VERIFICATION OF
MATHEMATICAL DESIGN DOCUMENTS

Zhiying Liu

A Thesis Submitted to the Graduate School
in Partial Fulfillment of the Requirements

for the Degree of
Doctor of Philosophy

Supervisors
Prof. David Lorge Parnas
Prof. Brian Fitzgerald

Software Quality Research Laboratory (SQRL)
Computer Science & Information Systems
University of Limerick

Submitted to the University of Limerick, October 2010
Verification of Mathematical Design Documents

Zhiying Liu

Abstract

The problem to be studied in this thesis is how to verify a set of mathematical design documents in the sense that if each component is implemented correctly, the whole system should work together harmoniously to satisfy the system requirements.

In general, a software modular design is often a procedure of 1) decomposition of a complex system into manageable modules that can be developed separately, 2) specifying and developing each component individually, and 3) combining the components back into a system. The collection of software components that is aimed at finishing a required task together must cooperate with each other through some communication paths. We consider the interconnection description of such a collection of software components as an integral part of a software design.

In this thesis, we present an approach to a) describing a network of components, b) checking whether the network is completely connected and whether it is consistent, i.e., whether the components can work together properly, c) determining the behaviour of the network, i.e., what the components will do if they are combined together as described in the network description, and d) checking whether the behavior of the network conforms to the requirements, i.e., if the network behaves as required.

We have shown how to determine the behavior of a network of components from the description of the internal structure and the behavioral description of each component, established the conformance relation for checking the correctness of a design, and presented a procedure for checking the consistency and correctness of a set of fully specified components on the basis of their interface descriptions and a description of the way in which the components are connected. Three case studies are presented to illustrate how the approach works. In these case studies, the following steps are included: specifying the system requirements, decomposing the system, describing the behavior of each component, defining their interconnections, followed by a completeness and consistency checking of the component network as well as the derivation of the network behaviour and the correctness checking of this design before investing time in building the components.

In the approach developed in this thesis, each component is viewed as a hardware-like device in which the value of an output variable can change instantaneously when input values change and all components operate synchronously rather than in sequence. The connections among components are through shared variables. The behavioural requirements for the software are described by a representation of the relation between the history of input and output values and the current value of the output variables. The behavioural description of the network is derived by using classical relation composition; and the conformance relation is defined using mathematical concepts and standard mathematical logic. Therefore the approach is fully mathematic-based and can be implemented automatically.
Declaration

The work described in this thesis is, except otherwise stated, entirely that of the author and had not been submitted as an excuse for a degree at this or any other university.

____________________
Zhiying Liu
October 2010

The work described in this thesis is, except otherwise stated, entirely that of the author.

____________________
Professor David Lorge Parnas

____________________
Professor Brian Fitzgerald
Acknowledgments

This work was supported by Science Foundation Ireland and Department of Computer Science & Information Systems.

First and foremost I would like to express my sincere thanks to my supervisor, Professor David Lorge Parnas, who has given me this invaluable opportunity and guided me to the challenging and interesting subjects, for his continuous support through the entire span of my studies, including the time after his retirement. Dave constantly read my reports and gave advice, helped me stay on the right track when I was losing directions, especially the steady guidance in the main topic and the essence of this work. Dave read through several versions of this thesis and made numerous corrections that helped greatly to improve the quality of the thesis. Without his constant advice, support, trust and encouragement, it would be impossible for me to finish this work.

My deep gratitude goes to my on campus supervisor, Professor Brian Fitzgerald, who has made available his support in many ways, for all of his great help that makes it possible for the work coming to an end. I am really indebted to him. Thank Ms. Margaret Fennessy and Ms. Marie Beaumont for their administrative assistance.

My thanks are due to Dr. Patrick Healy, Dr. Donal Heffernan and Prof. Ridha Khedri for serving as my examining committee and for the valuable feedback they provide.

I am grateful to Dr. Baltasar Trancon y Widemann for his valuable suggestions and inspirations that have undoubtedly contributed to this work. During his mentorship, Baltasar always gave me advice whenever I needed help. For many times when I was stuck, it was his suggestions that allowed me to move forward. I wish we could work together longer than only one year.

My gratitude is extended to Dr. Annette McElligott, Head of CSIS Department, for her kind support.

I want to thank Dr. Adam Balaban and Dr. Xin Feng for their precious time to discuss with me. Especially Xin, who acted not only as an experienced scientist listening to my questions and giving suggestions, but also as a good friend listening to my worries when I was feeling down. I like to thank Ms. Ita Page, who always provides good ideas, and Ms. Mhici Ni Mhurchu for their official and private help that made my study and life in Ireland smooth and pleasant. My special thanks go to Lillian Chik-Parnas, who has always encouraged me and given me wise advice in many aspects.

Thank my colleagues Salmi Baharum, Jidong Huang and Rachel Zhu for their friendship that makes the life in Limerick more delightful. My thanks are extended to all colleagues of the SQRL at the University of Limerick who shared their ideas with me through the weekly presentations and discussions.

Finally, I would like to express my appreciation to my parents, my husband Feng and my daughter Anna Yuhan, for their encouragement, patience, sacrifice, and love that helped me go through the hard time along the path towards my Ph.D. This thesis is dedicated to them.
3.1.3 Checking the Correctness of a Design ............................................................. 31
3.2 Describing Networks of Components ............................................................... 31
3.3 Other Approaches ............................................................................................ 34
  3.3.1 Component Diagrams in UML ................................................................. 35
  3.3.2 Architecture Description Diagrams in ADL .............................................. 35
  3.3.3 Data Flow Diagrams ............................................................................... 36
  3.3.4 Function Block Diagrams ........................................................................ 36
3.4 Completeness of a Network of Components ..................................................... 37
3.5 Consistency of a Network of Components ........................................................ 38
  3.5.1 Data Type Consistency ........................................................................... 39
  3.5.2 Delays in a Component .......................................................................... 40
  3.5.3 Delayed and Delay-free Loops in a Network .......................................... 40
  3.5.4 Why Exclude Delay-free Loops? .............................................................. 43
  3.5.5 When is a Network of Components Consistent? .................................. 45

4 Determining the Behavior of a Network of Components ...................................... 46
  4.1 Internal and External Behavior of a Network .................................................. 46
  4.2 Derivation of an Internal Behavioral Description of a Network ..................... 48
    4.2.1 Derive the Set of Possible Network Traces \(T_d\) .................................... 49
    4.2.2 Extend the Domain of Component Trace Functions ............................... 52
    4.2.3 The Derivation Relation ....................................................................... 56
    4.2.4 A Simple Example without Loops .......................................................... 58
    4.2.5 Does the Network Have Well-defined Behaviours with Delayed Loops? ...... 63
  4.2.6 The TFM Descriptions for a Network of more than Two Components ....... 66
  4.3 Simplifying Expressions in the TFM Descriptions of a Network ................. 67

5 Correctness of a Network of Components ............................................................ 70
  5.1 The Correctness of a White Box ..................................................................... 70
  5.2 The Basic Conformance Relation .................................................................. 71
  5.3 Verify the Description of the Behaviour of a Network .................................. 72

6 Case Studies ........................................................................................................ 77
  6.1 RS Flip-flop .................................................................................................. 78
    6.1.1 The Requirements TFM Specification of the RS Flip-flop ....................... 78
    6.1.2 The Network of Components ................................................................. 79
A2.4 The result comparison

A3 The Flight Reservation System

A3.1 The Definition and Specification of Data Types
A3.2 The Specification of the System Requirements
A3.3 The TFM Description of the Constituent Components
A3.4 The Derivation of the Network Behavioral Description
A3.5 Convert the System Trace Functions to Network Trace Functions
A3.6 Conformance Checking

Bibliography
List of Figures

Fig. 2.1: A Stack as a Component ................................................................. 21
Fig. 3.1: Looped Connections between Two Components .......................... 41
Fig. 3.2: Delay-free and Delayed Loops ....................................................... 42
Fig. 3.3: NOT Gate in an Unstable Network .................................................. 43
Fig. 3.4: OR Gate in a Network with Two Solutions ................................. 44
Fig. 4.1: Internal and External Views of a Network of two Components ....... 47
Fig. 4.2: C1 Connected to C2 to Produce N .................................................. 49
Fig. 4.3: The Network Event Descriptor and the Component Event Descriptors .... 50
Fig. 4.4: The Union of Event/Dummy Event Descriptors at t₀ for the Possible Cases .... 52
Fig. 4.5: Commuting Diagram for Variables ø₁ and ø₁ .................................. 55
Fig. 4.6: The Input/Output Waveforms of a “Flip-flop” .............................. 58
Fig. 4.7: Two Connected “Flip-flops” ......................................................... 58
Fig. 4.8: The Waveforms of the Relevant Variables in the Network ............... 63
Fig. 4.9: C₁ Connected to C₂ to Produce N₂ ................................................ 66
Fig. 4.10: Relation of Values in Network Trace T and Component Traces Tₖ ........ 68
Fig. 5.1: The First Commuting Diagram for Conformance Checking ............ 74
Fig. 5.2: The Second Commuting Diagram for Conformance Checking .......... 75
Fig. 6.1: A Network of Components for an RS Flip-flop .............................. 79
Fig. 6.2: A Network of Components for an RS Flip-flop with a Delay-device ...... 81
Fig. 6.3: The Footnote Convention for the Event Descriptors ...................... 83
Fig. 6.4: The Black Box of a Temperature Monitor System .......................... 90
Fig. 6.5: A Network of Components for the Temperature Monitor System ....... 92
Fig. 6.6: A Network of Components for the Flight Reservation System .......... 101
List of Definitions

<p>| Definition 2.1: software component | ................................................................. | 21 |
| Definition 2.2: component interface | ................................................................. | 22 |
| Definition 2.3: syntax of an event descriptor | ................................................................. | 24 |
| Definition 2.4: before value and after value of variable v | ................................................................. | 24 |
| Definition 2.5: syntax of a trace | ................................................................. | 24 |
| Definition 2.6: component interface TFM specification/description | ................................................................. | 26 |
| Definition 2.7: set of possible traces for a component | ................................................................. | 28 |
| Definition 2.8: syntax of a component dummy event descriptor | ................................................................. | 28 |
| Definition 2.9: descriptor within a component at any time t₀ | ................................................................. | 29 |
| Definition 3.1: network of components | ................................................................. | 32 |
| Definition 3.2: completely connected | ................................................................. | 37 |
| Definition 3.3: type | ................................................................. | 39 |
| Definition 3.4: subtype | ................................................................. | 39 |
| Definition 3.5: value-type | ................................................................. | 39 |
| Definition 3.6: value-subtype | ................................................................. | 39 |
| Definition 3.7: data type consistency | ................................................................. | 39 |
| Definition 3.8: immediate dependency description | ................................................................. | 41 |
| Definition 3.9: immediately influences | ................................................................. | 41 |
| Definition 3.10: delay-free loop | ................................................................. | 42 |
| Definition 3.11: network consistency | ................................................................. | 45 |
| Definition 4.1: relation between component trace pairs and network traces | ................................................................. | 50 |
| Definition 4.2: relation between network traces and component traces | ................................................................. | 53 |
| Definition 4.3: the TFM description of a network of two components | ................................................................. | 56 |
| Definition 4.4: the TFM description of a constructed component | ................................................................. | 57 |
| Definition 5.1: conformance relation | ................................................................. | 72 |
| Definition 5.2: syntax match between CITₜ and CIT | ................................................................. | 72 |
| Definition 5.3: relation π₀ between network traces and system traces | ................................................................. | 72 |</p>
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Defined Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>TFM</td>
<td>Trace Function Method</td>
<td>4</td>
</tr>
<tr>
<td>TAM</td>
<td>Trace Assertion Method</td>
<td>4</td>
</tr>
<tr>
<td>PVS</td>
<td>Prototype Verification System</td>
<td>6</td>
</tr>
<tr>
<td>SCR</td>
<td>Software Cost Reduction</td>
<td>6</td>
</tr>
<tr>
<td>TTS</td>
<td>Table Tool System</td>
<td>7</td>
</tr>
<tr>
<td>MIL</td>
<td>Module Interconnection Language</td>
<td>10</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
<td>11</td>
</tr>
<tr>
<td>EML</td>
<td>Extended Meta Language</td>
<td>11</td>
</tr>
<tr>
<td>SML</td>
<td>Standard Meta Language</td>
<td>11</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
<td>12</td>
</tr>
<tr>
<td>CIT</td>
<td>Component Interface TFM specification/description</td>
<td>26</td>
</tr>
<tr>
<td>NoC</td>
<td>Network of Components</td>
<td>32</td>
</tr>
<tr>
<td>DFD</td>
<td>Data Flow Diagram</td>
<td>36</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
<td>36</td>
</tr>
<tr>
<td>RS</td>
<td>Reset Set</td>
<td>77</td>
</tr>
<tr>
<td>TMS</td>
<td>Temperature Monitor System</td>
<td>90</td>
</tr>
<tr>
<td>FRS</td>
<td>Flight Reservation System</td>
<td>98</td>
</tr>
<tr>
<td>VCAD</td>
<td>Variable Create, Alter, Destroy</td>
<td>103</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

The problem addressed in this thesis is how to verify a set of mathematical design documents in the sense that if each component is implemented individually according to its specification, the whole system should work together harmoniously to satisfy the system requirements. The motivation of this consideration is the decomposition and the integration of a complex software system and the problems that may arise in this development procedure.

