agents in a concurrent constraint program use two basic primitives to interact with one another by way of the global constraint store: ask and tell. Tell operations are used to add constraints to the store and ask operations are used to determine if a constraint is entailed by the store. The ask operation has three possible outcomes:

- If the store contains enough information to imply the truth of the asked constraint, then the ask succeeds.

- If the store contains enough information to prove that the asked constraint is false, then the ask fails.

- However, if the store does not contain enough information to imply either the asked constraint or its negation, then the ask operation blocks. This suspends the asking agent until such time as the store contains enough information to entail either the asked constraint or its negation. This blocking action forms the basis of a synchronization primitive. Such a primitive is required in any framework involving concurrent processes.

These ideas form the basis of a family of concurrent constraint programming languages, the most basic of which is Determinate CCP, or simply cc. Derivatives of cc, such as tcc, extend its expressiveness and flexibility but all members of the family share the same declarative nature where programs are concurrently evaluated logical formulae (i.e. agents).

Computation of a cc program begins with an initial agent and an empty store. This agent may start other agents, each of which may add constraints to the store or check if constraints are entailed by the store. As constraints are added, the store evolves monotonically. That is, if the information a leads to a set of constraints, these will also be true given more information a \& b. Computation continues until the program reaches a state of quiescence where no more information is being added to the store. Figure 2.1 shows the grammar of a cc program.

The computation of a cc program can only move forward in the presence of positive information; it lacks the ability to detect missing, or negative information. To address this limitation, cc has
been extended to Default cc by including the expression of defaults. Defaults allow conclusions to be drawn in the absence of conflicting information [24]. Syntactically, this adds

\[(Agents)\quad A := \begin{cases} \{c\} & \text{(Tell)} \\ \text{if } c \text{ then } A & \text{(Ask)} \\ A, A & \text{(Parallel Composition)} \\ [X]^c A & \text{(Existential Quantification } \exists X A(X)) \\ [X]A & \text{(Universal Quantification } \forall X A(X)) \\ H & \text{(Procedure Call)} \end{cases}\]

\[(Procedure \ Call)\quad H ::= p(X_1, \ldots, X_n)\]

Figure 2.1: cc Program Syntax

Rules of this form are evaluated only after all other agents which could add positive information to the store have run to quiescence. However, evaluating A may add more constraints to the store, possibly even the antecedent c. To make this contradiction explicit we hypothesize the negation of the antecedent (¬c) and add it to the store before evaluating A [15]. Note that the use of else does not have the same meaning as if ¬c then A where the store must contain enough information to entail ¬c, while the else succeeds when the the-store does not contain enough information to entail c.

A further extension to the cc family of languages has been to add the notion of time to Default cc, creating Timed Default cc or more simply, tec [29]. Tec divides time up into a sequence of steps and at each time step a separate Default cc program is executed. Each of these iterations begins with an empty store and there is no relation between successive stores other than those which are made explicit. The representation of time in cc is accomplished with the addition of the next operator which defers an agent’s execution until the subsequent iteration. Syntactically, this adds

\[(Agents)\quad A ::= \begin{cases} \text{if } c \text{ else } A & \text{(Else)} \end{cases}\]

\[(Agents)\quad A ::= \begin{cases} \text{next } A & \text{(Next)} \end{cases}\]

This use of a series of Default cc executions is useful for modeling reactive systems and is based on the hypothesis of Perfect Synchrony [1]. Reactive systems are those which respond to external events (including clock ticks) and the hypothesis of Perfect Synchrony says that this response
CHAPTER 2. BACKGROUND

\[(Program)\quad P ::= \quad H :: A | P, P\]

\[(Agents)\quad A ::= \quad \{c\} \quad (Tell)\]

\[| \text{if } c \text{ then } A \quad (Ask)\]

\[| A, A \quad (Parallel\text{\ Composition})\]

\[| [X]^A \quad (Existential\text{\ Quantification } \exists X\ A(X))\]

\[| [X]^A \quad (Universal\text{\ Quantification } \forall X\ A(X))\]

\[| \text{if } c \text{ else } A \quad (Else)\]

\[| \text{next } A \quad (Next)\]

\[| \text{abort } A \quad (Abort\text{\ processing, call } A)\]

\[| H \quad (Procedure\ Call)\]

\[(Procedure\ Call)\quad H ::= \quad p(X_1, \ldots, X_n)\]

Figure 2.2: tcc Program Syntax

(including the detection of missing information) is instantaneous. Viewed in this light, a reactive system can be modeled as a sequence of executions, each of which begins with a set of inputs and quiesces with a set of reactive outputs. Clock ticks are just another signal and these ticks can come from more than one source, thus a multiformal notion of time is supported.

With these extensions, tcc becomes a powerful modeling language, able to model and diagnose systems which change over time and able to detect missing information. In combination with consistency-based diagnosis this forms a powerful diagnostic tool, able to model and predict the behaviour of state-based dynamic systems. Figure 2.2 shows an example of a model of a JK flip-flop written in tcc that uses defaults and models time.

\[
\text{jkff}(J,K,Q) ::= \\
\quad \text{if } \text{clpp}(Q:0) \text{ then } (\\
\quad \quad \text{if } \text{clpp}(J:0) \text{ then } \text{next } \{\text{clpp}(Q:0)\} \quad \text{else } \text{next } \{\text{clpp}(Q:1)\} \\
\quad \quad ) \\
\quad \text{else } (\\
\quad \quad \text{if } \text{clpp}(K:1) \text{ then } \text{next } \{\text{clpp}(Q:0)\} \quad \text{else } \text{next } \{\text{clpp}(Q:1)\} \\
\quad \quad ) .
\]

Figure 2.3: Definition of a JK Flip-Flop Written in tcc.

The expressiveness of tcc can be further extended by defining a series of combinators. These
always(A) :: A, next always(A).
whenever C do A :: if C then A else next (whenever C do A).
do P watching C :: if C else (P, next (do P watching C)).
do P watching C timeout B :: if C then B else (P, next (do P watching C timeout B)).

Figure 2.4: Definitions of \textit{tcc} combinators

agents, written in \textit{tcc} itself, provide a set of commonly used and powerful constructs. These combinators are:

\textbf{Unconditional Persistence:} The combinator \texttt{always(A)} will run the agent \texttt{A} in every iteration.

\textbf{Extended Wait:} The combinator \texttt{Whenever c do A} will wait until \texttt{c} is entailed before \texttt{A} is executed.

\textbf{Watchdogs:} The combinator \texttt{do P watching c} will execute agent \texttt{P} in every iteration up until (but not including) the time when \texttt{c} is entailed. The combinator \texttt{do P watching c timeout B} behaves similarly but will also run agent \texttt{B} when the store entails \texttt{c}.

Figure 2.4 shows the definitions of these combinators.

\subsection*{2.3 Diagnosis of Dynamic Systems}

In the real world, most of the artifacts that one would like to be able to diagnose are not static systems, such as combinational logic circuits, they are dynamic systems that change over time. With dynamic systems, the behaviour depends not only on the inputs, but also on when those inputs occur. This makes diagnosis of dynamic systems inherently more difficult than diagnosis of static systems.

In their early work on this subject, Hamscher and Davis [18] used digital circuits with state to show that it is rarely possible to diagnose dynamic systems using only the initial inputs and final outputs as the observations. Doing so leaves the problem too unconstrained, causing an unacceptable number of components to become possible diagnoses. The corollary of this result is that diagnosing dynamic systems requires the ability to predict intermediate values in the operation of the model and to be able to compare these with intermediate observations from the artifact. However, this is not always possible in practice as some states and state variables are not directly observable.
Therefore, the diagnosis procedure should be able to accommodate whatever temporal observations are available, using them to refine the set of diagnostic candidates.

Diagnosis of dynamic systems also requires the use of dynamic models. These models can be used to represent the continuous state-based behaviour of the artifact, or they can be discrete state-based. Discrete state-based models are generally simpler, describing finite state machines where outputs and next-state transitions both depend on the current inputs and the current state. With state-based models, the current state is represented using state variables and time is represented with the state transitions.

In modeling dynamic systems with state machines, one difficulty that arises is determining what granularity of changes in the artifact should be represented by the state transitions in the model. If this granularity is made too fine, the diagnostic procedure may become intractable. On the other hand, a representation that's too coarse may not be able to discriminate between different candidates [18]. For example, in the case of a flip-flop it may be reasonable to model state changes at the same granularity as the system clock, but in other cases it may be necessary to use a finer granularity in order to model the delays of the individual gates which make up the device. The author is unaware of a formal approach to choosing the correct granularity for state changes, in [18] an intuitive, ad hoc approach is taken. However, if modeling a periodic system where the state variable determining state changes is a clock tick, then the Nyquist frequency should be used as a guideline.

Finally, another issue that arises in dynamic systems is the temporality of faults. This is because faults may be permanent or they may be transient, manifesting themselves for only a portion of the time observed. Any temporal diagnostic procedure should be able to diagnose permanent failures as well as being able to detect, diagnose and localize transient failures.

2.4 Other Approaches

Despite many years of research in model-based diagnosis there have been relatively few practical applications of the approach. In this section two such implementations are examined. The first, developed by Darwiche and Provan [6, 5] of the Rockwell Science Center, has been successfully used to diagnose faults in a small factory system. The second example by Williams and Nayak [32], both associated with NASA, has actually been launched into space aboard NASA's Deep-Space One. This spacecraft is designed to test several new technologies including on-board diagnosis, which can
detect and recover from failures without the assistance of earthlings. This is an exciting milestone for model-based diagnosis.

While both of these approaches have been used to model and diagnose dynamic systems, they still only represent stages in the evolution of model-based diagnosis, not the final result, and their limitations highlight the characteristics that a more general approach should possess. These are the characteristics of declarativeness, where models succinctly describe what the components do, not the process by which they work, expressibility, meaning that the modeling language limitations should not restrict the desired behaviour from being described, composibility, where models can be constructed by mixing together available components, flexibility, meaning that models will not 'break' when confronted with slight variations in input values and finally multi-purposeness meaning that models can be used for other purposes besides just diagnosis.

2.4.1 The Rockwell Approach

The Rockwell approach of Darwiche and Provan is unique in that it uses the topology of the artifact to guide the construction of the system description. This “Structured System Description” or SSD, addresses a major problem with previous approaches which is that the space of possible failures grows exponentially as the complexity of the system description increases. In the Rockwell approach, the SSD is written using propositional logic sentences. However, the manner in which they are combined is constrained by a set of rules; for example, cyclic dependencies are not allowed. The major benefit of this technique is its efficiency, as the diagnosis search space grows only linearly with complexity of the system. This, in turn, allows a set of conflicts (and thus the possible diagnoses) to be generated quickly given an observation. These ideas were first derived for combinatorial systems, but have been extended to model dynamic systems as well. This has been done through the use of the next operator from temporal logic. With this addition the Rockwell approach models dynamic systems as a time sequence of static systems and allows dynamic systems, such as a manufacturing plant, to be modeled and diagnosed.

However, even though this approach to model-based diagnosis has been applied successfully, it has several restrictions which can really be viewed as limitations. An analysis of these limitations and how they constrain the diagnostic process is very useful in identifying and underlining the characteristics that a flexible and powerful, model-based diagnostic procedure should possess.

One of the main constraints imposed by the Rockwell approach is that the modeling language is restricted to propositional logic. This limitation on expressiveness restricts the models and increases
CHAPTER 2. BACKGROUND

the difficulty in accurately representing the artifact with a model. A second limitation is that SSDs represent groups of components and separate SSDs can only be combined through the use of special sensor and actuator objects. This identifies a second desirable quality of a model-based diagnosis procedure, that of composibility. Just as actual components are combined together to progressively build up the artifact, a composable modeling language should allow models to be constructed by combining components together in an arbitrary manner. A third limitation actually stems from the central idea of the Rockwell approach, this being that the system description must be expressed with a specific structure. Creating an SSD means following several rules and guidelines and ignoring these can be "devastating computationally." For example, instead of modeling a 5-input AND-gate with one node and five parents, one should model it as four cascaded 2-input AND-gates [6]. While these rules maintain the efficiency, they can mean that the model may not reflect the true structure of the artifact. A more declarative and expressive approach would maintain an accurate mapping between the model and artifact.

Finally, this approach is geared solely towards diagnosis. However, a dynamic model has other applications as well, including simulation, envisionment (hypothesizing possible futures), planning and control. A multi-purpose model-based approach to diagnosis should also support some or all of these features.

2.4.2 The NASA Approach

A second example of an application of model-based diagnosis is Williams' and Nayak's 'Livingstone', the diagnostic component of NASA's Deep Space One (DS1) spacecraft launched in October, 1998. Although it also has modest scientific goals, DS1's main purpose is to test and validate several new technologies including an ion rocket engine and an autonomous control system called Remote Agent that is based on artificial intelligence. A key component of the Remote Agent is Livingstone, which uses a single model to perform a variety of functions including: confirming hardware modes, detecting and diagnosing failures and reconfiguring hardware in order to isolate faults and recover from failures.

As with the Rockwell approach, models are written using propositional logic extended with the next operator from temporal logic. However, in this case the artifact is modeled as a transition system, using two sets of formulae. One set uses state variables to describe feasible states of the system. The other formulae describe possible transitions between the states. These possible state transitions describe both nominal transitions and failure transitions. The nominal transitions describe how the
the system evolves normally and the failure transitions describe possible failures of the components. The occurrence of a fault is then represented by the model taking a failure transition to a failure state.

This approach is used to accomplish two main goals of the system, mode identification (MI) and mode reconfiguration (MR). Mode identification is used to determine the current state of the system, given the last known state and any current observations. This is equivalent to diagnosis because all components that must have failed in order to be consistent with the current observations are identified. Mode reconfiguration is then used to recover from these failures, using the model and the current system state to plan how this recovery should be accomplished.

The NASA approach is more advanced than that of Rockwell, for several reasons. The first is that the models used are more multi-purpose, with the same model being used for a variety of purposes, such as diagnosis and planning. Another is that the models are more expressive, capturing both normal and abnormal behaviour in the form of nominal and failure transitions. The NASA models have also been extended further by associating probability and cost functions with the transitions. Probabilities are used during mode identification to help identify the most likely mode. Cost functions are used in planning to determine the least expensive route that will move the system to the desired state.

However, despite these improvements, the NASA approach still has several restrictions would not be desirable in a more flexible and powerful approach. The first is the use again of propositional logic as the modeling language. This does not allow the use of defaults and limits the expressiveness of the technique. The second restriction is that sensor inputs must first be preprocessed to translate them into qualitative summaries (such as high, low or nominal) before they can be used by the model. A more flexible approach would allow the raw sensor values to be used directly which still being able to use the data in a qualitative manner.

2.5 Summary

These two examples (especially the NASA approach) are good examples of the state of the art in model-based diagnosis of dynamic systems and they demonstrate that such an approach is indeed practical. However, these approaches still have some limitations which underscore the need for a diagnostic method that is declarative, expressive, composable, flexible and multi-purpose.

A consistency-based diagnosis procedure using tcc as the modeling language has the potential to
fulfill these requirements. Tcc programs are made up of concurrently executing agents. Each of these agents is a small logic program, able to declaratively model a subcomponent of the artifact. These agents allow models to be written in default predicate logic, greatly improving their expressiveness over simple propositional logic. The agents then run concurrently with the constraints between them determining how they will interact. Agents are self-contained entities and can be mixed in with pre-existing agents, creating an approach where the overall behaviour is a composition of all of the interacting agents. Finally, models created using tcc can also be used for simulation, planning and control. Their accurate representation of the underlying structure of the artifact allows models to be used for several purposes.
Chapter 3

Modeling, Fault Detection and Diagnosis with cc

While the previous chapter introduced consistency-based diagnosis and the basic cc programming language separately, this chapter discusses their combination. Consistency-based diagnosis requires models of the systems being analyzed and, as a modeling language, cc has many beneficial properties.

This chapter first discusses modeling aspects of cc including using the underlying constraint systems to create component models. Modifying the operation of cc to incorporate consistency-based diagnosis is then discussed including the difficulties encountered when component constraints are lifted. Finally, the addition of fault models to the process is examined.

3.1 Modeling with cc

A real-world system, such as an automobile engine, consists of a set of interconnected components all working together whose individual behaviours are governed by both their natural function and their interactions with other components. For example, a gear’s natural function is to spin, but when integrated into a larger system this behaviour is constrained by the other gears it meshes with. Similarly, a cc program consists of a set of concurrently executing agents whose behaviour is dictated by their prescribed function (each is a logical formula), and by their interactions with other agents.

This natural correspondence between real-world systems and cc programs makes cc highly applicable as a modeling language because agents naturally model system components. When using
CHAPTER 3. MODELING, FAULT DETECTION AND DIAGNOSIS WITH CC

cc for modeling, an agent’s name denotes the component it models and the agent’s arguments correspond to the component’s input and output port values. The body of each agent then contains the rules describing the corresponding component’s behaviour. Agents are denoted by \( p(\bar{X}) \rightarrow B \) where \( p(\bar{X}) \) is the agent’s name and parameters and \( B \) is its body.

A cc language is defined over top of one or more constraint systems which implement the rules for evaluating different types of constraints. Three common constraint systems are \( \text{clp}(\mathbb{R}) \), \( \text{clp}(\mathbb{FD}) \) and \( \text{clp}(\mathbb{B}) \) which define constraints over real numbers, integers with finite domains and booleans, respectively. Constraint systems can also be custom defined for less general domains. One such system is \( \text{clp}(\mathcal{P}) \) which has been designed especially for modeling with cc. \( \text{Clp}(\mathcal{P}) \) defines a set behaviours common to component ports, the terminals that are used to connect components together and which are used to pass data by way of variables. \( \text{Clp}(\mathcal{P}) \) restricts component ports that are connected together to have the same value, just as a wire that connects two electrical components has one and only one voltage.

A \( \text{clp}(\mathcal{P}) \) constraint has the form \( (\sigma : V) \) where \( \sigma \) is an atom that identifies the port and \( V \) is the port’s value. \( \text{Clp}(\mathcal{P}) \) constraints are defined by the following two axioms:

\[
\forall P \exists V \, \text{clpp}(P : V)
\]

\[
\forall N \forall X \forall Y ((\text{clpp}(N : X), \text{clpp}(N : Y)) \rightarrow (X = Y))
\]

which state that every port has some value and if any two ports have the same name (i.e. are connected), then they will have the same value.

Through \emph{asks} and \emph{tells}, agents communicate values via \( \text{clp}(\mathcal{P}) \) in a producer/consumer fashion. A consumer agent will block on an \emph{ask} until a producer agent \emph{tells} the corresponding \( \text{clp}(\mathcal{P}) \) constraint to the store. A \( \text{clp}(\mathcal{P}) \) \emph{tell} has the syntax \( \{\text{clpp}(P : V)\} \) where \( P \) is an existential variable representing the port name and \( V \), the port’s value, is a variable or a ground term. When this constraint is added to the store then \( P \) is replaced by a Skolem constant. A \( \text{clp}(\mathcal{P}) \) \emph{ask} has the syntax \( \text{if clpp}(P : V) \text{ then } A \) and this creates several possible situations of free and uninstantiated variables. Table 3.1 outlines the rules for possible interactions.

In a cc program, \( \text{clp}(\mathcal{P}) \) can be used to mix constraint systems by associating a constraint variable from another constraint system with a port name. Figure 3.2 gives the cc model of the and-or gate shown in figure 3.1. In this example \( \text{clp}(\mathcal{B}) \) is used to model the logic of the gates and

---

1A Skolem constant is a ground term used to represent an instance of an existential variable.
### Table 3.1: $Clp(P)$ Entailment Rules.

$clp(P)$ is used to associate these boolean variables with the ports of the components. The $clp(B)$ constraints define the relationship between input and output variables and the $clp(P)$ constraints relate these variables to the gate's ports, ensuring that gates which share ports (i.e. variable $Q$) hold only one value.

![Figure 3.1: AND-OR Circuit.](image)

```
and(A, B, Q) :: [Va, Vb, Vq]$((
    if (clpp(A:Va),clpp(B:Vb))
    then ((clpb(Vq ::= Va * Vb)), [clpp(Q:Vq)]))
).

or(A, B, Q) :: [Va, Vb, Vq]$((
    if (clpp(A:Va),clpp(B:Vb))
    then ((clpb(Vq ::= Va + Vb)), [clpp(Q:Vq)]))
).

and-or(A, B, C, Z) :: [Q]^((
    and(A, B, Q), or(Q, C, Z)
).
```

![Figure 3.2: $Cc$ Program for the AND-OR Circuit.](image)
3.2 Diagnosis of cc Programs

In the previous section it was shown that there is a one-to-one correspondence between agents in a cc program $P$ and the components in a modeled system, with agent names representing the components. Recalling from the previous chapter that the components of a system are labeled COMPS, we label these agent names which represent the components COMPS($P$). Also, in the previous chapter a system description was defined to be a set of constraints that describe each component's correct behaviour. In other words, for each component $p$'s behaviour $B_p$, the system description contained a clause $\neg ab(p) \rightarrow B_p$. Therefore, a system description defined by a cc program $P$ is given by:

$$SD(P) = \{(p(\bar{X}) \rightarrow (B_p \lor ab(p))) | (p(\bar{X}) \rightarrow B_p) \in P\} \cup CS,$$

where $CS$ represents the underlying constraint system. Given these definitions, we can rewrite the definition of consistency-based diagnosis from the previous chapter in terms of diagnosing cc programs.

**Definition 3.2.1 (Diagnosis of a cc Program)** Given a cc program $P$ describing a system $(SD(P), COMPS(P))$, a set of assertions OBS and $\Delta \subseteq COMPS(P)$ containing only components assumed abnormal, then $D(\Delta, COMPS(P) - \Delta)$ is a consistency-based diagnosis for $(SD(P), COMPS(P))$ and a set of assertions OBS iff $SD(P) \cup OBS \cup D(\Delta, COMPS(P) - \Delta)$ is consistent.

This diagnosis $D(\Delta, COMPS(P) - \Delta)$ is a partitioning of COMPS($P$) into two sets: $\Delta$, which consists of agents representing those components assumed abnormal, and COMPS($P$) $- \Delta$, the remaining agents representing the normally behaving components. Given the definition of $SD(P)$, members of $\Delta$ have definitions of the form $((p(\bar{X}) \rightarrow (B_p \lor ab(p))) \land ab(p))$, or more simply $ab(p)$, and members of COMPS($P$) $- \Delta$ have definitions of the form $((p(\bar{X}) \rightarrow B_p) \land \neg ab(p))$. Since the predicate $ab(\cdot)$ does not appear in a cc program, checking a hypothesis $\Delta$ for consistency reduces to removing the agents in $\Delta$ from $P$ and testing to see of the remaining agents run to quiescence. In other words, the (possibly conflicting) constraints imposed by the agents in $\Delta$ are lifted by removing the agents themselves from the program $P$. This new program, $P_\Delta$, is then checked for consistency with $OBS$.