We do not consider non-functional requirements. This thesis assumes that the requirements are functional requirements. What it meant by “functional” was clearly described in [88]:

“It is important to note that we are not using “functional” in its vernacular sense, but with its standard mathematical meaning. In the vernacular “function” often means purpose, role, or activity, as in: “The function of this system is to control the level of water in the tank”. In mathematics, function means a mapping between two sets of elements (called domain and range, respectively) such that every element of the domain is mapped to exactly one element in the range”.

It can be seen that the functional requirements here is different from other definitions such as the one in [17], which defines functional requirements as “what a system must be capable of performing” and non-functional requirements as those “not what the system will do, but how they will do”. In the Parnas-Madey view, the essential properties of computer systems, and their components, including how the system should perform, when really understood, are functional and can be expressed in relational terms. The non-functional requirements or “softgoals” such as: “the system should be modifiable”, “the system should have good performance” and “better, cheaper faster system” [16, 17] are not considered in this thesis since this kind of requirement is vague and not precisely documented. For these statements to be meaningful, it would be necessary to say what aspects should be easily modified and how easy, what level of performance is considered
good, and what would make a system "better". We believe that a requirements document is not complete until everything required is described in the functional documentation, as stated in [88].

1.1 Decomposition and Integration

In general the process of modular design of a software system comprises:

- Analyzing and specifying the system requirements;
- Decomposing the complex system into manageable modules;
- Specifying the behavior required of each module;
- Describing the structure of the system and the interaction of the components;
- Analyzing and verifying the design to be sure that if each component meets its requirements, the behavior of the collection of components will meet the system requirements;
- Developing, inspecting and testing each of the components;
- Assembling the components to a system and testing them as a whole.

Modularity allows dealing with each module while ignoring the interaction of modules and dealing with the overall characteristic of all modules and their relationships without considering details of any modules.

However, the nature of software development activities, such as frequent requirement, design and implementation changes, reuse of some parts of one system in another, and the massive amount of information generated in the development, makes the assembling of the large numbers of inter-related components difficult.

Recognizing that excessive inter-module dependencies are an indicator of poor software design, in [76] Parnas pointed out that the effectiveness of a modularization depends critically upon the criteria used in dividing the system into modules, and proposed information hiding as a useful criterion for decomposition. Information hiding means the hiding of design decisions that are most likely to change inside each work assignment, thus

- making it possible to write a program that uses the module without knowing any unnecessary information, and
• insulating other parts of the program from extensive modification, should the
design decision be changed.

Besides reduction in complexity and cost of development task, modularization with
information hiding has other benefits such as flexibility, easier augmentation, reusability
and simplicity of the module interfaces.

However, information hiding removes only unnecessary dependencies. It allows people
to work on the components separately but the components will still depend on each
other, since they are supposed to work together to perform a more complex task. To
ensure the success of a modular design, other good design principles such as separation
of concerns [27, 30] and stepwise verification [28, 109, 70] are necessary. A design must
clearly identify the composing components, each interface of component must be
precisely specified, and the connections of the components must be clearly described in a
way that allows the design be verified in a sequence of small steps.

1.2 Black Box and White Box

Stepwise refinement and verification process requires a box-structured system design
that produces an implementation from a specification. Such a system design is defined
by a sequence of small design steps that permit the immediate verification of their
correctness [70]. The principle of information hiding distinguishes between concrete,
visible internal structure in a system with persistent information about the system’s
external behaviour. The external behavior can be described by its trace functions with no
reference to any internal structure information [92], which is a black-box description.
Each black box can be decomposed to small boxes that will interact with each other to
produce the functionalities required by the big black box; the external behavioral
description of each internal small boxes and the interaction description of these internal
boxes are a white-box description of the big box.

To verify that a white box has been designed correctly to provide the black-box behavior,
some derivation work has to be done on the white box in order to be compared with the
intended black box. These verifications should be carried out as early as possible in the
development procedure.
Boehm observed in 1987 [9] that finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the requirements and design phase. Today, there might be more dramatic cost escalation factors, depending on the size of the system, the way the system has been developed, the kinds of errors made, etc. In any case, the insight or belief that the cost of change escalates as development proceeds has been a major driver in focusing industrial software practice on thorough requirements analysis and design, early verification and validation, and up-front prototyping and simulation to avoid costly fixes [10].

In order to prevent developers from wasting time implementing components that will not work together or will not behave as expected, the consistency of the component interfaces and the correctness of a design should be checked logically in the earlier stage of the development; and the checking makes sense only when a design has been conducted and documented in a disciplined and checkable way.

1.3 Precise Documentation of Component Interfaces

In [90] Parnas and Madey considered that documentation can often be as important as the product itself. Applying the information hiding principle requires that one defines and documents the interface precisely without revealing any information that should be hidden. Only in this way can one expect that the implementers of other modules do not use information about the internal structure so that future changes will affect as few modules as possible. To define precisely what goes to interface that is visible to its users and what remains hidden in the implementation and can be changed at any time without affecting the users of the component is not an easy task. It needs good design principles and documentation methods.

TFM, the Trace Function Method [92], is the result of decades-long effort to find a practical way to document component interfaces. Starting with the work in 1970s [77, 40, 63, 7, 82, 43], different module specification methods and techniques have been proposed. The early work in software specification techniques is surveyed in [39] and reviewed in [104]. Among the various methods and technologies in formal specification, the Trace Assertion Method (TAM) proposed by Bartussek and Parnas [7] supports the information hiding principle and presents a realistic model for specifying the interfaces
of software modules. This TAM had been improved in the work of [86, 89, 104] and in [92] it was developed to the new version – the TFM.

TFM can be understood as a discrete version of the input-output relation in [90]. The behavioural requirements for the software can be described by a representation of the relation between the history of input and output values, which are described as functions of time. Since software components can be viewed as operating at discrete points in time, in TFM the time functions are replaced by event sequences.

A TFM specification/description begins with the specification of the input/output variables and their types. It then describes the external programs that give access to the data stored by the component and the corresponding event descriptors. Finally, it describes the value of each output variable as a function of the history of inputs/outputs. For more details about TFM, readers are referred to [92]. The work in this thesis is based on documents written using the TFM, but it is also compatible with other documentation method of providing component interface descriptions by documenting the relation between input/output histories and current output values. The essential property of this documentation is that the value of each output at time $t$ is described as a function of the history of the input and output values up to and including time $t$.

### 1.4 Consistency Checking of Mathematical Documents

Although mathematical documents are precise and unambiguous, they are also highly detailed; oversights and other errors are quite common. However the big advantage of this kind of documents is that they can be checked logically and automatically. To detect the possible errors in such documents before other decisions are based on incorrect information, we should validate documents as early as possible [82, 88]. The validation of design documents using pre-established criteria is a technical task that can be carried out only if the design document is precise enough to permit systematic analysis [92].

The checking of documentation includes:

1) Checking completeness of individual documents;
2) Checking the consistency of documents describing communicating components, and
3) Checking that a set of components will satisfy the overall requirements assuming that each individual component satisfies its specification and the inter-component connections are as described.

In this thesis, it is assumed that completeness of the interface specifications for the individual components have already been checked. The main concern in this work is checking the compatibility of the interconnected components and the conformance between a set of design documents and the system requirements specification. If the component interface descriptions are written in the TFM [6, 92, 95], the individual component documents can be pre-checked by tools such as those introduced in [55, 105].

**1.5 Some Earlier Approaches to Documentation Checking**

In this section some earlier approaches related to documentation checking are reviewed. Since we classify documentation checking into the checking of local properties in the document of one component, and the checking of global properties among related documents, we review some of the work in both categories as below.

**1.5.1 Checking of Individual Documents**

In parallel with his advocate for mathematical documentation, Parnas [88] expounded the importance of proving certain theorems to detect earlier errors in a mathematical document; and formulated some theorems such as domain coverage theorem and disjoint domain theorem that one should prove to check for simple, application-independent properties. These theorems can be transcribed to the language of PVS (Prototype Verification System) and be proved automatically using PVS proof checker [97, 55].

PVS is a research prototype developed at the Computer Science Laboratory of SRI international [73] and has been used in the formal specification and verification such as the work in [59, 5]. Since the PVS theorem prover accepts input which is encoded in its own specification language, the main focus of later work using PVS is to translate a specification into an acceptable form. For example in [59] the authors described how safety demonstration on requirements can be accomplished by taking advantage of PVS and developed an automated method of translating SCR-style requirements (Software Cost Reduction language) into PVS input language. The newer work in [5] describes an ALFA2PVS (ADA Language for Flight Software Application to PVS) translator that
converts the software in ALFA to PVS specification, so that the software can be verified with respect to underflow/overflow errors and divide by zero conditions without the actual execution of the code.

Several studies have been conducted based on the Table Tool System (TTS) developed in McMaster University [99] - a system for writing and checking tabular specifications. The work of Jing [55] describes an extension to TTS that allows the tool to derive theorems from the tabular specifications and send them to PVS so that the theorem prover can be used to check the tables for completeness and consistency conditions automatically. In [110], Wu focused on extending untimed functional method to the PVS real time method and provides guidance on the use of theorem provers to specific classes of problems and in developing libraries to facilitate specification and verification. Wei [105] developed a tool for automatically checking the errors in tabular expressions like wrong syntax, undefined variables and circular definitions in SCR requirement specifications. It provides syntax to represent the simplified SCR behavior requirements and describe the requirements specification using a framework file and a TTS context file.

Model checking [20, 94] was proposed as a verification technique that encompasses a set of algorithms for verifying properties of state transition systems by a search of their associated state transition graphs. However, the state explosion problem, wherein the number of system states grows exponentially with the number of system components, generally limited such techniques to designs with less than a million states [18]. Later on techniques such as symbolic state space exploration [14, 42], bounded model checking [18], predicate abstraction [33], symbolic model checking [13], and bounded event tracing [32] came into being to improve the performance of model checking.

Researchers at the Naval Research Laboratory (NRL) have been worked on developing the SCR method for specifying the requirements of software systems using tables, and on model checking of the SCR requirement specification since 1990s [46, 47]. The properties checked include domain coverage, type correctness, missing cases, circular definitions, and determinism in requirements specifications. This model represents a software system as a finite state automaton, which produces externally visible outputs in response to changes in monitored environmental quantities. State explosion problem was
also discussed [8] and limited by verifying abstractions of the original requirements specification. In the work of [13], symbolic model checking has been extended to infinite state models using symbolic representations that encode infinite sets. This work investigates the application of an infinite state symbolic model checker called Action Language Verifier to formal requirements specifications of safety-critical systems represented in the SCR tabular notation, which is used to check properties such as domain disjointness and domain coverage.

The requirement specifications expressed in Requirements State Machine Language by Heimdahl et al. [44, 45] can be checked to avoid conflicting requirements and undesired nondeterminism. In [44] the prototype implementation uses Binary Decision Diagrams to perform the analysis, and the work in [45] investigates how the PVS and its theorem proving component can help to increase the accuracy of their analysis.

The thesis work of Wei [106] studies abstraction in model checking based on exact-approximation, which allows for verification and refutation of temporal properties within the same abstraction framework. The work starts with symmetry reduction that exploits symmetry in programs for abstraction, addressing problems such as identification of virtually symmetry and support for symbolic model checking. It also extends counter abstraction to fully virtually symmetric programs, which avoids the bottleneck problem of building orbit relations in symbolic symmetry reduction. Exact-approximation can be achieved using different partial modeling formalisms.

Methods and tools for SAT-solving check the satisfiability of various kinds of formulas and become an important complement to traditional model checking [112, 18, 67]. Satisfiability is the problem of determining if the variables of a given Boolean formula can be assigned in such a way as to make the formula evaluate to true. In combination with traditional techniques it is able to handle more verifications tasks consistently.