While logically sound, this removal of an agent or agents creates a problem for the consistency checking procedure because it effectively creates a 'hole' in the system description. This gap can prevent values from being propagated forwards and backwards along the remaining constraints.

Consider the circuit in figure 3.1 along with the observation $A = 1, B = 1, C = 0, and Z = 0$. 
In checking the diagnosis $D(\{A1\}, \{O1\})$ all constraints on the behaviour of $A1$ are removed, as shown in figure 3.3. In removing these constraints, the relationships between $A$, $B$ and $Q$ become undefined and this presents a difficulty in determining a value for $Q$. On the other hand, $Q$ is no longer constrained by $A1$ and can take on any value that is consistent with $O1$. Therefore, a process of \textit{theory completion} is required in order to choose a value for $Q$ that will produce a consistency when the remaining agents are run.

Theory completion uses the fact that agents in a \textit{cc} program are written as 'information transducers' in the form of \textit{if c then A'}. These information transducers extract data from the store by \textit{asking} the constraint $c$ and then processing this information in agent $A$. Usually $A$ will also \textit{tell} any results back to the store. In the case above, the agent for $A1$ has been removed from the program which means that there is no agent to \textit{tell} a value for $Q$. This causes the agent for $O1$ to block indefinitely while trying to extract the value of $Q$ with an \textit{ask}.

Theory completion simply involves \textit{telling} $c$, the antecedent of the blocked agent. In the case above, the antecedent $c_{1pp}(q:Va)$ from the \textit{or} agent would be told directly, this is acceptable because $Q$ can have any value due to the fact that $A1$'s behaviour is undefined. In general, it is only possible to use theory completion to \textit{tell} antecedents which contain \textit{unrestrained} variables. \textit{Unrestrained} variables are those port variables of agents that belong to $\Delta$. Those antecedents of agents which contain \textit{unrestrained} variables are \textit{unrestrained antecedents}. Theory completion is the act of \textit{telling} different combinations of \textit{unrestrained} antecedents in order to create a consistency.

This process of theory completion is very similar to abduction where, given the rule $A \rightarrow B$ and the fact $B$, then $A$ is hypothesized, provided it does not conflict with any other fact. However, with theory completion $A$ is hypothesized without any prior knowledge of the fact $B$. If this creates an inconsistency then $A$ is retracted, otherwise both $A$ and $B$ are asserted. This is acceptable since with diagnosis we are only checking for consistency.

In many cases, there are multiple agents all blocked on the same antecedent $c$ and often times their execution depends on mutually exclusive values for $c$. In other words, when a value for $c$ is told to the store, all of the blocked agents will unblock but only one will continue, the rest will disappear.
because the constraints in their antecedents have not been satisfied. Therefore, it is not sufficient

to simply tell the antecedent of the first agent blocked on c. Instead, all of the antecedents of all

blocked agents must be hypothesized in turn until a consistency is found or all possibilities have

been exhausted.

Besides allowing agents to run to quiescence in order to check for consistency, the theory com­
pletion process has a beneficial side effect, it creates explanations for the diagnoses. An explanation

is an instance of ground values for the unrestrained variables which creates a consistency between

PA and OBS. For example, in figure 3.3 values for A, B, C and Z are given by the OBS, theory

completion then tells an uninstantiated value for Q to the store and this lets the agent for the OR
gate proceed, adding a binary constraint relating C, Q and Z to the store. From these values and

the relational constraint, a value for Q can be established creating an explanation for the diagnosis

D({A1}, {O1}), namely that A = 1, B = 1, and Q = 0.

Finally, in theory completion, it is important to delay hypothesizing unrestrained antecedents

until all other agents have been evaluated. Consider the circuit in figure 3.4. If, in this example,

a2's constraints are lifted (i.e. ∆ = {A2}), then Q is a unrestrained variable and A3's antecedent

is a unrestrained antecedent. If A3 is then evaluated first many different values for Q may be

hypothesized until one is found which is consistent with A1. However, if A1 is evaluated first, then

Q's value can be established right away and consistency can quickly be checked. In more complex

systems, the theory completion process may flounder unless remaining agents are first allowed to

establish values for as many unrestrained variables as possible before any values are hypothesized.

### 3.3 Integrating Fault Models

One of the criticisms of consistency-based diagnosis has been that, while it produces diagnoses that

are consistent with the observations, many times these diagnoses are highly improbable or physically
impossible. Consider the three lightbulb circuit in figure 3.5, originally presented in [31], along with the observation that bulb $B3$ is lit, but the other two bulbs, $B1$ and $B2$ are not lit. While the most obvious diagnosis is $\Delta = \{B1, B2\}$, the consistency-based diagnosis procedure also generates other diagnoses like $\Delta = \{S, B3\}$. The explanation for this second diagnosis is that the battery $S$ is faulty because it not producing a voltage and bulb $B3$ is faulty because it is lit in the absence of a voltage.

While the second diagnosis is extremely unlikely, the diagnostic procedure gives all diagnoses equal weight because it has no knowledge of how components commonly fail. However, for most types of components there are common fault modalities and it is often desirable to include this additional information about component behaviours into the system description. These fault models describe the way components fail and are often assigned probabilities so that the likelihood of failure can be determined for single and multiple faults. One difficulty with fault models, however, is that if all fault modes have not been accounted for, then the diagnostic procedure can fail to find a diagnosis. To deal with this problem, an additional ‘unknown fault’ model is often introduced [10] which imposes no behaviour constraints and so always succeeds.

Diagnosis with cc is able to incorporate fault models in addition to implementing a purely consistency-based diagnosis procedure, permitting the most likely diagnoses to be determined. This section discussed three methods by which this can be accomplished. The first uses a special variable to determine the state of each component in the system. The second method uses consistency-based diagnosis to refine this technique while the third method replaces the normal behaviour with fault behaviour in the component models.
The first approach requires that the system description be written in a unique way, such that each component's description includes a *mode* variable. For each component, the value of this variable determines which set of constraints will govern the component's behaviour. If *mode* = *ok*, then the nominal set of constraints will be used to give the component its normal behaviour. Otherwise, if *mode* indicates a fault, then the set of constraints which provide the corresponding fault model's behaviour will be used instead. Figure 3.6 gives the *cc* agent for a battery component where, if *Mode* = *ok*, then the rule which provides normal values for the terminals will be used (energized = 1) but if *Mode* = *dead*, then the rules which model the behaviour of a dead battery will be used.

```
battery(Mode,Left,Right) :-
  if (Mode=ok) then {Left=1,Right=1},
  if (Mode=dead) then {Left=0,Right=0}.
```

Figure 3.6: Agent for the Battery Component.

With this approach, a diagnosis is found by making the mode variable for each component a *unrestrained* variable. Doing so will cause the diagnosis procedure to use *theory completion* to hypothesize different values for the components' modes until a set of modes are found where the behaviour resulting from the model is consistent with the observations. When a consistency is found, those components whose modes have been hypothesized as abnormal will form a diagnosis.

If there is no 'unknown fault' mode, then it is assumed that all fault modes are completely characterized and any diagnoses will actually be abductive diagnoses because the hypothesized modes will then entail the observations. Using this approach, the results of Codognet and Saraswat[4], who used *cc* to perform abductive diagnosis on the three lightbulb example (see figure 3.5), have been reproduced.

However, all combinations of component modes must be hypothesized and their entailment relations checked until a consistency is found. This make this first technique of using theory completion to search for combinations of component modes to explain the observations very computationally expensive as the number of possibilities to check is exponential in the number of components and fault modes. In order to improve this efficiency, this technique can be refined by incorporating
CHAPTER 3. MODELING, FAULT DETECTION AND DIAGNOSIS WITH CC

consistency-based diagnosis.

A second, more efficient approach works in two steps. First, consistency-based diagnosis is used to generate candidate diagnoses and then, in a second step, theory completion is performed on the mode variables of the components in these diagnoses in order to discriminate among them. This approach is more efficient because consistency is easier to determine than entailment and the number of fault modes is not a factor in determining the initial candidate diagnoses. This approach can be used without making any changes to the cc diagnosis procedure. In the first step, consistency-based diagnosis is used to generate candidate diagnoses by fixing the value of the mode variable for each component at ok so that only normal behaviour is used. Then, in the second discrimination step, the different failure modes for the components in \( \Delta \) are hypothesized while all other modes remain normal.

The main benefit of these first two approaches is that they can be accomplished without any change to the cc diagnosis procedure. Unfortunately, the use of a mode variable goes against the spirit of consistency-based diagnosis which uses declarative models of component behaviour. A third approach to integrating fault models requires a slight modification to the cc diagnosis procedure, but allows the use of a normal system description, modeling only normal behaviour. With this approach fault models for the components are stored separately and the normal consistency-based diagnosis procedure is used to generate candidate diagnoses. Then, in a second step, the normal behavioural constraints of the components in the diagnoses are replaced with the fault model constraints, one at a time, until consistency between the fault modes and the observations is found.

3.4 Summary

The most important facet of model-based diagnosis is actually developing the models of the artifacts to be diagnosed. \( \text{Cc} \) makes an excellent candidate for a diagnostic modeling language, in part because its concurrently executing, logic-based 'agents' correspond well to an artifact's subcomponents. Also, because \( \text{cc} \) works in conjunction with underlying constraint systems, the power of the language can easily be extended by adding new constraint systems. One such custom constraint system is \( \text{clp}(P) \), which has been added to model component ports.

In order to implement diagnosis with \( \text{cc} \), the interpreter which executes the \( \text{cc} \) programs (artifact models) has been modified to incorporate the consistency-based diagnosis procedure. This is done by removing those agents in a candidate diagnosis from the artifact model and then checking to see
if the remaining agents will run to completion, thereby determining their consistency. In order to allow this to happen, however, a process of theory completion is required which has the side benefit of providing explanations of the diagnoses in the form of example port values for failed components.

Finally, pure consistency-based diagnosis often produces some diagnoses that are useless because they are impossible or extremely improbable. In order to produce more realistic diagnoses, the cc diagnosis procedure can be extended to incorporate fault models. Three possible ways of accomplishing this have been discussed.
Chapter 4

Modeling, Fault Detection and Diagnosis with \textit{tcc}

In the previous chapter, modeling with \textit{cc} was first discussed before diagnosis of \textit{cc} programs was introduced (from a mainly theoretical standpoint) and resulting issues were explored. This chapter follows the same general approach with \textit{tcc}, but also goes into more detail about the practical aspects of \textit{tcc} diagnosis. This is because a physical model of a dynamic system was constructed to evaluate the new ideas presented here. This testbench, a model of a factory built from LEGO \footnote{LEGO is a trademark of the LEGO Group (INTERLEGO AG in the USA).}, is an example of a small, real-time system, and in order to control it, a real-time language was required. As it turned out, it was possible to use \textit{tcc} as this real-time control language. This revealed a new aspect to \textit{tcc}, in addition to its uses for modeling and diagnosis. In the end, all of these aspects come together, as the same techniques that are used for diagnosing the model can also be used to diagnose the control program.