Most of the above work checks the requirement specification and investigates whether one table or one document is internally complete and consistent. These techniques are suitable for checking the individual component specifications.
1.5.2 Checking of a Set of Related Documents

The checking of individual documents is only the first step towards a correct design, because, even if each component is individually well-behaved, the components might not fit together to fulfil a required task. They might not be compatible, for example, the types of connected inputs and outputs might not match. Therefore it is necessary to check the consistency of documents describing communicating documents at the same level, and the conformance between the behaviour of a set of components and the black-box requirements a level above.

For this purpose, the question of how the components are connected and how the connections are documented must be answered first. This leads to the consideration of software architecture. In [27] Dijkstra proposed the notion of a layered structure, in which programs were grouped into layers, and pointed out the elegant conceptual integrity exhibited by such an organization, with the resulting gains in development and maintenance ease. This is a view of program structure.

From the module structure point of view, Parnas stressed this line with his contributions concerning information hiding modules, modular structures and program families by:

• Stating that one should begin the decomposition with a list of difficult design decisions or design decisions which are likely to change [76], because of the benefits of applying information hiding (given on page 2). Information hiding can be elaborated by a hierarchically-structured document called module guide - an informal document explaining what information is hidden by each module in the system [85].

• Showing that there are different types of hierarchical structures which are described by different relations for different purposes; and that one should be clear about the different meanings of the term and be sure which kind of layered structure one needs [78]. For example the “uses” relation and the hierarchical structure it characterizes can be used to design software for ease of extension and contraction, since the “uses relation” can identify subsets or extensions [83]. The relation between information hiding and software structure is explained in [108].
• Defining program families, describing the importance of identifying commonalities among family members before analyzing individual members, and the benefits of postponing the decisions in a later stage; showing that the concept of program families provide a way of considering program structure more objectively; and proposing the information hiding module specification method, which is the basis for object-oriented programming where inheritance served as a way of dealing with variability, to construct program families [80]. Information hiding allows decision postponement, therefore an information hiding structure is suitable for program families and it has grown into the field of product lines. The connection between information hiding and good product line architecture has been discussed by Weiss [107].

The study of software structure has become popular since the early 1990s with research work about software architecture concentrating on architectural styles, description languages, documentation, and formal methods [22], and the recent work about software architecture evolution such as [38]. Software architecture can help assure that the system has certain properties and behaviors when it has been built. Describing structures as a set of views, each of which addresses different concerns, is now accepted as a standard architecture practice [108].

In this section we review some earlier approaches to software structure, the consistency checking between parts of a system and the correctness of an implementation. We classify these approaches in four categories.

1) Specialized languages to describe software structure

The concept of “programming in the large” and “programming in the small” was introduced in [26] and it was stated that we needed a module interconnection language for knitting software modules together and for providing an overview that formally records the intent of the programmers. This information can be used to check the consistency between modules by a compiler. There have been many proposals to introduce a separate language to describe the system structure ensuring the consistency among different parts of a system, and here we list a few of these languages.

MIL and ADLs
In 1976 [26] Deremer and Kron proposed the MIL (Module Interconnection Language) which provides grammar constructs for deciding the various module interconnection specifications required to assemble a complete software system. MIL was aimed to ensure consistency among different parts of a system. The idea was further developed in the work of [102, 93]. In MIL, each module provides/defines a set of facilities that are available to other modules and requires/uses facilities provided by other modules. This is called a definition/use relationship. The consistency checking indicates whether each use of a facility is provided by the facilities defined. The use of MILs presents a development and maintenance system for consistency checking among different modules. But MIL-based consistency checking is limited to simple type checking, there are no semantics for defining module functions.

In 1990s the MIL was extended by ADL (Architecture Description Languages) using communication protocols and constructs for the definition of system functional semantic properties [21]. Many ADLs have been proposed, for example ACME [35-37] and Wright [3, 4] at Carnegie Mellon University, Rapide [66] at Stanford, and many others. Researchers tried to overcome the lack of semantics in the languages and provide support for specifying both functional and non-functional characteristics of components. In ADLs architectural connectors are defined to describe the interactions between the components, and protocols are used as a mechanism for describing interactions. ADLs are intended to provide a basis for rigorous analysis of the system design, making possible the early detection of design errors and flaws and help ensure correctness [100]. While all ADLs provided built-in internal consistency and completeness rules for artifacts rendered in those languages, only a few allowed users to define what was meant by consistency, and only a few dealt with consistency between different artifacts [21].

**Extended ML**

EML (Extended Meta Language) is a language covering both specification and implementation [56]. It is a framework for specification and development of SML (Standard Meta Language, a functional programming language) [72] programs. EML specifications look like SML programs except that it permits axioms to be included in signatures and in module bodies. With this addition the language can be used for stepwise refinement proceeding gradually from an initial formal specification to yield
eventually an executable SML program. In programming terms, this corresponds to implementing a program module by decomposing it into a number of sub-modules. The end-product is an interconnected collection of generic SML modules, each with a specification of its interface with the rest of the system. Correctness of the final executable SML program with respect to the original specification can then be established by proving the correctness of each refinement step [56]. But there is the problem of over-specification since EML specifications are programs that include executable axioms.

In EML the word “module” means a structure or an implementation. A structure contains definitions of types, values, and exceptions and may also contain sub-structures [56]. It is different from the Parnas “information hiding module”. The definition about the interconnection among the modules and the relations between modules at different levels are not provided either.

**UML and Rational Unified Process**

Booch, Rumbaugh and Jacobson have developed UML (Unified Modelling Language), which includes a set of graphical notation techniques to create visual models of software systems. It is used to specify, visualize, construct and document the artifacts of an object-oriented software system to help the visualization of the architectural blueprints of a system, and facilities the understanding of the system [96]. The UML is a language to express various models, but it does not tell how to develop these models. As a necessary complement to UML, Kruchten [61] proposed the Rational Unified Process hand-in-hand with UML to provide a guide to the effective use of this language, and illustrated how to describe architecture through a set of design viewpoints and views and how to express these views in the UML. The rational unified process is a use-case-driven approach. The use cases describe what the system is supposed to do and serve as the basis for the development process. In the development process, use cases drive activities such as creation and validation of the design model, and help synchronize the content of various models [61].

In UML different views are produced for a system. These views should be properly related to each other in order to form a consistent description for the developed system.
The ambiguity inherent in UML poses a great risk of inconsistency among the views and within each view. The topic has attracted attention from the research community such as the work reported in the workshop on "Consistency Problems in UML-based Software Development” [52] and ASEA 2008 (see [103]); many proposals are made for the mitigation of the inconsistency problem in UML. The essence of these proposals is to clear the existing ambiguity and to seek the formalization of UML. Unfortunately most of the proposals are not implemented in UML CASE tools [34].

2) Model checking for program/document verification

This approach is also used in checking the completeness and consistency of individual requirements document, as described in Section 1.4.1. There is some work [57, 15, 19, 106] using it to check the correctness of design documents or programs.

In [57] Kemmerer proposed testing specifications to guarantee the consistency between specifications and a formal model. A formal model is a criterion expressing the critical requirements of the system that must hold in every state. It specifies the effect of performing each operation based on certain entry conditions being satisfied. Symbolic execution is used to check the validity of a specification. After each transformation is symbolically executed, a tool will check to see if the current state is consistent with the criterion. A problem with this work is that the predicates defining the current state can get unmanageable after executing a sequence of transformations, particularly when the transformations contain conditional expressions that cannot be readily resolved.

In [15] Chechik and Gannon presented an approach to check the consistency between SCR-style requirements and a detailed design written in PDL (Program Design Language) and the implementation of these ideas was realized in a prototype. An SCR specification describes the system as state machines with event-driven transitions, and the detailed design expressed in PDL is control flow based. The design is said to be consistent with its requirement if the design’s state transitions are enabled by the same events as those of the requirements and all the requirements state transitions appear in the design. This work gives a way of checking between requirement and design using logic formulas and state transitions, and uses an example for the analysis.
Clarke et al. [19] present an algorithm that checks behavioral consistency between an ANSI-C program and a circuit given in Verilog using Bounded Model Checking. Both the circuit and the program are translated into a formula that represents behavioral consistency. The formula is then checked using a SAT solver. They translate C programs that include side effects, pointers, dynamic memory allocation, and loops with conditions that cannot be evaluated statically. Experimental results are described on various reactive circuits and programs, including a small processor given in Verilog and its Instruction Set Architecture given in ANSI-C.

3) The relational approach for program/document verification

Both the relational approach [69, 84, 70, 90] and pre/post condition approach [48, 49] can be used for proving the correctness of programs with respect to their specifications. A detailed survey and comparison of these two approaches can be found in [31]. The relational approach used for program verification was described in [98, 41] and the correctness theorem of a program is defined in [71].

In [70], the relation of program usage hierarchies defined in [78, 79] are discussed and block diagrams are presented. Block diagram coalesce the uses of each data abstraction in the usage hierarchy into a single node and coalesce the usage relations among data abstractions into arcs among nodes. The “blocks” in the block diagram represent data abstractions or programs. The block diagrams summarize the system structure to aid general understanding of a system’s parts and data flow from the parts. If the sequential and concurrent usage of modules shown in block diagrams is left as programmer details for later expansions, the design process is inherently error-prone because the correctness of uses among modules cannot be immediately verified.

The work mentioned above is to check the correctness of programs, which has less relevance to our work that is intended to documentation checking. In this thesis we have used the method in [90], where the relational approach was generalized for the mathematical software documentation and verification.

In [90] Parnas and Madey proposed the contents of software documents and showed how to define the required contents of a document without talking about structure or notation. By describing what information should be contained in each document and how to
express this information as relations, the completeness and consistency of the documents can be assured, since the relationships between these relations/functions can be defined. The application of relational approach in documentation and the verification of those mathematical documents are proposed in this paper. For example, the correctness of an internal module design with respect to its specification can be verified by using an abstraction function that is an intrinsic part of the design documents. Given the data structure, an abstraction function, and a description of the function implemented by each of the sequential programs, it is possible to verify that the component will satisfy a functional specification. This verification can be based exclusively on the functional documentation and be performed before the component programs are implemented.

The recent work of Courtois for safety justification of computer based systems [24, 25] is based on the relational approach, where a four layered structure is proposed in the justification of the dependability claims. A dependability claim is either a kind of meta-requirement addressing a property that a system requirement specifications should have, for example the completeness and feasibility of the system, or an extra-requirement addressing needs that were not explicitly part of the original system requirement specifications. The relation that constructs the four-layered model is extension, and a claim at some level should be supported by evidence at the same level or lower level. In [24, 25], the dependability claims basically claim that the system specifications are valid and the system behavior is dependable.

4) Improve the way in building and documenting components and intermediate verification of each step of refinement by relational approach

A long-held and often stated view of Dijkstra, and many many others, is that the correct approach to software design would be trying to design in such a way that the designers can keep things under intellectual control and that there was no need to search experimentally for bugs after they are complete. Since the 1970s, Dijkstra's chief interest had been formal verification coupled with program derivation. He proposed that one should start with a mathematical specification and apply mathematical transformations to the specification until it is turned into a program that can be executed. The resulting program is then known to be correct by construction [29].
The approach to deriving designs from requirements proposed in [58] is a refinement of this idea. The authors defined what it means one relation is “part of” another relation. A relation Q is part of P then the subsystem having Q as its requirements is said to be an architectural component of S having P as its requirements. This approach allows the decomposition of a relation into relations that are "part of" the original relation. These parts are required to perform part of the function of the system, which makes them suitable for architecture design, with each part of the sub-relation constituting the specification of an architectural design component. To divide a relation into parts, the relation will be first converted to a partitioning graph and then using a graph clustering technique to obtain the architectural components. However, to apply this method to a concrete system, one must specify how the relations represent requirements and what the domain and the range of these relations are.

Nevertheless, no matter how a design is arrived at, i.e., even if it is derived by the aforementioned method, the consistency of these components and the correctness of a design need to be checked. Most experienced professionals know that a program has to be tested no matter what method was used to derive it. Human error is always with us.

Starting from the work on information hiding in the early 1970s, the precise mathematical documentation and intermediate verification [77, 82, 74, 86, 90, 91], software structure [78-80] to the new trace function method in 2006 [92], Parnas has developed a set of systematic principles, some of which were collected in the book by Hoffman and Weiss [50], describing much of the foundation of software engineering. These principles have been applied to each stage of software development in industry as well as applied to many language designs such as C++, Java and SequenceL [23]. From the aforementioned work one can conclude that quality cannot be achieved merely through verifications and testing strategies or tools, although they are necessary; design and documentation are critical; only well designed and documented software can be verified logically.

To deal with the complexity of software systems and to improve the quality of them, four principles should be applied, as concluded in [65]:

- **Divide and conquer**
A large or complex task must be reduced to a set of smaller simpler tasks. The solutions to the sub-problems are then combined to give a solution to the original problem. The software system decomposition and composition is the example application of this principle.

- **Separation of concerns and information hiding**
  It must be possible to carry out each of the smaller tasks independently of the others, i.e. without knowing how problems that arise in completing the other tasks are being solved. Separation of concerns [27, 30] is the process of separating a computer system into distinct tasks such that one can focus attention upon some aspects at a time. Progress towards separation of concerns is achieved with the help of information hiding [76] that makes interfaces markedly simpler, clearer and less likely to change.

- **Refinement**
  When a task is decomposed into smaller tasks, everything to be done as part of the original task must be included in one, and only one, of the subtasks. The function implemented by a constructed program can be computed from the black-box descriptions of the functions implemented by its sub-programs without looking at descriptions of how those sub-programs are implemented. This is nicely illustrated in [69, 62].

- **Stepwise verification**
  Each refinement must be checked to confirm that correct completion of all of the subtasks assures correct completion of the original task. To minimize rework, the correctness of the refinement should be verified before the work on the smaller tasks begins.

Precise specifications of the tasks are essential in this process; without them, separation of concerns and stepwise verification are impossible. To fully benefit from these principles, each specification must be significantly simpler than the implementations that satisfy it.
1.6 The Scope of the Thesis

In this thesis we will follow approach 4) introduced in Section 1.5.2 to study the compatibility of a set of components and the conformance between the design documents and its black-box requirements.

The components produced by the development procedure cannot be independent; they must communicate and work together harmoniously. The way that the system components should communicate must be precisely described after the components are identified and specified.

It is helpful if the same component description notation and method of checking refinements can be used throughout the development process. This thesis presents a way to achieve this. The approach proposed in this thesis can be used to check the consistency and correctness of a set of design documents if the documents are provided as follows:

- The black-box requirements of a system are specified by a method describing the relations between input/output history and the output values;
- The system is decomposed with the guidance of information hiding;
- The interface of each component is precisely documented by the same method that specifies the black-box requirements.

To know whether the set of components can work together properly, we must know how they communicate with each other and what the structure of these components is. We present a way of describing the connections of the components. Based on the interface TFM specifications/descriptions and the network description, we define when the network is completely connected and consistent, and establish the conformance relation to verify the correctness of a design. Verification of each refinement requires being able to derive a description of the behaviour of a product from the descriptions of its components and their interconnection. This derived description must be compared with the specification of the product to confirm that its behaviour will be satisfactory.

In this thesis, we present an approach to a) describing a network of components, b) checking whether the network is completely connected and whether it is consistent, i.e., whether the components can work together properly, c) determining the behaviour of the
network, i.e., what the components will do if they are combined together as described in the network description, and d) checking whether the behavior of the network conforms to the requirements, i.e., if the network behaves as required.

We have established the derivation relation between the internal behavioural description of a network and the behavioural description of the constitute components, and the conformance relation for checking the correctness of a design. A procedure for deriving the behaviour of a network and checking the correctness of a set of fully specified components on the basis of their interface descriptions and a description of the way in which the components are connected is presented. Some techniques on how to simplify the derived trace function expressions are also developed. Three case studies are presented to illustrate how the approach works.

In this approach, each component is viewed as a hardware-like device in which an output value can change instantaneously when input values change and all components operate synchronously rather than in sequence. The connections among components are through shared variables. The behaviour of a component is described by TFM but it is also compatible with other method of providing component interface descriptions by documenting the relation between input/output histories and output values, the derivation of the behaviour of the component network uses standard relation composition, and the conformance relation is defined using classical mathematical concepts and logic introduced in [87, 98, 90, 41, 11].

1.7 The Structure of the Thesis

The remainder of this thesis is organized as follows.

Chapter 2 gives some basic definitions about component, interface, TFM interface specifications/descriptions and the related concepts to be used throughout this thesis. Chapter 3 defines the network of components and explains the purpose of describing the component interconnection information. The completeness and consistency of a network are discussed, including the concepts of value data-type consistency, delay-free loop free consistency, and the reason to exclude the delay-free-loops in our analysis.

Chapter 4 establishes the relation for deriving the TFM descriptions of a network of two components; proves commutability and associability of the operatoration and extend the
relation to a network of more than two components; describes the deriving procedure of a TFM description of a network from the interface documents of the constituting components and the network description by using a simple example.

Chapter 5 proposed the conformance relation for checking the correctness of a set of design documents. In this chapter, the conformance formula between the derived TFM descriptions of the network and the system requirement specifications are established and two commuting diagrams are provided.

Chapter 6 provides three case studies to show the complete procedure of:

- Specifying a system requirements by TFM;
- Decomposing the system into components using information hiding principle;
- Documenting the component interface TFM descriptions;
- Describing the interconnections of the components;
- Checking completeness and consistency of the component network;
- Determining the behavior of a network of components according to the derivation relation;
- Comparing and checking whether the behavior of the network of components satisfies the system black-box requirements according to the conformance relation.

The main contents, contributions and limitations of this thesis are summarized in Chapter 7. Some possible future research questions are also pointed out in this chapter.
Chapter 2
Component, Interface and Interface Documentation

In component-based software development, components are the constituent parts of a system. Components communicate with each other through their interfaces. In this chapter we define the basic concepts about component, interface, interface documentation and the terms used in interface documentaries.

2.1 Component

Definition 2.1: software component [92]

A software component is a collection of programs intended to be used as a complete unit.

It is important to note that a component is usually a set of programs, not just a single program. A well-designed component should have an interface that is simple, clearly specified, and sufficient for its intended use. Usually each component performs a certain function or a set of related functions and will communicate with other components to make a system.

We view a component as a hardware-like device in which an output value may change instantaneously when input values change. An example of this is the stack shown schematically in Fig. 2.1. We view it as a device with two inputs and two outputs. One input provides a command; it has values from the set \{Push, Pop, none\}. The other input variable can contain a value to be pushed on the stack.

![Fig. 2.1: A Stack as a Component](image)

The two outputs provide the current depth of the stack and the value on top of the stack. We have come to think of a stack as a software mechanism, but it should be noted that in 1957 Bauer and Samelson applied for a patent on a design that included 2 “Keller” or
stacks. This can be seen in [12]. The methods used in this thesis can be used with either hardware or software implementations of a component.

2.2 Interface

**Definition 2.2: component interface [92, 65]**

Given two specified components $A$ and $B$, $B$’s interface to $A$ is the weakest assumption about $B$ that is sufficient to prove the correctness of $A$.

Any assumption that must be made to prove the correctness of another program is part of the interface. For example$^1$, if you have to assume that memory address 20223 is accessed every 40 ms, to prove that a cooperating program works, the fact that the address 20223 appears on the memory bus is part of the interface. If you have to assume that the code is written in purple ink, that fact is part of the interface. Such assumptions may not be necessary with well-designed interfaces (for which the criteria such as information hiding should be used) but the definition applies to all kinds of interfaces.

An interface is generally an abstraction of the component’s implementation. It is the best if the interface abstracts from implementation details that are most likely to change. If that is done, the interface will be less likely to change.

Component $A$’s interface to component $B$ is usually different from $B$’s interface to $A$.

While in general, interfaces are properties of pairs of components, it is often useful to define a public interface for a component. The public interface may be viewed as comprising two parts [92, 65]:

- The **upper public interface** of a component is the set of assumptions that users of the component may make about its behaviour.
- The **lower public interface** is the set of assumptions that the component designers made about the components that this component uses.

A complete description of $B$’s interface to $A$ tells us exactly what changes in $B$ will affect $A$. If the public interface of $B$ is different from $B$’s interface to $A$, the public interface should contain less information. If $B$'s interface to $A$ is not implied by $B$'s public interface,

---

$^1$ The examples (the definition too) are from David Parnas, it may seem strange but it is real.
there will be changes in $B$ that do not violate $B$'s public interface but do make it necessary to change $A$. This bad practice is all too common.

The interface of a component separates the methods of external communication from internal operation, and allows the component to be internally modified without affecting the way outside components interact with it, as well as provide multiple abstractions of the component.

### 2.3 Precise Documentation of Component Interfaces

We use the TFM [6, 92, 95] to document interfaces of components; each output value is constrained by a relation between the values of that output variable and a trace describing the input/output history of the component.

Throughout this thesis the concepts and terminology used in [92] will be adopted. Note that this section is not a full introduction to TFM interface documentations. For complete and exact understanding on the TFM, readers are referred to [92]. This section presents the definitions of some basic terms and concepts of TFM, which can be found or implied in [92]. These definitions will be used in defining the consistency rules and the conformance relations as well as the extensions in the context of a network. To define the conformance formally, we first need the formal definitions of a TFM specification/description of the behaviors of a component and all the corresponding terms and terminologies.

#### 2.3.1 Trace, Event and Event Descriptor

In the TFM [92], component history is represented by a sequence of event descriptors, called a trace. Each event is a change in the value of input and output variables; an event descriptor describes the value of those variables before and after the event.

The event descriptors represent sets of triples. There is one triple for each relevant variable. Each triple contains

- a unique name for the variable,
- the value of that variable before the event, and
- the value of that variable after the event.
Where time is important, it can be viewed as a global variable and requires no special
treatment.

For the convenience of set operations in the calculations later in this thesis, denote \( e \) as a
set of triples \((v, p, q)\), where \( v \) is the unique name of the variable, \( p \) and \( q \) are the values
of the variable before and after the event.

**Definition 2.3: syntax of an event descriptor**

For a component that has a set of input and output variables \( V \), an event descriptor \( e \)
for that component can be represented as:

\[
e = \{(v_1, p_1, q_1), (v_2, p_2, q_2), \ldots, (v_n, p_n, q_n)\}
\]

where \( n = |V| \wedge v_i = 1, 2, \ldots, n \in V \wedge p_i, q_i \in \text{Type}(v_i) \wedge \exists v \in V \text{ s.t. } ((v, p, q) \in e \wedge p = q) \). Here time \( t \) is considered as one of the input
variables in the data structure and \( p = q \) always holds for the triple \((t, p, q)\), which
means that the value of \( t \) is assumed unchanged in an event.

The before/after values of a variable in an event are expressed by a prime before/after the
variable name in TFM. To make the convention formal and for convenient usage, we
give two function definitions as follows:

**Definition 2.4: before value and after value of variable \( v \)**

For an event \( e \), \( \nu(e) \) returns the value of variable \( v \) immediately before \( e \), and \( \nu'(e) \)
returns the value of \( v \) after \( e \). According to Def. 2.3, the functions can be written as:

\[
\nu(e) = \begin{cases} 
  p & \text{if } (v, p, q) \in e \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

\[
\nu'(e) = \begin{cases} 
  q & \text{if } (v, p, q) \in e \\
  \text{undefined} & \text{otherwise}
\end{cases}
\]

**Remark 2.1:** If the value of a variable \( v \) is not changed when an event occurs, the
variable name itself can be used as a function name. For example \( \nu(e) \) can be used to
express the value of \( v \) for event \( e \), if there is no value change for variable \( v \) in that event
\( e \). Even though we still use a triple \((v, p, q)\) where \( p = q \) to represent that variable in an
event descriptor.

As stated in Def. 2.3, \( t \) is assumed unchanged in an event, we always use \( t(e) \) to express
the value of the time when event \( e \) happens.
For variables that change continuously such as time or temperature, we are not interested in all events (there would be an infinite set of them) but in a sample. The trace only contains descriptions of that sample of the events, and we are assuming that values between events are not relevant.

**Definition 2.5: syntax of a trace**

A trace $T$ for a component in TFM is a sequence of event descriptors $\{e_1, e_2, \ldots, e_n\}$ ordered by time $t(e_1) < t(e_2) < \ldots < t(e_n)$.

**Remark 2.2:** Note that we use “$<$” instead of “$\leq$”, which means that if any variables change values at the same time, that is considered to be one event. This is obvious and easy to understand in a single component. However, in the scope of a set of interconnected components, two component event descriptors in two different components can have the same time value. In such a case, the two component event descriptors will be united to one event. Component event descriptors describe state changes within components, and they are not called events individually when talking about a set of inter-related components. More detail about these concepts will be presented in Section 4.1.