In the first section of this chapter using \textit{tcc} to create dynamic models is explored. \textit{Tcc} is much more expressive than \textit{cc}, both because of its ability to represent time and because of its ability to define default behaviour, and this means that \textit{tcc} models can exhibit complex behaviour. In the second section, a different tack is taken and using \textit{tcc} as a real-time control language is discussed. Within this context, the testbench that was constructed and is controlled by \textit{tcc}, is introduced. The third section then discusses diagnosis of \textit{tcc} programs. After a theoretical introduction, the practical aspects of diagnosing faults in the LEGO factory are discussed. This includes the \textit{tcc} model of the
artifact, the \textit{tcc} interpreter used to execute the model and the steps required to perform diagnosis. Finally, different modeling techniques and diagnosing faults due to timeouts are discussed before a summary of these ideas is given.

4.1 Modeling in \textit{tcc}

As has already been mentioned, \textit{tcc} builds on \textit{cc} because \textit{tcc} programs run as a series of \textit{cc} program executions, or iterations. Although each of these iterations is assumed to take zero processing time to complete, it is also assumed that a finite amount of time passes between iterations. In other words, separate \textit{cc} iterations are temporally distinct and the order in which they occur determines their order in time. This builds the concept of time right into the language because events that occur in subsequent iterations are assumed to happen after events from previous iterations; this allows temporal models to be expressed. Syntactically, this adds the \texttt{next} operator to \textit{cc}, which defers an agent's evaluation until the subsequent iteration. Models written in \textit{tcc} use the \texttt{next} operator to model temporal behaviour by scheduling certain actions to occur after others.

A second improvement over \textit{cc} is \textit{tcc}'s ability to express defaults. Defaults are due to Reiter [24] and allow inferences to be made from missing information. For example, the statement \texttt{if C else A.} allows the conclusion \texttt{A} to be drawn if \texttt{C} is not detected (not found to be true). Defaults greatly extend the expressiveness of \textit{cc} because they allow 'catch-all' statements that define behaviour without having to enumerate every situation in which it should occur. Defaults add the \texttt{else} statement to the syntax of \textit{cc}.

These new ideas can be demonstrated with a \textit{tcc} model of an inverter. \textit{Cc} is only able to model an idealized inverter because it can only represent the output occurring simultaneously with the input. However, there will always be some delay between an actual logic gate's input and output and \textit{tcc} can be used to write models that capture this reality. Figure 4.1(a) shows how this is done by adding a delay element, \(\delta\), to the output of the idealized inverter causing the result to appear after the input is received. Figures 4.1(b) and 4.1(c) are two different \textit{tcc} models for the inverter that both use the \texttt{next} operator to defer the assertion of the output constraint until the iteration after the input is received.

Although they both reproduce the behaviour of the inverter, the models in figures 4.1(b) and 4.1(c) show two alternate approaches to modeling in \textit{tcc}. Figure 4.1(b) uses \texttt{clp(P)} to extract the input value from the store and then uses the boolean constraint system \texttt{clp(B)} to determine the
negated value of the input. \( Clp(P) \) is then used once more to assign a new value for the output port in the next iteration. Figure 4.1(c) uses defaults and ground values for port values to achieve the same effect. In this case, only an input value of 0 can be detected. If the input is 0, then the output will be 1. If the input is not 0, then the default action is taken and the output value will be 0. This demonstrates how defaults can be used to detect the presence or absence of information (in this case the presence of absence of an input value of 0). In later examples, defaults will be used to detect if certain actions have not occurred, allowing timeouts to be detected and diagnosed. This second approach can be also executed more efficiently because it does not require the use of intermediate variables. In both forms of the model, however, \( clp(P) \) constraints are used to relate values to ports. This pattern is consistent with all of the component models used in this thesis.

This use of temporal models for the logic gates can be used to reveal temporal faults such as glitches when gates are used to build up larger circuits. For example, temporal models of AND, OR and XOR logic gates were used to construct a model of the full-adder\(^2\) shown in figure 4.2. This model was then executed and used to construct the execution trace of table 4.1 which shows how the

\[^2\text{Unit delay assumed for all gates.}\]
input values propagate through the circuit. The table gives the values for the three inputs and two outputs of the full-adder as well as for the three intermediate variables $p$, $q$ and $r$, all over eight time steps (dashes are indeterminate values). Executing this temporal model reveals a glitch because when the input $X$ changes from a 1 to a 0 in iteration 3, it causes the output to unexpectedly change to 0 in iteration 5 before stabilizing again at 1 in iteration 6. This unexpected behaviour would not be revealed by the purely combinational models of $cc$.

![Figure 4.2: Full Adder with Gate Delays](image)

**Table 4.1: Full-Adder Execution Trace**

<table>
<thead>
<tr>
<th>Iteration</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$Y$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Carry In</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$p$</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$q$</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>$r$</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Carry Out</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Output</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Even though $tcc$ can be used to create temporal models, it cannot model systems that vary continuously with time. Instead, $tcc$ models behaviour as occurring in discrete time steps, although continuous behaviour can be approximated by making these timesteps arbitrarily short. In other words, $tcc$ is a language for describing state machines and executing a $tcc$ program is equivalent to navigating a path through the state diagram.

In tracing a path through the state diagram, each $cc$ iteration determines what the present state is and what the next state will be. The current state is maintained until the start of the next
iteration when an instantaneous transition occurs and the previous next state becomes the current state. This relationship is shown in figure 4.3. *State variables* determine what the present state is and, in the context of *tcc*, these are variables that span iterations through the use of the `next` operator.

These ideas can be demonstrated by using *tcc* to model an actual state machine implemented in digital logic. Figure 4.4(a) shows the state diagram where the different arcs between states correspond to different input values. Figure 4.4(b) shows the realization of this state diagram using JK flip-flops. This circuit creates a state machine because the outputs are fed back into the inputs and the flip-flops are clocked. The clocking prevents the inputs from immediately affecting the outputs, creating a series of discrete states.

The *tcc* model of this circuit is shown in figure 4.5. In the *tcc* model, the clock is inherent in the execution of the program with one clock tick per iteration. In this model the current state is determined by reading in the values for the two state variables, Q0 and Q1. Then, after reading in these values, the next state is determined from the current state and the input I, this is then told with a `next`. The outputs are fed back into the inputs by assigning the next values for the state variables Q0 and Q1 at the end of an iteration and then reading these values in again at the start of the subsequent iteration. The initial state of the system is set with an asynchronous clear which immediately clears both flip-flops.

This state machine can now be used for other purposes besides simply modeling. Of course the main objective is to use it for diagnosis, but planning is also possible. In planning, an initial state and a desired final state are known ahead of time and the task is to determine what inputs will take the system from the initial state to the desired final state. This can be accomplished by making the
input variable \( I \) a unrestrained variable. The \( tcc \) interpreter will then use theory completion to find a sequence of \( I \)s that produce a path through the state machine that is consistent with the desired states.

![State Diagram and Digital Circuit](image)

Figure 4.4: State Diagram and Digital Circuit

### 4.2 \( tcc \) for Real-Time Control

In this section, using \( tcc \) as a real-time control language is discussed. The first part of this section looks at how this has been accomplished and the second part introduces the artifact that was built and which \( tcc \) was used to control.

#### 4.2.1 \( Tcc \) as a Control Language

Although \( tcc \) has been presented so far as a modeling language, it is also highly applicable to controlling real-time systems. This is because \( tcc \) exhibits all of the characteristics of a class of languages known as synchronous reactive languages. The properties of these languages, which include ESTEREL [1, 21], SIGNAL [14] and LUSTRE [16] are first discussed in this section before it is shown how a slight modification to \( tcc \) allows it to be used for writing real-time control programs.

Synchronous reactive languages have been developed for controlling reactive, real-time systems. A reactive system, as its name implies, reacts to inputs from the external environment, generating outputs in response. After responding to those inputs it quiesces and waits for the next inputs to occur. This produces an environment where computation occurs in bursts of activity followed by periods of inactivity. The computational model for this reactive approach to control is based upon
CHAPTER 4. MODELING, FAULT DETECTION AND DIAGNOSIS WITH TCC

not(1,0) :: (  
  if clpp(I:0) then {clpp(0:1)}  
  else {clpp(0:0)}  
).

or(A,B,Q) :: (  
  if clpp(A:1)  
    then {clpp(Q:1)}  
  else ( if clpp(B:1)  
               then {clpp(Q:1)}  
               else {clpp(Q:0)}  
     )  
).

% J and K are the two inputs,  
% C is asynchronous clear, Q is output  
jkff(J,K,C,Q) ::  
  if clpp(C:1)  
    then ( {clpp(Q:0)}, next {clpp(Q:0)}  
  else( if clpp(Q:0)  
   then ( if clpp(J:0) then (next {clpp(Q:0)})  
         else (next (clpp(Q:1)))  
   else ( if clpp(K:1) then (next (clpp(Q:0)))  
                else (next (clpp(Q:1)))  
     )  
).

% definitions for the system components  
inv(I,0) :: not(I,0).  
orl(A,B,Q) :: or(A,B,Q).  
or2(A,B,Q) :: or(A,B,Q).  
ff0(J,K,C,Q) :: jkff(J,K,C,Q).  
ff1(J,K,C,Q) :: jkff(J,K,C,Q) .

abvars(_).

%definition for the system  
state_machine(I,C,Q0,Q1) :: [Ibar,J0,J1,K1]^('  
  orl(I,Q1,J0), invl(I,Ibar), or2(Q0,Ibar,K1),  
  ff0(J0,I,C,Q0), ff1(J1,K1,C,Q1),  
  {clpp(J1:1)})  
).

Figure 4.5: Tcc Model of the State Machine
the hypothesis of Perfect Synchrony [28] which assumes that the system's reaction to a set of inputs (including the detection of missing information) is instantaneous. Also, with synchronous reactive languages only deterministic behaviours are allowed to be specified, meaning that the same set of inputs will always produce the same outputs. This gives the language a precisely defined semantics and allows for formal verification of programs.

In effect, synchronous reactive languages describe deterministic finite state machines. In these state machines each computation phase is a state change triggered by the inputs, with the output values being determined by those inputs and the current state. Computation quiesces when a new state is reached with the transitions between states assumed to be instantaneous. In reality, there will always be some delay between the inputs being received and the outputs being produced but the hypothesis remains valid so long as the environment does not change while reactive outputs are being calculated [21]. The minimum time over which the controlled system can be guaranteed to remain constant creates an upper bound on the computation time of a reaction. Synchronous reactive languages offer an attractive approach to describing finite state machines because they describe their behaviour, not their structure. This is useful because small changes in the logic of a program can drastically change the structure of a finite state machine that implements it, making state machines very inconvenient to program directly [21].

One characteristic that all but the simplest of real-time control systems must possess is the support of concurrent processes. This is because complex real-world systems contain many elements all working in parallel and, most often, these separate elements need to be controlled simultaneously. With a conventional real-time system running on a single processor this means that these concurrent processes must share the processor through multitasking. Usually, this creates a high degree of nondeterminism with respect to how these processes are interleaved and this can lead to faults that are spurious and difficult to diagnose. Synchronous reactive languages, on the other hand, support the notion of logical concurrency where the different tasks that control the separate components of the system are written as concurrent processes. However, although the control structures are written as parallel functions, together they form a single, deterministic finite state machine removing any nondeterminism. Furthermore, the safety of these logical processes and maximum reaction times can be analyzed apriori.