From Def. 2.3 it seems that any value of type $v_i$ can appear in the triple as $p_i$ and $q_i$. However a well-defined component would include trace functions\(^2\) (see the definition in [92]) describing the behavior of the component. Any meaningful and possible trace is constructed according to these trace functions. The before and after values of a variable in an event is not arbitrary, the after value of an output variable in an event will be calculated according to the defined trace function, while the before value of a variable in an event $e$ is equal to the after value of the same variable in the event immediately before event $e$ in a trace, and so on. Except the before values of the variables in the first event, all the other before/after values of the variables in any other events are precisely calculated by the trace functions.

In short, the Def. 2.3 and Def. 2.5 are just to describe the format of an event descriptor and a trace. The semantics of them will be described later in this section.

---

\(^2\) In general these functions should be relations if the non-deterministic case is included. For simplification in describing the values of a variable here we assume determinism.
2.3.2 The TFM Specifications/Descriptions of a Component

For a non-deterministic component, the value of each output variable is constrained by a relation between the output values and the history of the values of the component input and output variables. For deterministic components, the output values will be a function of the history of the input values.

The traces in the domain of these output relations begin with the first event and include the current event; consequently, expressions describing the value of an output may refer to the value being computed and to other values at the same time.

**Definition 2.6:** component interface TFM specification/description

A Component Interface TFM specification/description (CIT) from [92] can be denoted by a quadruple \((I, O, F)\) where:

- \(I\): the set of input variables of the component,
- \(O\): the set of output variables of the component,
- \(F\): an indexed set of trace relations (functions if the behaviors of the component are deterministic) for computing each output variable. We can write \(F = (f_o)_{o \in O}\).

**Remark 2.3:** We use \(T\) to denote the set of possible traces of the component. It is also the domain of the functions in \(F\).

The domain of every function is the set of possible traces of the component, and the range of the function is a set of values of a certain type for an output variable in \(O\).

 Normally a member of \(I\) is denoted as \(i\), a member of \(O\) as \(o\), and a member of \(T\) as \(T\).

2.4 The Set of Possible Traces of a Component

In a TFM specification/description documentation \(CIT = (I, O, F)\), the sets of \(I\), \(O\) and \(F\) are explicitly given. The set of possible traces \(T\) can be derived from the three sets in the definition. Since traces are always used in the sequel, here we introduce one way to construct the set \(T\). This describes the semantics of traces and event descriptors.

The set of possible traces \(T\) for a component can be inductively defined as:

1. The empty trace, denoted as “\(\_\)”, is in \(T\).
2. If a trace \(T\) is in \(T\), and if an \(e\) is an event descriptor of this component that can follow \(T\), then \(T.e\) is in \(T\).
3. Nothing else is in \(T\).
The following rules determine whether or not an event descriptor described in Def. 2.3 is an event descriptor of a component that can follow a possible trace \( T \). If the following properties of a trace are satisfied, we say \( e \) can immediately follow the possible trace \( T \) to form a new possible trace \( T.e \):

- **Time Condition**: \( \text{TC}(T.e) \equiv T \neq \_ \Rightarrow t(r(T)) < t(e) \)
- **Consistency Condition**: \( \text{CC}(T.e) \equiv T \neq \_ \Rightarrow \forall o \in O, o'(r(T)) = 'o(e) \)
- **Initialization Condition**: \( \text{IC}(T.e) \equiv \forall o \in O, (\_, 'o(e)) \in f_o \)
- **Output Condition**: \( \text{OC}(T.e) \equiv \forall o \in O, (T.e, o'(e)) \in f_o \)

No other conditions are needed to determine that \( e \) can follow \( T \).

where \( r(T) \) is the latest event in \( T \), defined in [92].

In general, we suppose that the documents are relations, which includes the cases when the documents are functions. If the behavior being documented is deterministic, the relation \( f_o \) for variable \( o \) will be a function thus \( \text{IC} \) becomes \( \forall o \in O, 'o(e) = f_o(\_ ) \) and \( \text{OC} \) is \( \forall o \in O, o'(e) = f_o(T.e) \).

Each condition describes a property of traces stated or implied in the TFM documentation, as can be found in [92]:

- **TC**: together with the induction it says that in every trace the sequence of event descriptors is ordered by time in ascending order. Note that this is only needed when we consider time as a global variable. If time is not among the variables this would make no sense.
- **CC**: together with the induction it says that the before value of any output variable at an event \( e \) equals to the after value of the same variable at the event immediately before \( e \).
- **IC**: It says the before value of an output variable in the very first event of any trace are determined by the value of an empty trace defined in the trace relation for that variable. Condition IC can be so understood that the output values for an empty trace are stored as the before values in the first event.
- **OC**: together with the induction it says that a trace relation \( f_o \) of variable \( o \) determines the after value of \( o \) in each event. OC and CC jointly say that both the
before and after values of a variable are constrained by its trace relation. This condition describes the most essential part in TFM documentation.

Using the four conditions, the set of possible traces in a component can be defined as the following:

**Definition 2.7: set of possible traces \( T \) for a component**

For a component with a TFM specification \((I, O, F)\), the set of possible traces \( T \) is defined inductively as:

1. \( \_ \in T \)
2. \( \forall T, e, T \in T \land AC(T.e) \Rightarrow T.e \in T \)
3. \( \forall T, e, T.e \in T \Rightarrow T.e \in T \land AC(T.e) \)

where \( AC(T.e) \equiv TC(T.e) \land CC(T.e) \land IC(T.e) \land OC(T.e) \) means “All Conditions”.

If the characteristic predicate of the set \( T \) is \( P(T) \), then

\[
P(T) \Leftrightarrow (T=\_) \lor (P(p(T)) \land AC(T)) \equiv (T=\_) \lor (P(p(T)) \land TC(T) \land CC(T) \land IC(T) \land OC(T))
\]

where

- \( TC(T) \equiv p(T) \neq \_ \Rightarrow t(r(p(T))) < t(r(T)) \)
- \( CC(T) \equiv p(T) \neq \_ \Rightarrow \forall o \in O, o'(r(p(T))) = 'o(r(T)) \)
- \( IC(T) \equiv T \neq \_ \Rightarrow \forall o \in O, (\_, 'o(o(T))) \in f_o \)
- \( OC(T) \equiv T \neq \_ \Rightarrow \forall o \in O, (T, o'(r(T))) \in f_o \)

### 2.5 The Descriptors at any Time Instants in a Trace

Motivated by the situation described in Section 4.2.1, we define another descriptor to describe states in between component state changes for the convenience of domain extension that will be presented in Chapter 4.

**Definition 2.8: syntax of a component dummy event descriptor**

For a component that has a set \( V \) of input and output variables, and an event \( e \) in this component, the operation \( e' \) is defined as follows:

\[
e' = \{(v_1, q_1, q_1), (v_2, q_2, q_2), \ldots, (v_n, q_n, q_n)\} \text{ where } n=|V| \land v_i=1,2,\ldots,n \in V \land q_i = v_i'(e).
\]

It can be seen that \( e' \) contains the redundant after values of each variable from \( e \). It describes the state after \( e \) and before the next state change immediately after \( e \). Although
e’ contains redundant after values within a component, these values are not necessarily redundant within a system of more components.

As stated in Remark 2.2, if time $t$ is considered as a input variable, $t(e)$ can be used to return the value of the time when $e$ occurs. We may define a function to pick out a component event descriptor from a trace $T$ at any time $t$, if $t$ happens to be the instant when there is a state change. At any time instant between two adjacent component state changes, the before and after values of all variables remain the same and can be described by a dummy event descriptor $e’$. We use a uniformed descriptor to express the event/dummy event descriptor as the following:

**Definition 2.9: descriptor at any time $t_0$**

For any trace $T = \{e_1, e_2, \ldots, e_n\}$, $t(e_1) < t(e_2) < \ldots < t(e_n)$, we define $E(T)(t_0)$ as:

$$E(T)(t_0) = \begin{cases} 
\text{unspecified} & \text{if } \neg \exists i \text{ s.t. } t_0 > t(e_i) \\
e_i & \text{if } \exists i, 1 \leq i \leq n, \text{ s.t. } t_0 = t(e_i) \\
e'_i & \text{if } \exists i, 1 \leq i < n, \text{ s.t. } t(e_i) < t_0 < t(e_{i+1}) \\
e'_n & \text{if } t_0 > t(e_n)
\end{cases}$$

$E(T)(t_0)$ is a set of triples in the form of a full event descriptor, either describing a state change, or describing a state after a state change. The variables in the descriptor before the very first event should get initial values of the variables. These values are not really interesting and the assigned value “unspecified” is just to make the function $E(T)(t_0)$ total.

**2.6 The Domain Extension for Trace Functions**

Domain extension is a formal mathematical operation. For example the domain of the function $f(x, y) = x - y$ can be a set of pairs of values of $x$ and $y$, and it can also be viewed as a function with domain $(x, y, z)$. The domain of a component trace function is the history values of input/output variables of this component and it can be extended to the history values of variables that are not in this component. This will be discussed in detail in Chapter 4.
Chapter 3

Networks of Components and their Consistency

Among the software development activities introduced in Section 1.1, the interconnection description of the components or the software structure is the key information for verifying the design and for assembling the components into a system.

In this chapter, we will discuss why and how to describe this interconnection information among components and how to verify whether this set of components can fit together.

3.1 The Purpose of Describing a Network of Components

After a system has been decomposed and all the constructing components have been identified and specified, the design documents should be checked, before starting further implementation, to confirm that the components can fit together and that the resulting system will behave as required, if each of these components is correctly implemented. Describing a network of components is necessary to achieve this goal. In the following we explain the necessity by listing for which points this description is beneficial. These points are also the main research questions we will consider in this thesis.

3.1.1 Checking Component Compatibility

When components are assembled to perform a specified task, there will be both explicit and implicit connections between them. The explicit connections include the invocations and procedure parameters; designers are usually conscious of these. There may also be explicitly shared variables that are used for communications. However, there may be shared resources and variables that are part of the communications but overlooked by the designers and reviewers. It is essential for a valid analysis of the network that all communication channels are identified and included in the documentation. By performing some certain checks on these clearly documented connections, it can be seen whether or not these components can fit together and be combined to a well-defined bigger component.
3.1.2 Combining Components to a Constructed Component

Connecting a group of components together creates a constructed component. The original components are called the internal components of the constructed component. The interconnection of the internal components is not described in the constructed component interface documentation. From the outside, the constructed component is exactly the same as a simple component. However, the interconnection information inside is essential to the construction of the constructed component. Only through this interconnection information can we know whether or not the set of components can be combined together to construct a bigger component and what this constructed component will perform. The combination of components will be formally defined in Chapter 4. Here we introduce this concept just as a prelude to explain why we need to define a network of components.

3.1.3 Checking the Correctness of a Design

To check whether or not these components together satisfy the system requirement, we need to derive the behavioral description of an integrated system. For this purpose the interconnection and the composition of components must be clearly defined. A component network description, defined in the next section of this chapter, is used to describe how the components communicate with each other internally and how the components within a network communicate with outside of the constructed software system.

A component network description is part of a design. Since components communicate only through their interfaces, how the components communicate with each other should be designed and documented after the component interfaces are specified, so that the design documents can be verified before any implementation is conducted. A component network description contains necessary information for completeness and consistency checking among the components and conformance verification between a set of design documents and the requirement specifications.

3.2 Describing Networks of Components

If component \( A \) receives values from an output variable of component \( B \) as an input of \( A \), we view it as connecting the output line of \( B \) and the input line of \( A \). The communication
paths are a set of shared variables. Since components will connect to each other through their interface inputs/outputs, if one component sends one of its outputs to an input of another component, we consider one component to be writing to a global variable and the other to be reading the same variable. The values of the connected input and output variables are always equal without delay. We call the set of connected components a network of components or a component network. This is a generalization of the concept of network of processes in [75].

In our approach, all communications among components are via shared variables. One component writes to a variable and other components read that variable. Higher level communication mechanisms such as program calls and message transfer are always implemented using shared variables.

**Definition 3.1: network of components**

A network of components (NoC) is described by a quadruple \((\mathcal{A}, \Sigma, \Omega, \mathcal{R})\) where:

- \(\mathcal{A}\) is a set of components with input and output variables.
- \(\Sigma\) is the set of network input variables whose values are determined by the environment outside the network.
- \(\Omega\) is the set of output variables of the network. The variables are visible outside the network and their values are determined by the network.
- \(\mathcal{R}\) is a set of rules describing the connection of input and output variables in the network.

**Remark 3.1:** we use \(\Gamma\) to represent the set of interconnector variables whose values are determined within the network.

We denote a member of \(\Sigma\) by \(\sigma\), a member of \(\Gamma\) by \(\gamma\), and a member of \(\Omega\) by \(\omega\), an input to a component by \(i\) and an output of a component by \(o\).

The five kinds of network rules in \(\mathcal{R}\) are explained as follows. The following notations are used to represent the network rules:

- \((o \rightarrow \gamma)\) indicates that the value of the interconnector variable \(\gamma\) is determined by component output variable \(o\)
- \((\gamma \rightarrow i)\) indicates that the value of the component input \(i\) is determined by the interconnector variable \(\gamma\)
• \( (\sigma \rightarrow i) \) indicates that the value of component input \( i \) is determined by the environment of the network.

• \( (o \rightarrow \omega) \) indicates that the value of network output \( \omega \) is determined by the component output \( o \).

• \( (\gamma \rightarrow \omega) \) indicates that the value of network output \( \omega \) is determined by the interconnector variable \( \gamma \).

• For convenience we may write \( (o \rightarrow \gamma \rightarrow i) \) if both \( (o \rightarrow \gamma) \in \mathcal{R} \) and \( (\gamma \rightarrow i) \in \mathcal{R} \). Where the identity of the interconnector is irrelevant we may also write \( (o \rightarrow i) \).

Within a network each variable should have one unique name. Strictly speaking, the interconnector variables \( \gamma \) are superfluous, but we prefer to include the name of these shared variables that are used for communication in the documentation.

From the documentation point of view, the interconnector variable is a place to mention the network role of a shared variable. The role of a shared variable for the writing component, reading component, and the network interconnector might be different. For example, consider a distance sensor component with an output variable named “distance”. If this sensor is mounted pointing upwards in a network, the interconnector used for its output can be named “headroom”; and if it is mounted downwards, the interconnector may be called “depth”. For a component that reads the variable within a network, the name and meaning of the input variable might be quite different. For example, a component might perform a generic task such as averaging or predicting and is not restricted to distances. The input variable of this component might be something like “raw data” and the output could be “average” or “next possible value”.

Some variables in \( \Omega \) and \( \Gamma \) may be overlapped, i.e. a component output can be sent to the other components as inputs and meanwhile sent to the environment as an output of the network.

A network is static; it is an abstraction of how the inputs and outputs of the components are connected and how the network inputs/outputs and the components inputs/outputs are associated. The connections do not change over time and are delay-free such that the values of connected input and output variables are always equal and can only change
simultaneously; there is no delay between the connected output and input variables. Delay in the system occurs only inside a component.

In summary an NoC is a collection of components that are connected through their interfaces to perform a required task. A node in an NoC is viewed as a hardware-like black box with input/output variables. The values of the output variables at a particular time are constrained by the history values of the inputs and outputs up to that time. These components can function in real-time. The output value can be immediately affected by a change in the input and all components are functioning simultaneously. An output of one component can be connected to many inputs of other components, but each input of one component can only be connected to one output of another component. This view is similar to the standard block diagrams in mechanical engineering but here we applied it to software system.

3.3 Other Approaches

As introduced in Chapter 1, terms such as “module interconnection”, “software structure”, and “software architecture” and so on are used in literature to describe the connection, dependency or communication among components of a software system. Although the various use of these terms is often considered to be closely related, close examination of the use of the terms shows that they are different. The problem of this sort was raised and clarified in [78] by examining the different “hierarchical structure” of several operating systems. Similarly, here we will take a close look at some of the approaches to diagrams about connections among parts within a software system.

As can be seen from the numerous literatures, the need for architecture diagrams in a software system has been well recognized. There are many such diagrams with different meanings for different purposes. For example, Kruchten proposed the so called 4+1 view model in [60], and stated that architecture can hardly be described entirely by one diagram and there are many views of a system. The important thing is to make clear the definition of these diagrams and what can be done based on them.

In this section we review a few of such diagrams to see their difference and similarity to our approach. In the order of increasing similarity to our approach, these structure
diagrams are component diagrams [96], architectural description diagrams in [68], data flow diagrams [101], and function block diagrams [53].

3.3.1 Component Diagrams in UML
A component diagram in UML displays the components in a system and the dependencies among the components [96]. In this diagram a component is a physical piece of code or a physical unit, such as a source code file, a runtime file, script, database, etc. There are runtime components, development components, and business components [61]. The connecting lines between two components are called “dependencies” and sometimes are stated as “uses”.

It can be seen that both the components and connections are different from our definitions. The lines that connect two components are not inputs/outputs but “dependencies” whose exact definition is not given. Although a component diagram also intends to show that a bigger component is constructed by the internal components in the diagram, this diagram alone cannot be used as a basis to check the compatibility of the components, to show how to combine the components and how to determine the behaviour of the constructed component. In our approach a component is a runtime concept and the purpose of describing a network of components is to check the compatibility of the components and to derive a description of the behaviour of the constructed component from the descriptions of the behaviours of the constructing components.

3.3.2 Architecture Description Diagrams in ADL
As mentioned in Chapter 1, architectural connectors are defined in ADLs to describe the interactions among the components. ADLs are usually declarative and describe the system architecture as a set of components, connectors, and configurations of these elements. Here we look at the architecture description diagram in Koala [68], among the many ADLs.

In this diagram, the boxes with arrows represent interfaces defined by sets of function calls. If the arrow points into a component, then the component provides or implements that interface; if it points out of the box, then the component requires access to the interface. The lines or connectors represent connections between required and provided
interfaces and represent the runtime function-call paths. Connectors in other ADLs represent more general connector semantics that can encompass streams, events, and message-passing protocols [68].

The connection in this diagram is more or less like the component diagram in UML. The connecting lines between components represent connections between required and provided interfaces, which are “dependencies” between two components. The connections do not represent data exchange as in our approach, the compatibility between components such as data type consistency cannot be checked, and the behavioral description of the constructed component cannot be deducted based on this view.

3.3.3 Data Flow Diagrams

A DFD (data-flow diagram) is a graphical representation of the flow of data through an information system [101, 111]. It is designed to show how a system (program) is divided into smaller parts (programs) and to highlight the flow of data between those parts (programs). It shows the flow of data from external entities into the system and output from the system, the sequence in which the data is processed and transformed, and sometimes the format of the intermediate storage. A survey of formal representation techniques for DFD is presented in [54].

The common characteristics of DFD and NoC is that in both diagrams, the connecting lines between different components show how the data is sent from the output of one component to the input of another.

In a DFD the nodes are processes and data stores, the connections are sequential data flows. Usually these processes are subroutines that do not run simultaneously. In our model all components are supposed to be functioning concurrently and synchronously.

3.3.4 Function Block Diagrams

An FBD (Function Block Diagram) is constructed using function blocks that are connected together to define the data exchange [53]. FBD is supported by the IEC 61131-3 standard [1] - the International Electrotechnical Commission standard governing the concepts and programming languages for industrial control systems. The primary concept behind a FBD is data flow; and the data flow from the inputs to the
outputs, through a function block. A function block diagram is described as a set of elementary blocks. Input and output variables are connected to blocks by connection lines. An output of a block may also be connected to an input of another block. The data at both ends of one connection line must be of the same type. Readers may note the natural similarity between this diagram and the block diagram for control systems. The FB concept from the original IEC 61131-3 standard was developed to a new version in the later IEC 61499 standard [2], in which the event flow (besides the data flow) is added. The similarity with our component network lies that a node in this diagram describes a function block (a set of functions in our approach); the connecting lines are inputs and outputs, and the data type consistency is discussed in both approaches. But we do not distinguish data flow and event flow, event flow in our model is just a special case of data exchange, for example the program call is viewed as connection between global variables, as discussed in Section 6.2.4.

3.4 Completeness of a Network of Components

Within a network a component will accomplish what it should provide only when its assumptions are satisfied by other components. A network is complete if the assumptions of each component are satisfied by either other components inside the network or the network inputs from outside of the network.

We consider a network to be completely connected if both of the following conditions are met:

- All component input variables are connected either to one of the output variables of other internal components or to one of the system input variables.
- Each network output variable is connected to one and only one component output or to a network input.

**Definition 3.2: completely connected**

A network of components, $NoC = (\Delta, \Sigma, \Omega, \mathcal{H})$, is said to be completely connected if and only if:

- For each component input $i$, $\mathcal{H}$ contains either exactly one rule $\gamma \rightarrow i$ or exactly one rule $\sigma \rightarrow i$.
- For each interconnector $\gamma$, there is one and only one rule $\sigma \rightarrow \gamma$ in $\mathcal{H}$.
• For each network output $\omega$, there is one and only one rule ($o \rightarrow \omega$) in $\mathcal{H}$.
• There are no other rules in $\mathcal{H}$.

**Remark 3.2:** From Def. 3.2 it can be seen that one component output $o$ can be the input of 0, 1 or more components and an output of the network. See the following figure. And one component input can be connected with one and only one interconnector variable or a system input variable.

The first condition guarantees that all the required inputs of a component should be provided by its environment and by a single source. By this rule two kinds of problems can be detected: 1) there are inputs that are not connected to any other variables; and 2) there might be an input that is connected by more output variables. We consider both cases as an incomplete network. Case 1) usually results from an error that either this input is not necessary or the design is incorrect. If case 2) happens, we may add a multiplexer component of which all the outputs mentioned above are the inputs, and an appropriate output is produced and then connected to that intended input variable.

The second condition indicates that the value of $\gamma$ is determined by the value of $o$. The third condition requires that each network output must be calculated by exactly one component as a component output.

If two components read from the shared input variable, the input variable of each component will use different name (as in Fig 3.1 shown) and will be connected to the same variable, $\gamma$, for instance. This is not applicable to output variables since the network completeness restrictions (the second and third conditions) do not allow shared output variables.

### 3.5 Consistency of a Network of Components

Consistency means the ability of the components to work together without contradiction and leading to defined behavior. When components are combined together to form a
system, the data types of the interconnected variables should be compatible and the constructed component should have well-defined behavior.

3.5.1 Data Type Consistency

Data types in each network rule should be compatible. If variable $y$ will receive values from variable $x$, the set of possible values for variable $x$ must be a subset of the set of possible values for variable $y$. We use the concepts of data types and type consistency as defined in [81] but have updated the terminology to be closer to recent literature.

Definition 3.3: type

Types are sets of variables with specified properties. The specification of the type is the characteristic predicate of the set of variables.

This is intentionally a much more general definition than is usual. See the reason of such definition in [81]. Note that a variable may be a member of more than one type.

Definition 3.4: subtype

Given two types, $T_1$ and $T_2$, $T_2$ is a subtype of $T_1$ if every variable in $T_2$ is an element of $T_1$.

Definition 3.5: value-type

A type is a value-type if the characteristic predicate of that type is defined only in terms of the values that the variables of that type can have. For example, if $A$ is in value type $T$ and $B$ has the same set of possible values as $A$, then $B$ is also in type $T$.

Definition 3.6: value-subtype

Given two value types $T_1$ and $T_2$, $T_2$ is a value-subtype of $T_1$ if every possible value of a variable in $T_2$ is a possible value of a variable in $T_1$. If we use a function $\text{val}$ to return the set of possible values of a type $T$, then we may say $T_2$ is a value-subtype of $T_1$ if and only if $\text{val}(T_2) \subseteq \text{val}(T_1)$.

Note that $T_2$ being a value-subtype of $T_1$ does not imply that it is a subtype of $T_1$. These relations are reflexive and transitive.

Definition 3.7: data type consistency
The data types of the variables in a network are consistent if and only if:

- For each \((o \rightarrow \gamma \rightarrow i) \in \mathcal{R}\), the type of \(o\) is a value-subtype of both the type of \(\gamma\) and the type of \(i\).
- For each \((\sigma \rightarrow i) \in \mathcal{R}\), the type of \(\sigma\) is a value-subtype of the type of \(i\).
- For each \((o \rightarrow \omega) \in \mathcal{R}\), the type of \(o\) is a value-subtype of the type of \(\omega\).

### 3.5.2 Delays in a Component

Since the interconnection among components is delay-free, any necessary delay will be designed inside the components. Delay in a software component means that the values of an output of the delay component at a time \(t\) do not depend on the values of inputs of the delay component at that time. The system memory offers practically limitless storage for data buffer, and then the stored values are played back to produce a delayed effect.

By TFM, in such a case an output is described as a function of the history of the input and output values before time \(t\), i.e., the value of an output variable does not depend on the most recent event descriptor \(r(T)\). Instead, it is a function of \(p(T)\) (a trace without the last event descriptor). This delay effect can be observed from the trace function table. If the output depends on \(p(T)\) only, we call it one step delay in the component; if the output is a function of \(T\) or \(r(T)\), then the component is delay-free.