However, the most important feature of these languages is their inherent assumption that computation occurs in phases of activity followed by periods of quiescence that are nonzero in length. This computational model builds the concept of time into the language structure because
events that occur in separate computational cycles are assumed to be separated in time. This means that an event can be programmed to occur in the future simply by scheduling it for an upcoming cycle. As stated above, a computation cycle occurs as a reaction to an external input and if this input is the tick of a clock then computation cycles can be made to occur at regular intervals. This makes it possible to precisely schedule when this future event will occur. This is known as a "control-based" paradigm [33] distinguishing it from the more common "data-oriented" style usually found in real-time programs. In the data-oriented style time is represented using a variable that stores the value of a timer, whereas with the control-based approach the passage of time is built right into the structure of the program. The control-based approach simplifies timing analysis of the program because it can be done statically. With the data-oriented approach, timing analysis is much more difficult because the passage of time can only be viewed by executing the program and tracing its behaviour.

Synchronous reactive languages interact with the real world through the use of signals and drivers. Signals are abstract data entities which require drivers to be transformed into physical signals for outputs or to be generated from physical signals in the case of inputs. Signals are also used to communicate among concurrent processes during an execution cycle. When interfacing to the real world, it is important that any external action caused by a signal have a bounded response time in order to remain consistent with the synchrony hypothesis. In this framework clock ticks are just another signal and multiple clocks producing multiple signals are possible, therefore a multiform notion of time is supported.

*Tcc* shares an almost identical structure with these synchronous reactive languages. It too executes in alternating phases of activity and quiescence and supports the notion of logical concurrency. The only modification required to be made to *tcc* in order to allow it to be used for real-time control is the addition of signals and a driver. In order to add signals to *tcc*, a new custom constraint system, *clp(S)*, was created. *Clp(S)* works very much like *clp(P)*, where constraints interact with the store through *asks* and *tells*, except that non-ground values are not allowed to be told. Also, *clp(S)* communicates with the outside world through the use of a customized driver. When *clp(S)* signals are told to the store, they are intercepted by the driver which performs the desired real-world action. The driver also translates external events into *clp(S)* signals that are automatically added to the store, possibly triggering other actions to occur. This behaviour is shown in figure 4.6. Even with the addition of *clp(S)*, there is still one difference between *tcc* and other synchronous reactive languages: external signals cannot cause *tcc* computation cycles. However, this difference is due to
CHAPTER 4. MODELING, FAULT DETECTION AND DIAGNOSIS WITH TCC

implementation decisions, not fundamental differences between the languages, and is dealt with by allowing the tcc computations cycles to run freely and continuously. This creates a situation identical to the cycles being triggered by an external clock with a frequency higher than but synchronized with all other external clocks.

Clp(S) signals have the form clps(S:V) where S is the name of the signal and V is a legal value for that signal. The names and values for clp(S) signals are application dependent and relate to the system being controlled. For example, a system with a conveyor belt might have a signal name of conveyor and possible conveyor values of on, off, forward and reverse. Turning the conveyor on is then simply a matter of telling the constraint \{clps(conveyor:on)\}.

As explained in chapter 2, tcc combinators are constructs, written in tcc itself, that extend the expressiveness of the language. The parallel nature of tcc and the expressiveness of these combinators combine to create a powerful, declarative control language. Two combinators that are especially useful for real-time control are the delay and the watchdog combinators. The delay combinator \texttt{next}(Num, Signal) do A will execute agent A after there have been Num occurrences of the signal Signal. The watchdog combinator \texttt{watchdog}(Count, Trigger, WatchedFor, Recover) will abort the current execution path and replace it with the exception handler agent Recover if the constraint WatchedFor is not detected within Count occurrences of Trigger. Figure 2.4 showing

---

Figure 4.6: Interaction Between clp(S), tcc and the Driver
the combinator definitions is reproduced in figure 4.11 with the additional definitions of these new combinators.

As an example of using the delay combinator, figure 4.7 shows a motor-controlled pneumatic valve that is used to control the movements of an pneumatic piston. The motor moves the valve lever up or down to direct the airflow into outlets A or B. Figure 4.8 shows an agent that will cause the lever to move from position A to B, using a clp(S) constraint to turn the valve motor on. The delay combinator is then used to ensure the motor is turned off again five clock ticks later.

![Figure 4.7: Motor-Controlled Pneumatic Valve](image)

valve1_down :: \{clps(valve1: on)\),
next(5, clps(tick: on)) do \{clps(valve1: off)\}.

![Figure 4.8: Tcc Agent For Switching Pneumatic Valve Position](image)

The watchdog agent is used for implementing timeouts. A timeout is the detection of an error due to an expected event not occurring. This detection of missing information is accomplished using tcc’s else operator, which implements default logic. For example, the conveyor belt shown in figure 4.9 moves the token to left, using the three sensors to detect it passing underneath. The conveyor is then supposed to turn itself off a short time after the token has passed underneath the last sensor. If however, due to some failure, the last sensor never detects the token, the conveyor will never shut off. This potential problem is handled with a watchdog which knows that the token should be detected within a set time of it being placed on the conveyor. If the token is not detected, the watchdog shuts down the conveyor and reports an error.
Use of the watchdog combinator is shown in figure 4.10. The agent `watched_conveyor` simultaneously launches the `conveyor` agent, which runs the conveyor belt, and the `watchdog` combinator which will invoke an exception handler if a timeout occurs. The `conveyor` agent turns on the conveyor and waits until the token is detected under `conveyor_sensor3`. When the token moves under the sensor, the sensor will have the value `on`. The agent then waits for five more clock cycles before turning the conveyor off in order to make sure that the token has had enough time to travel past the end of the belt. If, however, there is a problem and the token is never sensed by `conveyor_sensor3`, then the conveyor belt will never shut off. This possibility is handled by the `watchdog` agent which will execute the exception handler agent `halt` if the token is not detected by `conveyor_sensor3` within 35 clock ticks of the `watchdog` being launched. The exception handler is run by means of a call to `abort` \(^3\) which removes all agents currently being processed before launching the exception handler agent with an empty store. In this case, the agent `halt` simply turns the conveyor belt off.

```
conveyor_sensor3  conveyor_sensor2  conveyor_sensor1

\[
\begin{array}{c}
\text{token} \\
\end{array}
\]

Figure 4.9: Factory Conveyor Belt

\[
\begin{align*}
\text{conveyor} &:: \\
&\quad \text{do \{clps(conveyor:on)\}} \\
&\quad \text{watching clps(conveyor_sensor3:on)} \\
&\quad \text{timeout ( do \{clps(conveyor:on)\}} \\
&\quad \quad \text{watching clps(conveyor_sensor3:off)} \\
&\quad \quad \text{timeout ( next(5,clps(tick:on)) do \{clps(conveyor:off)\})} \\
&\quad \}.
\end{align*}
\]

\[
\begin{align*}
\text{watched_conveyor} &:: \\
&\quad \text{conveyor,} \\
&\quad \text{watchdog(35, clps(tick:on), clps(conveyor_sensor3:on), halt).}
\end{align*}
\]

\[
\begin{align*}
\text{halt} &:: \{\text{clps(conveyor:off)}\}.
\end{align*}
\]

Figure 4.10: Tcc Agent For Controlling the Conveyor Belt

\(^3\) See chapter 2 for the syntax of `abort`. 
always(A) :: A, next always(A).

whenever C do A :: if C then A else next (whenever C do A).

do P watching C :: if C else (P, next (do P watching C)).

do P watching C timeout B ::
  if C then B else (P, next (do P watching C timeout B)).

next(0,Signal) do A :: A.
next(N,Signal) do A :: [N]~(   
  if (Signal) then ((clpfd(N1 \#= N-1)), next next(N1,Signal))   
  else next (next(N,Signal) do A)
).

watchdog(Count, Trigger, WatchedFor, Recover) :: [V]^(
  clpp(V:Count)),
  do (dec_n_test(V,Trigger,Recover)) watching WatchedFor
).

%this agent used by watchdog
dec_n_test(V,Trigger,Recover) :: [CV,NV]$({
  if Trigger then     
  (     
    if (clpp(V:CV), clpfd(CV-1 #= NV)) then
     (     
      if clpfd(NV #= 0) then abort(Recover) else next {clpp(V:NV)}
     )
  )
  else
  (     
    if clpp(V:CV) then next {clpp(V:CV)}
  )
  }).

Figure 4.11: Definitions of tcc combinators
4.2.2 The Artifact

This thesis presents a new approach to diagnosing faults in dynamic systems and, in order to verify and evaluate these ideas, a physical model of a dynamic system was constructed as a testbed. The artifact in this case consists of a computer-controlled model of a manufacturing plant, built from LEGO. Constructing a physical model allowed tec to be tested as the control language for the system and a manufacturing plant was chosen for a testbed because this type of engineered system is an important potential application for model-based diagnosis. Manufacturing plants are dynamic systems usually with many concurrently operating but interdependent components. Furthermore, it was felt that by constructing a physical model, as opposed to writing a computer simulation, insight could be gained into the difficulties that would be encountered if these ideas are implemented in an industrial setting. This section gives an overview of the LEGO factory.⁴

![Figure 4.12: The LEGO Factory](image)

The model that was constructed simulates an environment where objects are moved around a factory in order to be processed at different stations. In the LEGO factory these items are represented by tokens which circulate in a loop composed of a conveyor belt, a mechanical arm and a motorized

⁴A note about the terminology used: the LEGO model of the manufacturing plant will be called the LEGO factory and the tec model of the plant which forms the SD will be called the tec factory model.
cart. The cart brings the tokens into position under the mechanical arm, which picks them out of the cart and places them at the start of the conveyor belt. The tokens then travel along the conveyor, passing under three different sensors along the way. These sensors are able to detect the presence of a token directly beneath them. When the token reaches the end of the conveyor, it slides back into the cart, which has travelled back down to receive it. Figure 4.12 shows a photograph of the testbed that was built and the diagram in figure 4.13 gives its structure.

<table>
<thead>
<tr>
<th>Signal</th>
<th>Allowed Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick</td>
<td>[on, off]</td>
</tr>
<tr>
<td>conveyor_sensor1</td>
<td>[on, off]</td>
</tr>
<tr>
<td>conveyor_sensor2</td>
<td>[on, off]</td>
</tr>
<tr>
<td>conveyor_sensor3</td>
<td>[on, off]</td>
</tr>
<tr>
<td>arm_yaw_sensor</td>
<td>[on, off]</td>
</tr>
<tr>
<td>cart_top_sensor</td>
<td>[on, off]</td>
</tr>
<tr>
<td>cart_bot_sensor</td>
<td>[on, off]</td>
</tr>
<tr>
<td>conveyor</td>
<td>[on, off, forward, reverse]</td>
</tr>
<tr>
<td>cart</td>
<td>[on, off, forward, reverse]</td>
</tr>
<tr>
<td>valve1</td>
<td>[on, off, forward, reverse]</td>
</tr>
<tr>
<td>valve2</td>
<td>[on, off, forward, reverse]</td>
</tr>
<tr>
<td>arm_yaw</td>
<td>[on, off]</td>
</tr>
<tr>
<td>electromagnet</td>
<td>[on, off]</td>
</tr>
</tbody>
</table>

Table 4.2: Valid Signals and Values for the LEGO Factory

Movement in the LEGO factory is controlled by five DC motors and the current state of the system is detected with six sensors. One of the motors moves the conveyor belt, one moves the cart and three are used to operate the mechanical arm. The arm is the most complicated component in the system, consisting of two links each actuated by a pneumatic piston. Each piston is controlled by a pneumatic valve that is opened or closed with an electric motor; depending on the valve position, the piston is either extended or retracted. The third motor connected to the arm is used to swivel it around its vertical axis. As mentioned previously, three of the six sensors are used to detect the token on the conveyor belt. The other three are used to detect motion limits. Two of these sensors are needed to determine when the cart has reached its top or bottom limits of its travel and one is used to detect if the arm is positioned over the cart or the conveyor belt. Table 4.2.2 shows the \( \text{clp}(S) \) signals used to interface with the LEGO factory.

The motors of the model are controlled by a Motorola 68HC11EVB microcontroller board running a small monitor program. This program implements several low level commands that the
driver, connected to the \textit{tcc} control program, issues to control the system. The controller board also samples all of the sensor inputs on command and returns the sensor readings to the driver. Communication between the controller board and the driver occurs over a serial interface. Table 4.3 shows the low level commands implemented by the controller board.

As discussed in section 4.2.1, \textit{tcc} is used to control the operation of the factory. \textit{Clp}(S) \textit{tells} are converted by the driver software into commands that are sent to the controller board. Similarly, at the start of every iteration of the \textit{tcc} interpreter, the driver polls the controller board for the current state of the system and then adds this information to the store as a set of \textit{clp}(S) constraints. For example, the \textit{tcc} control program fragment:

\begin{verbatim}
{clps(cart:on)},

whenever clps(cart_top_sensor:on) do {clps(cart:off)}.
\end{verbatim}

adds the \textit{clp}(S) constraint \texttt{cart:on} to the store and the driver code simultaneously sends the appro-
appropriate command (MON 2 1) to the controller board which then turns on the cart motor. Then, in every iteration, the driver will add the current state to the store. This will add \texttt{clps(cart\_top\_sensor:off)} to the store, assuming the cart has not yet reached its upper limit. When the cart does reach it, \texttt{clps(cart\_top\_sensor:on)} will be added, causing the \{\texttt{clps(cart:off)}\} command to be issued which stops the cart’s motor. As time progresses, a sequence of stores is produced which traces the execution of the program and captures the evolving state of the LEGO factory. This execution trace is used to provide the observations.

<table>
<thead>
<tr>
<th>Commands</th>
<th>Parameters</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>VER</td>
<td></td>
<td>Print version of the software</td>
</tr>
<tr>
<td>MON</td>
<td>(&lt;m&gt; &lt;1</td>
<td>0&gt;)</td>
</tr>
<tr>
<td>MDI</td>
<td>(&lt;m&gt; &lt;1</td>
<td>0&gt;)</td>
</tr>
<tr>
<td>MDC</td>
<td>(&lt;m&gt; &lt;HHHH&gt;)</td>
<td>Set motor (m)’s duty cycle (speed)</td>
</tr>
<tr>
<td>RIN</td>
<td></td>
<td>Return the 8 sensor inputs as a byte value</td>
</tr>
<tr>
<td>!</td>
<td></td>
<td>Return the 8 sensor inputs as 2 ascii characters (nibbles)</td>
</tr>
<tr>
<td>QUI</td>
<td></td>
<td>Exit monitor</td>
</tr>
</tbody>
</table>

where \(<m>\) is a motor number from 1-8 and \(<HHHH>\) is a 16 bit hexadecimal value.

Table 4.3: LEGO Controller Commands

### 4.3 \textit{tcc} Diagnosis

This section discusses using \textit{tcc} to perform diagnosis from two standpoints. In the first subsection a theoretical definition of \textit{tcc} diagnosis is presented. These concepts are then applied to the practical example of the LEGO factory. This example requires a system description and a \textit{tcc} interpreter in combination with the diagnosis procedure in order to locate the faults. These aspects are discussed separately.

#### 4.3.1 Diagnosis of \textit{Tcc} Programs

In section 3.2 the logical foundations for the consistency-based diagnosis of \textit{cc} programs were described. Essentially, a failure was detected when the observations conflicted with the behaviour predicted by the \textit{cc} model. A diagnosis for the failure was then a labeling, \(ab(c)\) or \(\neg ab(c)\) of all components in the \textit{cc} model, which would restore consistency with the observations. It was then shown that labeling a component \(c\) as abnormal (i.e. \(ab(c)\)) was equivalent to removing the component’s agent from the \textit{cc} system model. The consistency of the remaining agents was then checked.
through a process of *theory completion*. In this section these ideas are extended to deal with the more complex nature of \( tcc \).

With \( tcc \), programs are executed as a series of \( cc \) iterations and this affects their diagnosis in two main ways. First, a series of observations are required, with a unique set of observations applied to each iteration. This does not mean that observations are required for every iteration, as some states in the state machine may not have any observable variables, but the more observations there are, the more specific the diagnoses will be, and if there are no observations then a conflict can not be created. However, only one observation in one iteration is required to create an inconsistency. This means that in order for there to be consistency between the model and the observed behaviour all observations in all iterations must be consistent with the expected behaviour.

The second difference between \( cc \) and \( tcc \) diagnosis is that \( tcc \) programs can schedule agents for evaluation in future iterations. The act of deferring an agent's execution will never raise an inconsistency in itself, so if this delayed agent does create a conflict, it will only happen in the iteration in which the agent is evaluated. This makes diagnosis more computationally expensive because the event of scheduling the agent is separated from the time in which the agent is evaluated. In this case, the diagnosis procedure may need to backtrack across iterations in order to resolve the conflict. Therefore, the fact that agents can be delayed with \( tcc \) means that for diagnosis there are two classes of assertions: external observations and internally generated agents that have been carried forward. An error is detected if either of these creates a conflict.

As with \( cc \), a model written in \( tcc \) consists of a set of agents, each representing the constituent components of the artifact being modeled. This program is labeled \( P \) and the components it models are \( \text{COMPS}(P) \) which are defined over an underlying constraint system \( CS \). Each agent in \( P \) has the form \( p(\vec{X}) \rightarrow B_c \) where \( p(\vec{X}) \) is the agent’s name and parameter list and \( B_c \) is its normal behaviour. In defining the system description, the behaviour is augmented with the distinguished predicate \( ab(-) \) which removes the normal behavioural constraints, \( B_c \), if a component is labeled abnormal. Therefore, the system description is defined as before as

\[
\text{SD}(P) = \{(p(\vec{X}) \rightarrow (B_c \lor \text{ab}(p))) \mid (p(\vec{X}) \rightarrow B_c) \in P\} \cup \text{CS}.
\]

The difference from section 3.2 is that this same \( \text{SD}(P) \) is used repeatedly over several iterations. The initial iteration is labeled \( \alpha \) and the final iteration \( \Omega \). In each iteration \( i \), a set of assertions is presented. These assertions are made up of the external observations \( OBS_i \) as well as \( STATE_i \).
the agents carried forward from previous iterations (either of which may happen to be null for some or all iterations). This leads to the definition of a \textit{tec} diagnosis, which will restore consistency over all iterations \(i \in [\alpha, \Omega]\).

**Definition 4.3.1 (Diagnosis of a \textit{tec} Program)** Given a \textit{tec} program \(P\) composed of a set of components \(\text{COMPS}(P)\), a system description then says that each component behaves either normally or abnormally. Given a set of components \(\Delta \subseteq \text{COMPS}(P)\) containing only components assumed abnormal, then \(D(\Delta, \text{COMPS}(P) - \Delta)\) is a consistency-based diagnosis for \(\text{SD}(P)\), \(\text{COMPS}(P)\) and a set of assertions \(OBS\) iff

\[
\text{SD}(P) \cup D(\Delta, \text{COMPS}(P) - \Delta) \cup \bigwedge_{i=\alpha}^{\Omega} (OBS_i \cup \text{STATE}_i)
\]

is consistent. Where the \(\text{STATE}_i\) assertions are a function, \(F\), of the previous iterations, defined as:

\[
\text{STATE}_\alpha = \emptyset
\]

\[
\text{STATE}_i = F(\text{SD}(P) \cup D(\Delta, \text{COMPS}(P) - \Delta) \cup \bigwedge_{j=\alpha}^{i-1} (OBS_j \cup \text{STATE}_j)).
\]

The function realized by \(F\) is determined by the operational semantics of the interpreter.

This diagnosis \(D(\Delta, \text{COMPS}(P) - \Delta)\) is a partitioning of \(\text{COMPS}(P)\) into two sets: \(\Delta\), which consists of agents representing those components assumed abnormal, and \(\text{COMPS}(P) - \Delta\), the remaining agents representing the normally behaving components. Given the definition of \(\text{SD}(P)\), members of \(\Delta\) are of the form \(((p(\bar{x}) \rightarrow (B \lor ab(p))) \land ab(p))\), or more simply \(ab(p)\), and members of \(\text{COMPS}(P) - \Delta\) are of the form \(((p(\bar{x}) \rightarrow B) \land \neg ab(p)\). Since the predicate \(ab(\cdot)\) does not appear in a \textit{tec} program, checking a hypothesis \(\Delta\) for consistency reduces to removing the agents in \(\Delta\) from \(P\). This is done for all iterations \(i \in [\alpha, \Omega]\).

The remaining agents in \(P_\Delta\) are checked for consistency using the same \textit{theory completion} process as with \textit{cc}. Recalling from chapter 3, the removal of the agents in \(\Delta\) from \(P\) has the potential to cause some or all of the remaining agents to block indefinitely. This is because the \textit{asks} in the remaining agent’s antecedents can only be satisfied by the agents that have been removed from the program. With \textit{theory completion}, constraints that these \textit{asks} are blocked on are simply told to the store, allowing the blocked agents to continue. This is acceptable because the agents in \(\Delta\) can behave in any manner, including \textit{telling} the constraints needed to allow the remaining agents to run to completion.
With tcc, the removal of agents from $P$ in every iteration creates a large hole in the system description. This means that observations become very important in supplying missing information about behaviour. Also, just as with cc, theory completion creates explanations by supplying input and output values for the components that form the diagnoses. However, with tcc a series of explanations are generated which can be used to characterize a failure more completely than with cc.

4.3.2 The System Description

```prolog
arm( Token_pos ) :: { clpp(Token_pos:0) }.

sensor(Pos, Token_pos, Sensor) ::
  if clpp(Token_pos:Pos)
  then { clps(Sensor:on) }
  else { clps(Sensor:off) }.

belt( Token_pos ) :: [CP, NP]$(
  if (clpp(Token_pos:CP), clpfd(CP+l #= NP))
  then next ( clpp(Token_pos:NP) )
 ).

sensor1(Tp,S) :: sensor(1,Tp,S).
sensor2(Tp,S) :: sensor(3,Tp,S).
sensor3(Tp,S) :: sensor(5,Tp,S).