In reality, delays always happen because of the time needed for transferring or processing of data, therefore they are sometimes arbitrary but we model a delay as a device that can postpone the result output exactly to some steps with respect to inputs. A delay device is actually a memory whose size depends on how much history must be stored.

### 3.5.3 Delayed and Delay-free Loops in a Network

Communications in a network can be bidirectional. Consider a pair of components \(C_1\) and \(C_2\), which are connected in such a way that component \(C_1\) writes to variable \(\gamma_1\) and reads \(\gamma_2\), while component \(C_2\) reads variable \(\gamma_1\) and writes to \(\gamma_2\) (see Fig 3.1). Without knowing whether these two components are delay-free or delayed components, we are not sure whether they can be combined to a well-behaved constructed component. A component interface document should contain sufficient information to tell us whether or
not there is a delay between a change in an input value and the consequent change in the corresponding output value.

![Diagram of component connections]

Fig. 3.1: Looped Connections between Two Components

In a component TFM interface description, this kind of information can be determined from the expressions defining the output functions. The consideration of whether there are delays within a component motivates the following definition:

**Definition 3.8: immediate dependency description**

An immediate dependency description $D_c$ for a component $C$, is a set of rules of the form $i \rightarrow o$ stating that the value of output variable $o$ at some time may depend on the value of input variable $i$ at the same time. We call a rule with the form $i \rightarrow o$ a D-rule.

The $D$ rules can be derived from TFM documents and would be derivable from any other documentation that describes the output values as functions of input/output history.

The symbol “$\rightarrow$” in Def. 3.1 indicates a connection **between** components. For example, $o \rightarrow i$ means that variable $i$ from one component always gets the value of $o$ from another component without delay; while $i \rightarrow o$ means that within a component, calculating the value of $o$ will need the value of $i$ at the current time. The symbol “$\rightarrow$” indicates a delay-free dependency **within** a component.

For each output variable $o$ of a component $C$ there is a trace function, from this function it can be seen which inputs are directly used to calculate the output value of this variable and therefore the contents of $D_c$ can be obtained. A component $C$ is a delay-free component if there are any D-rules in the immediate dependency description $D_c$, and $C$ is a delayed component if $D_c$ is empty.

Using the D-rules, a delay-free loop can be defined as follows:

**Definition 3.9: immediately influences ($\Rightarrow$)**
Given connectors $\gamma_i$ and $\gamma_j$, $\gamma_i$ immediately influences $\gamma_j$ ($\gamma_i \leftrightarrow \gamma_j$), if there exists a component $C$ with input $i$ and output $o$ such that

- $(\gamma_i \rightarrow i) \in \mathcal{R}$, and
- $(o \rightarrow \gamma_j) \in \mathcal{R}$, and
- $(i \rightarrow o) \in D_C$

**Definition 3.10:** *delay-free loop*

Connectors $\gamma_1, \gamma_2, \ldots, \gamma_n$ form a delay-free loop if:

- for all $i$ ($1 \leq i < n$) $\Rightarrow \gamma_i \leftrightarrow \gamma_{i+1}$, and
- $\gamma_n \leftrightarrow \gamma_1$

**Remark 3.3:** we call a loop as a delayed loop if it is not a delay-free loop.

Now the case in Fig. 3.1 can be refined into the following three cases as shown in Fig. 3.2, where the dashed arrows indicate the delay-free dependency rules in $D$. For example, calculating $o_{11}$ need the current value of $i_{11}$ at that time. Obviously the delay-free or delayed dependency is a component property, but these dependencies can be transferred to interconnectors, as can be seen from the following figure:

![Diagram of delay-free and delayed loops](image)

In Fig 3.2, case a) is a delay-free loop since the interconnectors satisfy the conditions in Def. 3.10. In cases b) and c) the loops are broken in some of the components, in contrast with the delay-free loop these two cases are called delayed loops (Remark 3.3).

A sufficient, **but not necessary**, condition for guaranteeing that a network constructed from components that have well-defined behaviour itself has well-defined behaviour is
to eliminate delay-free loops, i.e., to require that there is delay in every loop if there is any. If the output of one of the components depends only on the earlier values of its input, the behaviour of such networks will be well defined (see section 4.3.5, Lemma 4.3). While if both of the output values depend on its current input values (no delays in either component), the network may not have a unique consistent state and its behaviour might be undefined. This is explained in more detail in the next section.

3.5.4 Why Exclude Delay-free Loops?
The presence of a delay-free loop indicates that the network may have undefined behaviour either because it has no consistent state or because it has more than one consistent state [75]. Here we use the two examples in [70] to explain the two situations.

1) It is unstable
A simple and obvious example of this case in logic circuits is shown in Fig. 3.3. The behaviour of a NOT gate is described by TFM as the following:

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>o</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

**OUTPUT FUNCTIONS:**

\[
o_1(T) = \begin{cases} 
    0 & \text{if } T = \_ \\
    \neg i_1'(\tau(T)) & \text{if } T \neq \_
\end{cases}
\]

From the output function we can see that there is a D-rule \( i_1 \rightarrow o_1 \), therefore except for the initial value there is always \( o_1 = -i_1 \); while because of \( o_1 \rightarrow i_1 \), as shown in Fig 3.3, there is always \( o_1 = i_1 \). These two results contradict with each other and the circuit can never reach a stable state.

2) It has more than one stable state
An example of this case in logic circuits is shown in Fig. 3.4. The behaviour of an OR gate is described by TFM as the following:
OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>O</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

INPUT VARIABLES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₁</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>i₂</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

OUTPUT FUNCTION:

\[
O(T) \equiv \begin{cases} 
0 & (T \neq _) \\
0 & (i₁'(r(T)) = i₂'(r(T)) = 0) \\
1 & \neg (i₁'(r(T)) = i₂'(r(T)) = 0)
\end{cases}
\]

According to the output function and the feedback connections, the network in Fig. 3.4 has two possible solutions, 0 or 1, and there is no way to discuss the initial state, since there is no state variable [75].

Moreover, if a network does have only one stable state and its behaviour is defined, that behaviour can also be implemented by a network that does not have a delay-free loop. Consequently, we consider delay-free loops to be a design error and exclude them in our analysis. If there is a single stable state for all possible inputs, we can write an expression that describes the outputs. From the expression (by parsing it) we can get a network of components that will do it.

In summary, the statement below is the motivation of ruling out delay-free loops from our consideration.

For a delay-free loop, only the following two cases can happen:

(i) It is unstable or has more than one stable state;

(ii) It is stable with exactly one solution, and there is an equivalent network without such a loop.

The meaning of “solution” becomes obvious if the case is formulated as equations consisting of mutually recursive functions:

\[
f_{o₁}(T) = f(p(T), f_{o₂}(T)), f_{o₂}(T) = g(p(T), f_{o₁}(T))
\]
where $o_1$ and $o_2$ are variables of two components, $f_{o_1}$ and $f_{o_2}$ are the output functions for the two output variables and $f_{o_1}(T)$ and $f_{o_2}(T)$ are values of these variables for a trace $T$ respectively, $p(T)$ is the trace containing all but the last event of $T$, $f$ and $g$ are functions that are rewritten from $f_{o_1}$ and $f_{o_2}$, which include the mutual dependency of each other.

By equivalent network we mean a network that always produces the same output as the original one when both are given the same input values.

3.5.5 When is a Network of Components Consistent?

The following definition summarizes conditions sufficient to make it possible (a) to confirm that a network of components, combined to a constructed component, has well-defined behaviour, and (b) to determine the behaviour if we know the behaviour of the components and a description of the network.

**Definition 3.11:** *network consistency*

A network of components is said to be consistent if and only if the following hold:

- The network is completely connected.
- Each network connection rule is data type consistent.
- There is no delay-free loop in the network.

The consistency of a network of components was discussed in [64].
Chapter 4
Determining the Behavior of a Network of Components

In this chapter we present a way to derive a description of the behaviour of a network from the descriptions of the components and their interconnections, if the network is consistent as defined in Def. 3.11.

4.1 Internal and External Behavior of a Network

The internal and external views of a network are the white-box and black-box views introduced in Section 1.2. In Fig 4.1, the two components $C_1$ and $C_2$ are connected to produce a component $C$. The block on the left side is the white box of the system, which includes the internal components and all the interconnections. The simple block on the right side is the black-box view of the system, which is the same as the black-box view of a simple component.

From the black-box view, the internal components and the interconnectors are invisible and the connection information is hidden. The events and traces in this scope are called system events and system traces; and the input/output variables of the constructed component are called system input/output variables. The system event descriptors include only variables visible from the outside of the constructed component.

We call events and traces in the network view as network events and network traces. The network event descriptors include all the internal component input/output variables.

However, we have to note that an internal component ($C_I$ in Fig 4.1, for instance) within a network may itself be a network that is viewed as a black box externally. Thus we do not see all of the state changes (the internal components within $C_I$, if there are any), only the ones on the level we are discussing ($C_I$ and $C_2$).
Now we can clarify the terms in the three levels as follows:

1) The component interface descriptions describing each component as a black box and a network description describing the interconnection of those components; in this view we have component events, component traces and component trace functions. The behavior in this level is described by TFM descriptions of all components and a network description.

2) The network (white-box) view of the system where the internal components and the connections are visible; in this view we have network events, network traces and network trace functions. The behavior in this level is an abstraction of the behavior of all the components and can be derived from 1).

3) The black-box view of the network of components where all the internal components and connections are invisible; in this view we have system events, system traces and system trace functions. The behavior in this level is specified as a black box.

If 1) and 3) are given, we can derive 2) from 1) and compare it with 3) to see whether 1) is the correct design for 3).

The definitions of events, event descriptors, traces, and component interface TFM specifications/descriptions in Chapter 2 are all suitable for the three levels.

In the sequel, for convenience we sometimes call the network view of a system as a network and the black-box view as a system, so that in the three levels the terms components, network and system are used.
The sets of variables included in the three kinds of event descriptors are as follows:

- Component event descriptor for $C_k$: $I_k$ and $O_k$ ($i_1, o_1$ for $C_1$ and $i_2, o_2$ for $C_2$ in Fig 4.2)
- Network event descriptor: $I_1, O_1, I_2, O_2$ ($i_1, o_1, i_2, o_2$ in Fig 4.2)
- System event descriptor: $\Sigma, \Omega$ ($\sigma, \omega$ in Fig 4.2)

Although a system is the same as a component from the black-box view, when a component is mentioned, the internal components of it are not considered but when a system (or constructed component) is mentioned, it is a black box of which the white-box view is also considered. When a network is mentioned, it is a white-box of a system and the behaviour of it will be derived from the internal components and their connections in order to be compared with the black box to see whether or not the white box is a correct design to meet the black-box specification.

4.2 Derivation of an Internal Behavioral Description of a Network

This derivation will be conducted in two steps:

i. Extend all the component trace functions to network trace functions;

ii. Replace the internal variables by its corresponding network trace functions to get a final output as a function of the network traces.

First we explain why is the domain extension necessary by using a network of two simple components where each has only one input and one output variable as shown in Fig 4.2. It will be extended to the cases of more than two components in a network where each component has more than one inputs/outputs.

In Fig. 4.2, the network $N$ is described as the following:

$$ N = (\Delta, \Sigma, \Omega, \mathcal{R}) $$

$$ \Delta = \{C_1, C_2\}, \Sigma = \{\sigma\}, \Omega = \{\omega\}, \mathcal{R} = \{\gamma\} $$

$$ \mathcal{R} = \{\sigma \rightarrow i_1, o_1 \rightarrow \gamma, \gamma \rightarrow i_2, o_2 \rightarrow \omega\} $$

The component TFM interface descriptions for $C_1$ and $C_2$ are $CIT_{k=1,2} = (I_k, O_k, F_k)$, where $I_k, O_k, F_k$ are defined in Def. 2.6 and $k$ denote the $k$-th component.
CIT\textsubscript{1} defines \( f_{o_1} \) as a function that maps component trace \( T_1 \) containing event descriptors with the complete history of \( o_1 \) and \( i_1 \) to the current value of \( o_1 \), and CIT\textsubscript{2} defines \( f_{o_2} \) as a function that maps component trace \( T_2 \) containing event descriptors with the complete history of \( o_2 \) and \( i_2 \) to the current value of \( o_2 \). Because the domain of \( f_{o_1} \) is \( T_1 \) (the range of \( f_{o_1} \) is the value of \( o_1 \)) and the domain of \( f_{o_2} \) is \( T_2 \), it is not convenient to do the function composition directly, since the resulted function will contain traces \( T_1 \) and \( T_2 \). However, if we can extend the domains of \( f_{o_1} \) and \( f_{o_2} \) to the same domain (a set of network traces), and replace the internal variable \( \gamma \) in the final output function by its network trace function, we will get a function that maps network traces \( T \) containing event descriptors with the complete history of \( \sigma, \omega, \gamma \) (or \( i_1, o_1, i_2, o_2 \)) to the final output values of \( \omega \).