factory(Tp) ::
  arm(Tp),
  for(6) do {
    belt(Tp),
    sensor1(Tp,conveyor_sensor1),
    sensor2(Tp,conveyor_sensor2),
    sensor3(Tp,conveyor_sensor3)
  }
).
```

Figure 4.14: Lego Factory System Description in tcc

Figure 4.14 shows a simple tcc model of the factory used as the system description for diagnosing the LEGO factory. Of all the components of the factory, only the actions of the arm and the conveyor belt (with its sensors) are modeled. The model, therefore, has 5 agents representing the arm, the conveyor and the three sensors. A sixth agent named ‘factory’ models the whole system by grouping
the components together and connecting them by way of the variable $T_p$, which holds the current token position.

This $tcc$ program models the system as a state machine with six states, as is shown in figure 4.15. When the factory agent is first run, the arm agent sets $T_p$ to 0, modelling how the token is placed at the start of the conveyor; this initializes the state machine. The three sensor agents and the belt agent are then run for six iterations. In each iteration the sensors return (by adding constraints to the store) whether the token is underneath them. The belt agent moves the token forward one position at a time by scheduling the next token position to be one greater than the current position.

![Figure 4.15: The Six States of the Lego Factory](image)

4.3.3 The $tcc$ Interpreter

The $tcc$ interpreter is a Prolog program that executes the $tcc$ models. The interpreter is invoked in the Prolog environment with the command `tcc(+Initial.agent, -BL, +Delta, -Expl)` where the inputs are Initial.agent (which is not only the initial agent but also the observations) and Delta, a list of agents whose constraints are to be suspended. Delta is empty when performing consistency checking or when simply using the model to simulate the system. The interpreter returns two lists: the blocked list, BL and the list of explanations. The blocked list contains the agent(s) whose antecedents could not be satisfied when the program reached its final resting point. The explanations are the agents in Delta whose unrestrained port variables have been instantiated with those values or constraints that allowed the models to continue.

The interpreter maintains three queues of agents as it runs. These are the ‘ready-to-run’ queue (RTR), the ‘blocked’ queue (BL) and the the ‘nexts’ queue (Next). The RTR contains agents in the current iteration that are waiting to be evaluated. The BL contains agents which have been evaluated, but whose antecedents are not satisfied (and have not been proven false) by the store;
whenever new information is added to the store, the agents on BL are reevaluated to see if they can proceed. The Next list contains agents whose execution has been deferred until the subsequent iteration through the use of the next operator. The interpreter also maintains the store for each iteration. When quiescence is reached, the store is cleared in preparation for the next iteration but it is also saved in a history list. This history of past stores allows the evolving state of the system to be examined.

The interpreter continues to execute new iterations as long as there are agents in the Next list. When this list is empty, the final iteration will be executed and the interpreter will return 'yes' if it reaches its final resting point. If, however, the interpreter encounters an inconsistency, it will then backtrack, finally aborting execution and returning 'no' if it is unable to find a consistent store. This is how consistency between the system description and the observations is checked.

For example, to simply simulate the system, the interpreter is invoked with the system model, but without any observations. Delta is set to null because all components from the model should be used in simulation. In the case of the tcc model from figure 4.14 the interpreter would be invoked with tcc([Tp]^factory(Tp), BL, [], Expl). Since the variable Tp is not formally declared in the program, it is declared here as an existentially quantified variable since it names a port.

Figure 4.16 shows the results of executing the above command. The model runs for six iterations before terminating successfully. The store is then printed out allowing the execution trace to be viewed. Although there is only one store, the clp(S) and clp(P) constraints are shown separately to improve readability. The sequence of clp(P) stores trace the path of the token through the system. In figure 4.16, s1 is the Skolem constant representing the existential variable Tp and the current value of this port variable is given on the left side of the ‘—’ in every iteration. On the right side of the ‘—’ are the negated values. For instance, at t(0)

\[ ^{-}(s1:1) \]

means that the value of Tp (represented by s1) was not 1 in iteration 0. This constraint is due to sensor 1 which did not sense a token beneath it in the initial iteration. Similarly, the sequence of clp(S) constraints trace the expected sensor readings produced by the simulation.

4.3.4 The Diagnosis Procedure

Figure 4.17 gives an overview of the procedure used to diagnose faults in the LEGO factory. The first step in the process is to actually run the system in order to collect the observations. This
is done by using the \textit{tcc} control program to run the model and then saving the history of stores produced. However, at this point there is a conceptual mismatch between the actual system and the predicted behaviour as the \textit{tcc} factory model represents the system as having only six states while the control program produces a history of many more than six states. This is because the \textit{tcc} model uses the token position to trigger state changes, while, with the \textit{tcc} control program, state changes are due to the tick of a system clock. For example, as the \textit{tcc} control program runs freely, it may collect ten samples while the token moves through position \(\tau 0\) and then, when the token triggers sensor 1, 10 more samples of \(\text{clp}(\text{conveyor}\_\text{sensor1}:\text{on})\) as the token moves under the sensor. To overcome this problem, the store is filtered by extracting just those six observations which coincide with the token positions shown in figure 4.15.

Once the appropriate observations have been extracted, they are combined with the \textit{tcc} factory model.
model and checked for consistency using the tcc interpreter. If the model does not predict any behaviour that conflicts with the observations, then correct behaviour of the system is verified. On the other hand, if the interpreter returns “no”, then there is an inconsistency. In this case, the hypothesis generator will re-run the interpreter with different combinations of agents in Delta in order to find which combinations of agents will remove the conflict.

Although the tcc factory model uses just six states to represent the physical system, it is still possible to find meaningful diagnoses with this simplification. When performing diagnosis, the observations are expressed as part of the initial agent. Thus the initial agent is literally $SD \cup OBS$. For example, if at time $T_3$ the observation for sensor2 was $conveyor\_sensor2:off$ then this would be added to the invocation with the call

$$tcc(\lceil T_3 \rceil \cdot (factory(T_3), next \ next \ next \ \{clps(conveyor\_sensor2:off)\}), BL, [], Expl).$$

The three nexts delay the observation from being evaluated until time $T_3$, at which time it conflicts with the predicted behaviour of $conveyor\_sensor2:on$. This inconsistency causes the interpreter to eventually return “no”. Having detected a fault, the hypothesis generator then re-runs the interpreter with different combinations of components in Delta in order to diagnose the problem.

The above example shows that it is not necessary that there be an observation for every state. In fact, just one observation can cause a conflict. However, more observations will usually refine the diagnoses. If the only observation available is that at time $T_3$ sensor.2 is off, then there are three possible single faults that would explain this: either the arm is broken and so it never placed the token on the conveyor, or the conveyor is broken and it never moved the token into position under sensor.2 or sensor.2 itself has failed and does not detect a token that is actually there. If we then add the observation that a token was also detected by sensor.3 at time $T_5$, then this rules out the
arm and the belt as diagnoses because the token must have been placed on the conveyor and moved to position 5 in order to have been detected by sensor_3.

4.4 Active and Reactive Models

In the previous example of diagnosing the LEGO factory, the model used (figure 4.14) represented the behaviour of the entire system, both the artifact and the controller. This type of model is termed an 'active' model because it models the system and how it operates together.

It is also possible to write 'reactive' models. Reactive models are meant to represent only the behaviour of the physical system, including how it responds to commands from the controller. Besides responding to the same \( clp(S) \) control signals that the physical model responds to, it also emulates the sensor signals produced by the artifact. These outputs will likely trigger new commands from the control program. Thus, in the case of reactive models, the system description is the conjunction of both the model and the controller \( SD = Reactive\ Model \cup Controller \).

For example, the agent in figure 4.18(a) is an active model for the arm component of the factory. This is because the agent embodies the controller as well as the arm and it sets the token position to 0 as soon as it is run. Figure 4.18(b), on the other hand, is a reactive model. It 'reacts' to the controller signal \( clps(electromagnet:off) \) by setting the token position to 0 and it keeps it at 0 until the controller turns on the conveyor with the signal \( clps(conveyor:on) \).

\[
\begin{align*}
\text{(a) Active} & \\
active\_arm(\text{Token\_pos}) & : : \\
& \{ clpp(\text{Token\_pos}:0) \}. \\
\end{align*}
\]

\[
\begin{align*}
\text{(b) Reactive} & \\
passive\_arm(\text{Token\_pos}) & : : \\
& \text{whenever } clps(electromagnet:off) \text{ do (} \\
& \quad \text{do (next } clpp(\text{Token\_pos}:0)) \text{ watching } clps(conveyor:on) \\
& \quad ) \}. \\
\end{align*}
\]

Figure 4.18: Active and Reactive Models of the Arm Component

This approach has several benefits, first the models are more declarative because they represent...
just what the artifact does given different control signals. But more importantly, the controller program is now part of the system description and so faults in the controller can also be diagnosed.

4.5 Diagnosis of Timeouts

In section 4.3.4, a state-based approach to diagnosing dynamic systems is presented. In this approach the dynamic system is modeled as a determinate state machine. The observations were then assumed to coincide with the observable states. Although this technique works, it has several limitations that reduce its effectiveness.

One of the biggest limitations is that the history of stores must be reduced so that only the observations that coincide with the model's states are retained. This becomes difficult when the model is not working properly. For instance, if sensor_3 in the LEGO factory fails, there will be no way to tell when the token has moved from \( T_4 \) to \( T_5 \) based on the sensor readings (see figure 4.15). One solution could be to know how quickly the conveyor moves and how often the sensors are sampled in order to determine which sensor readings correspond to which position. However, this approach is inexact because there are no guarantees that the actual conveyor belt speed is constant. A better approach would be to avoid filtering altogether and to use all sensor readings in the diagnosis.

If sensor_3 does fail to detect a token, then a second problem occurs besides the fact that the transition from \( T_4 \) to \( T_5 \) is difficult to detect. This problem is that sensor_3 is used as a trigger to turn off the conveyor belt and to terminate the control program. If sensor_3 fails, then these events will never happen. To deal with this possibility, the tcc control program includes a watchdog timer which assumes a failure and shuts the system down if a token is not detected under sensor_3 within a set number of clock ticks.

In modeling the system, the behavior of this watchdog timer should also be introduced into the tcc model. However, this creates a problem because the watchdog requires clock ticks as its observations. This means that it is no longer possible to abstract time out of the system description. Adding the watchdog timer model to the SD greatly expands the number of states in the system description, but more importantly it also allows nondeterminacy into the diagnosis problem. The reason for this is that the speed of the physical model is not always constant and this introduces some variability into the observations, even when the system is operating correctly.

In the untimed model of figure 4.15 the token path was divided up into 6 regions. In effect, the model created a 6 state, state machine with one state per region and the observations were chosen...
TABLE 4.4: Observations of Correct Behaviour for Untimed and Timed Models

<table>
<thead>
<tr>
<th>Observation #</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Untimed</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>sensor1:on</td>
</tr>
<tr>
<td>2</td>
<td>sensor1:off</td>
</tr>
<tr>
<td>Timed1</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>tick:on, sensor1:on</td>
</tr>
<tr>
<td>2</td>
<td>tick:on, sensor1:on</td>
</tr>
<tr>
<td>3</td>
<td>tick:on, sensor1:off</td>
</tr>
<tr>
<td>Timed2</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>tick:on, sensor1:on</td>
</tr>
<tr>
<td>2</td>
<td>tick:on, sensor1:off</td>
</tr>
<tr>
<td>3</td>
<td>tick:on, sensor1:off</td>
</tr>
</tbody>
</table>

so that there was one observation per state. The addition of time changes the model so that state transitions are taken on clock ticks. This adds many more states to the model because many clock ticks occur while the token travels between the sensors. Also, because the belt speed is variable and not synchronized with the clock, there is some variation in the observations. This is shown in table 4.5, which gives a series of sensor readings for the token travelling from \( T_1 \) to \( T_2 \). The unique set of sensor readings for the state-based model of figure 4.15 can turn into different, yet still correct series of readings when clock ticks are introduced, as with the example sensor readings Timed1 and Timed2.

With a correctly working system the SDUOBS is consistent, but if there is some variation in OBS in a correctly working system then it means that the SD must have a degree of flexibility in order to maintain consistency. The solution for this need for flexibility is to do the modeling with constraints. Although tcc supports modeling with constraints, all of the examples presented so far have used basic equality constraints. However the need for more flexibility demands the use of less restrictive inequalities.

To model the variability in the speed of the conveyor, it is divided up into 30 regions. This number was chosen to allow a range of behaviours to be modeled, while still keeping the total number of states manageable. The variable speed of the conveyor belt is then modeled by assuming it to move forward between one and two regions every clock tick. Figure 4.19 shows this flexible model for the belt which extracts the current position (CP) from the store and then tells the next position (NP) to be greater by between 1 or 2 conveyor steps.

As time progresses, the difference between the minimum and maximum distances that the belt might have travelled becomes increasingly large. At the same time, an increasingly large number of different possible observations will still produce a consistency. With these qualitative models, the
observations are more important than ever because they have the side effect of instantiating the constraints with ground values.

This approach of using constraints to define the behaviour of the artifact produces much more flexible, qualitative models that are able to accept deviations in normal behaviour. However, the use of inequalities in the model has implications for the simulation process. This is because it is no longer clear when antecedents are entailed by a constraint. For example, if sensor_1 is located at position 10 and a one unit wide token has a range of possible positions between 8 and 10, then the token may or may not be under the sensor. When simulating the model, the interpreter deals with this by assuming that the token is under the sensor at the first instance it could be under the sensor. Therefore, in the absence of conflicting information, the interpreter will assume the Tp=10. If however, the contradictory observation that sensor_1=off is added, this assumption must be retracted. Note that this does not cause an inconsistency because

\[
\begin{array}{c}
pos(\text{Token}) \geq 8 \land pos(\text{Token}) \leq 10 \land off(\text{Sensor}_1) \\
\end{array}
\]

is not inconsistent. Instead, this information is used to refine the constraints planned on the token position which is changed to

\[
\begin{array}{c}
pos(\text{Token}) \geq 8 \land pos(\text{Token}) \leq 9. \\
\end{array}
\]

This underlines the importance of sensor agents in using observations to refine the constraints.

This behaviour of simulating with constraints can be accomplished simply by making those variables with inequality constraints on them, unrestrained variables. This will cause \textit{theory completion} to be used to give the constraint variables specific values as long as this does not conflict with any other facts. If it does conflict, the assertion is retracted and the range on the variables is refined.
Figure 4.20 gives the qualitative reactive model for the LEGO factory, however this model only includes the third sensor at the end of the conveyor. The belt agent models the conveyor by assuming it moves between one and two positions forward every clock tick. It does this by extracting the current position from the store and then using the clp(S) constraint system to modify the range of possible position for the belt in the next iteration. If the correct set of observations does not include a clock tick, then the next position will be the same as the current.

The blank agent models the token on the conveyor belt. The token is modeled as having a width of four units where the leading edge is four units ahead of the trailing edge. The sensor, located at position 35, extracts the token's leading and trailing edges from the store in order to determine if the token lies underneath it. The abvars agent is a dummy agent that is placed in Delta for the sole purpose of making Pa and Pb (the variables that hold the token's trailing and leading edge positions, respectively) unrestrained variables. If Pa and Pb are not ground then the sensor agent will use theory completion to tell Pa <= 35 and Pb >= 35 unless these constraints are inconsistent with any constraints already placed on Pa and Pb. When the theory completion is done, the sensor will be assumed to be on. If however, an observation that the sensor is off conflicts with this, then the negation of the constraint will be told and the constraints defining the current position of the token will be refined.

The factory agent is written in a reactive style. In the LEGO factory the the arm picks the token out of the cart and then holds it over the conveyor belt with its electromagnet. When the magnet is turned off, the token drops down to its initial position on the conveyor and the token remains there until the conveyor belt is turned on. This is modeled in the factory agent by setting the trailing edge of the token to position 0 and keeping it at 0 until the signal that turns on the conveyor is sent. When the controller turns on the conveyor belt by telling clps(conveyor:on), the factory agent will then start the conveyor_step agent which runs all the component agents in the factory. This factory_step agent is run in every iteration until a conveyor:off signal is received and the simulation ends.

If the sensor is faulty, then it will not detect the token and the watchdog agent will be used to halt the system. This is a case of a timeout, and the error will be detected due to the conflict between the model predicting the token being detected by the sensor, but having no such observation occur. This timeout will be diagnosed when the sensor agent is put in Delta. This removes the behaviour of the sensor from the model and it will not detect the modeled token underneath. Again the watchdog agent ends the simulation, and in this case, the predicted behaviour will match the observations and
consistency between them will be restored, thus confirming the sensor as a diagnosis.

4.6 Summary

The LEGO factory that was constructed turned out to be very useful for several reasons. First, it demonstrated conclusively tcc’s ability to control real-time systems. Tcc’s properties of declarative-ness and parallelism made the task of writing the control program very straightforward. Second, this physical artifact demonstrated the ability of tcc to model a dynamic system as a state machine, but it also uncovered a weakness of this approach. This weakness is due the difficulty in choosing which particular observations, out of a large set, correspond to the distinct states of the state machine model.

This difficulty led to the creation of a more flexible model of the artifact using inequality constraints. This model was able to accurately represent the slight variations in the behaviour of the artifact and removed the need to filter the observations. Also, by creating a ‘reactive’ model, it was possible to incorporate the controller into the system description. This allowed the model to behave in response to controller signals just as the actual artifact would. Since the controller contained a watchdog timer, the effects of timeouts could then be simulated and their causes diagnosed. This proved very effective, although the very large number of possible states in the flexible, reactive model greatly increased the time required to prove an inconsistency.
belt(Tp) :: [CP]$\{
    if clpp(Tp:CP) then ( 
        if clps(tick:on) 
        then { 
            {clpf(CP+2 #>= NP)}, 
            {clpf(CP+1 #=< NP)}, 
            next {clpp(Tp:NP)} 
        } 
        else { 
            {clpf(CP #= NP)}, 
            next {clpp(Tp:NP)} 
        } 
    ) 
\}.

blank(Ta,Tb) :: [CP,NP]$\{ 
    if clpp(Ta:CP) then if clpfd(CP+4 #= NP) then {clpp(Tb:NP)} 
\}.

sensor(Ta,Tb,Pa,Pb) :: 
    if clpp(Ta:Pa) then ( 
        if clpp(Tb:Pb) then ( 
            if clpfd(Pa #=< 35 #/\ 35 #=< Pb) 
            then {clps(conveyor_sensor3:on)} 
            else {clps(conveyor_sensor3:off)} 
        ) 
    ) 

abvars(_Ta,_Tb).

conveyor_step(Ta,Tb) :: [Pa,Pb]^\{ 
    abvars(Pa,Pb), belt(Ta), blank(Ta,Tb), sensor(Ta,Tb,Pa,Pb) 
\}.

factory(Ta,Tb) :: ( 
    whenever clps(electromagnet:off) do ( 
        do (next {clpp(Ta:0)}) watching clps(conveyor:on) 
    ), 
    whenever clps(conveyor:on) do ( 
        do conveyor_step(Ta,Tb) watching clps(conveyor:off) 
    ) 
) .

Figure 4.20: Reactive, Qualitative Factory System Description in tcc
Chapter 5

Conclusions and Future Work

5.1 Conclusions

This thesis has presented a novel approach to model-based diagnosis by applying the techniques of consistency-based diagnosis to the concurrent constraint languages cc and tcc. This is an important result because it directly addresses one of the biggest drawbacks of model-based diagnosis, the difficulty in creating useful models. Cc and tcc allow models to be written that are expressive, declarative, composable, flexible and able to model temporal behaviour. In conducting this research, interpreters for cc and tcc were written and then modified to incorporate the consistency-based diagnosis procedure, creating a generic framework for diagnosing cc or tcc programs or for models written in these languages.

Consistency-based diagnosis was first applied to the simpler cc language. Cc offers many advantages as a modeling language; its models are declarative, allow concurrent processes to be easily represented and incorporates the expressiveness of predicate logic and these beneficial properties extend to diagnosis. Most of the standard examples commonly cited in the diagnosis literature have been reproduced with models that are short, clear and declarative. In applying consistency-based diagnosis to cc, a process of theory completion was identified, along with its ability to provide sample input and output values for failed components, which aid in ‘explaining’ the diagnoses. Furthermore, it was shown that fault models could be incorporated and that this framework could be used to perform another form of model-based diagnosis: abductive diagnosis.

This diagnostic approach was then extended to tcc, allowing temporal models of artifacts
exhibiting much more complex behaviour to be written for both simulation and diagnosis. This is important because this diagnostic approach with tec is the first we are aware of that is able to use a first-order temporal modeling language while existing comparable approaches use propositional temporal modeling languages. Additionally, tec's usefulness as a real-time control language was demonstrated by its ability to control a LEGO factory model constructed as a testbench. This too is a useful result because not only does it demonstrate tec's applicability in filling very different roles, it also means that faults in both the control program and in the model can be diagnosed with the same procedure. With this testbench, tec's ability to model and diagnose systems that can be represented as a series of discrete states was demonstrated.

Finally, it was shown that the inequality constraints of tec allow flexible, qualitative models to be constructed that are able to reflect a range of correct behaviours while still conflicting with observations of faulty behaviour. Furthermore, together the temporal and default logic properties of tec allow timeouts to be detected and their causes diagnosed. Timeouts are very commonly used to detect errors in industrial systems and, to the best of our knowledge, this is the first approach able to diagnose possible causes for these failures.

However, there are still limitations to this approach, these have been addressed in the next section.

5.2 Future Work

Although this research has shown that cc and tec can be adapted for use with consistency-based diagnosis, this work is really only a preliminary investigation. In the course of conducting this research many questions have arisen that demand further investigation before industrial application of these ideas will be feasible. In other cases, new opportunities have appeared that offer great potential to extend these techniques even further.

• Scalability
In almost all model-based diagnosis literature, including this thesis, only small, illustrative examples are presented. If these techniques are to achieve wide spread use, they must be able to scale up to the much greater complexity of real-world systems. However, many model-based techniques do not scale well because the search space for determining consistency grows exponentially with the number of components in the model. Tcc however offers some hope in this area, as it supports the creation of hierarchical models. The creation of hierarchical
models and their use in diagnosis deserve more research.

- **Mode Identification and Planning**
  In the work of Williams and Nayak for Deep Space One [32] it was shown that the same models used for diagnosis could also be used for mode identification and for planning. Similar to diagnosis, mode identification is the use of a system model and available observations to determine the current state of the artifact. Planning uses the same model to determine what inputs are required to take the artifact from its current state to a new, goal state. Both of these activities are possible with the modified tcc interpreter used for this thesis, but doing so has not been explored.

- **Probability and Cost Functions**
  With basic consistency-based diagnosis, there are often multiple diagnoses produced for an observed failure, but there is no way to discriminate among them, other than to say that single faults are more likely than double faults, etc. With the addition of failure probabilities, it becomes possible to rank diagnoses from most to least likely. If tcc models are being used for planning, then it is also possible to associate a cost function with the state transitions allowing the lowest cost route to a goal to be planned.

- **Sharing Constraints**
  Although cc and tcc are built on top of one or more underlying constraint systems, the implementation does not allow different constraints to be shared among variables. This is a problem with the implementation of the constraint system.

- **Improving Search Efficiency**
  One problem with using tcc for consistency-based diagnosis that has become apparent is that models constructed from constraints are consistent with a large range of behaviours. This means that when observations of faulty behaviour are introduced, it requires the examination of a very large search space to determine that this faulty behaviour is indeed inconsistent with the model. Solving this problem needs more research and may require the use of heuristics.

- **Extending tcc Diagnosis to hcc**
  Hybrid cc [26], or hcc, is yet another language in the family of cc languages. Hcc models systems that vary both continuously and discretely with time (hence "hybrid"). As such, it presents a whole new range of possibilities in diagnosis. Hcc may allow even more complex
systems to be diagnosed, especially those already characterized by differential equations.
References