The domain of all the component trace functions can be extended to the same one denoted as \( T_d \), the set of network trace functions can be denoted as \( F_d \). In the following sections we will discuss how to derive the \( T_d \) and \( F_d \) respectively.

4.2.1 Derive the Set of Possible Network Traces \( T_d \)

In a network of components, if at some time instant, there is a state change in one component, but no state changes in other components, there may be a network event (see the example in Fig. 4.6 and the result in the table on page 59), depending on how the components are connected. For the variables that are in the component (say \( C_i \)) where there is a state change, the network event descriptor at this time will copy the corresponding values from the component event descriptor. For those variables that are in the component (say \( C_2 \)) where there is no state change, the network event descriptor
should get the after values of the corresponding variables in the previous state change of that component for both the before and after values of these variables. This will be better explained in the following example shown in Fig. 4.3.

In the above figure, if the component event descriptors are $ce_1 = \{(i_1, p_1, q_1), (o_1, p_2, q_2)\}$ and $ce_2 = \{(i_2, p_3, q_3), (o_2, p_4, q_4)\}$, then the network event descriptor at $t_0$ is:

$$ne = \{(i_1, q_1, q_1), (o_1, q_2, q_2), (i_2, p_3, q_3), (o_2, p_4, q_4)\}$$

At time $t_0$ there is no state change in $C_1$, therefore the before and after values of $i_1$ and $o_1$ get the after values from the previous state change. According to Def. 2.8, we may simply denote the derivation of the network event descriptor as: $ne = ce_1' \cup ce_2$.

Because $ce$ and $ce'$ are of the same format we can have a consistent format of the network event descriptor by doing the same operation at any time instant, without dealing with the cases differently. Def. 2.9 is for this purpose.

Using Def. 2.9 the relation between component trace pairs and the corresponding network trace can be defined as:

**Definition 4.1: relation between component trace pairs and network traces**

A relation $U$ in a network $NoC = (\Delta, \Sigma, \Omega, R)$ of two components is defined as:

$$U = \{((T_1, T_2), T) \mid T_{k=1,2} \subseteq T, \forall t, (\forall i \in I \Rightarrow E(T_1)(t) = E(T_2)(t))\}$$

where $\forall i \in I \Rightarrow E(T_1)(t) = E(T_2)(t)$

$$'o'(E(T_1)(t)) = 'i'(E(T_2)(t))$$

$T$ is a network trace, $CIT_1 = (I_1, O_1, F_1)$, $CIT_2 = (I_2, O_2, F_2)$, $T_1$, $T_2$ are component traces and $C_1, C_2 \in \Delta$.

As discussed in Chapter 3, for each network rule $o \rightarrow \gamma \rightarrow i \in R$, $\gamma$ gets the value from $o$ and $i$ gets its value from $\gamma$ without delay. If at any time instant $t$ the before and after
values of the connected variables are not equal, \(E(T_1)(t)\) and \(E(T_2)(t)\) cannot be united into a possible network event/dummy event descriptor, therefore this pair of component traces \((T_1, T_2)\) cannot be united to a network trace \(T\). Based on this fact, we have the following remarks:

**Remark 4.1:** Def. 4.1 is used to derive the corresponding network trace \(T\) for every pair of component traces \(T_{k=1,2}\) which satisfies the consistency condition.

**Remark 4.2:** Relation \(U\) is a partial function (Partial when the domain is the whole set of possible traces in \(C_1\) and \(C_2\), total if the domain is the consistent component traces from the two components). It follows by the definition that different component traces will be united to different network traces if the consistency rule holds. A trace pair \((T_1, T_2)\) cannot be combined to form a meaningful trace if \(\not\exists\)consistentAt\((t)\) does not hold for any time instant \(t\).

From Def. 4.1, for a pair of component traces, theoretically we can derive a possible network trace if go through all time instants to check the value consistency for the connected variables and do union when they are consistent. However, practically there is no need to go through all time instants. Some special time instants will cover all possible cases of the descriptors. In each component the special points are:

- a point for each possible state change, described by a component event descriptor, and
- a point for the time interval between two consecutive state changes, described by a dummy event descriptor.

As described above, to get the set of consistent network traces we can go through the special time instants and check:

- Whether the event pair can possibly happen together;
- Whether the values of the connected variables are consistent for each time instant.

Actually these two checks are the same since time is one of the variables; both are the checking of value consistency for the connected variables. There are three cases in a synchronous model:

- Both components have state changes;
- One component has state change but the other has not;
Both components have no state changes.

It is not necessary to consider the third case; however in Chapter 6 we will see the convenience of including this case in the intermediate steps of derivation. By finding consistent \( ce_1' \) and \( ce_2' \) (see Fig 4.4) and doing a union of them, we can get all network states. Of course the states can also be obtained from doing \( ne' \) on all network events. The two results must be the same. This property will be illustrated in the flip-flop example in Fig 4.7 later in this chapter.

See the following diagram in Fig. 4.4, for example, \( ne = ce_1' \cup ce_2 \) means that there is a state change \( ce_2 \) in component \( C_2 \) but none in component \( C_1 \). The network event \( ne \) will get the state values (\( ce_1' \)) of \( C_1 \) at that time instant. By calculating the three cases a), b), and c) we will get all network event descriptors.

\[
\begin{align*}
\text{a). } & \text{ } \mathcal{R} \text{consistent } \land \ (\text{b). } \mathcal{R} \text{consistent } \land \ (\text{c). } \mathcal{R} \text{consistent } \land \ (\text{d). } \mathcal{R} \text{consistent } \land \\
& \quad \text{ne } = \text{ce}_1' \cup \text{ce}_2 \quad \text{ne } = \text{ce}_1 \cup \text{ce}_2 \quad \text{ne } = \text{ce}_1 \cup \text{ce}_2' \quad \text{ne}' = \text{ce}_1' \cup \text{ce}_2'
\end{align*}
\]

The arrowhead denotes the time instant when event happens.

Although in the TFM the set of all possible traces are not listed in the document, there are sets of predicates which characterize a set of traces, as defined in Def. 2.7. The set of possible traces in the network is \( T_d = \text{range}(U) \). The purpose of defining the relation \( U \) is not only for the derivation of the set of all possible network traces, but also for the formalization of domain extension for the component trace functions. In the next section we will see how the relation \( U \) is used in the formulas for the domain extension.

**4.2.2 Extend the Domain of Component Trace Functions**

Assuming the network trace function for variable \( \omega \) after the domain extension is denoted as \( g_{\omega \sigma} \), which is extended from the component trace function \( f_0 \) defined in component \( C_k \), and \( o \rightarrow \omega \in \mathcal{R} \).
For generality we suppose $g_\omega$ and $f_o$ are relations, and the three relations can be expressed in terms of their domains and ranges:

- $g_\omega$: $<\text{network trace}> \times <\text{value for variable } \omega> (\langle T \rangle \times \langle \text{Type}(\omega) \rangle)$
- $f_o$: $<\text{trace for component } C_k> \times <\text{value for variable } o> (\langle T_k \rangle \times \langle \text{Type}(o) \rangle)$
- $U$: $<\text{component trace pair}> \times <\text{network trace}> (\langle (T_1, T_2) \rangle \times \langle T \rangle)$

where $o$ is an output variable in component $C_k$.

From the three relations it can be seen that if the relation between the network traces and the component traces is given, the relation between the network traces and the values for variable $\omega$ can be established via relation composition. For this purpose we need the inverse relation of $U$.

**Definition 4.2: relation between network traces and component traces**

A relation from network trace to component trace is defined as:

$<\text{network trace}> \times <\text{trace for component } C_k=1,2>:\$

$\pi_k = \{(T, T_k)| \forall t, E(T_k)(t) = e_k\}$

$e_k = \{(v, p, q)| (v, p, q) \in E(T)(t) \land v \in (I_k \cup O_k)\}$ is the set of triples whose variable is in the data structure of $C_k$.

**Remark 4.3**: The relations $\pi_{k=1,2}$ are functions. The domain of this function is the set of network traces $T_d = \text{range}(U)$. For each time instant $t$, $E(T_k)(t)$ returns a set of triples. If there is at least one triple $(v, p, q) \in E(T_k)(t) \Rightarrow p \neq q$, then there is a state change in component $C_k$ at time $t$. If for all triples $(v, p, q) \in E(T_k)(t) \Rightarrow p=q$, then at time $t$ there is no state change in this component. Going through all the interesting time instants, the component traces that have image in $U$ can be extracted back. It may be shorter than the network trace but there is always one and only one $T_k$.

**Remark 4.4**: The relations $\pi_{k=1,2}$ are inverse of $U$. $U$ is to union the component event descriptors for all time points into network event descriptors (ne), while $\pi_{k=1,2}$ are to pick out component event descriptors ($ce_k$) from each network event descriptor ($ne$). The inverse relation $\pi_k$ can be written as the following using $U$:

$\pi_k = \{(T, T_k)| \exists T_\tau \in I_{T_\tau} \text{ s.t. } (\langle T_k, T_\tau \rangle, T) \in U \lor (\langle T_\tau, T_k \rangle, T) \in U\}$
For any component trace relation $f_\alpha$, if there is a network output variable $\omega$ and $o \rightarrow \omega \in \mathcal{R}$, the domain and range of $g_{\omega}$, $f_\alpha$ and $\pi_k$ are:

- $f_\alpha$: <trace for component $C_k$> $\times$ <value for variable $o$> ($<T_k> \times <\text{Type}(o)>$)
- $\pi_k$: <network trace> $\times$ <trace for component $C_k$> ($<T> \times <T_k>$)
- $g_{\omega}$: <network trace> $\times$ <value for variable $\omega$> ($<T> \times <\text{Type}(\omega)>$)

The relationship of these three relations can be described by the diagram in Fig 4.5. The component traces $T_1$ and $T_2$ in $C_1$ and $C_2$ can be combined to form a network trace $T$ through relation $U$, if the consistency conditions defined in Def. 4.1 are satisfied.

On the other hand, the network trace $T$ can be mapped to $T_1$ and $T_2$ by $\pi_1$ and $\pi_2$, respectively. In component $C_1$, the values of variable $o_1$ are constrained by relation $f_{o_1}$; and in component $C_2$ the values of variable $o_2$ are constrained by relation $f_{o_2}$. In the network, the values of $\omega_1$ and $\omega_2$ are constrained by relations $g_{\omega_1}$ and $g_{\omega_2}$, respectively, where $o_1 \rightarrow \omega_1 \in \mathcal{R}$ and $o_2 \rightarrow \omega_2 \in \mathcal{R}$. Any network trace $T$ can be first mapped to component traces and then to the values of outputs through the route $\pi_1 \circ f_{o_1}$ and $\pi_2 \circ f_{o_2}$ as well as through the $g_{\omega_1}$ and $g_{\omega_2}$ directly. The two lines from $T$ to the values of $\text{Type}(\omega_k)$ and the two lines from $T_k$ to the values of $\text{Type}(o_k)$ are used to indicate that $f_{o_1}$, $f_{o_2}$, $g_{\omega_1}$ and $g_{\omega_2}$ are relations. In these cases the formula $\pi_1 \circ f_{o_1} \subseteq g_{\omega_1}$ and $\pi_2 \circ f_{o_2} \subseteq g_{\omega_2}$ must hold, whose proof is given in that of theorem 4.1. Here “$\circ$” is the standard relation composition operator.

The diagram in Fig 4.5 consists of two independent “commuting diagrams” for variable $\omega_1$ and $\omega_2$ respectively. The solid triangles above and below the dashed line can be drawn separately starting from a network trace $T$, which means that these two groups of triangles do not depend on each other and they are for two network output variables. For each network output variable $\omega_k$ there is such a diagram that commutes for every network trace $T$ in $\mathcal{T}_d$. 
Theorem 4.1:

For any $f_o$ in a component trace relation set $F_k$, if there is a network output $\omega$ and $o \rightarrow \omega \in \mathcal{R}$, then $\pi_k \circ f_o \subseteq g_\omega$ is the relation between network trace relation and the component trace relation, assuming $g_\omega$ is the network trace relation.

Proof:

According to the definition of $\pi_k$, any network trace $T \in T_d$ will be mapped to a component trace $T_k \in T_k$, written as $T_k = \pi_k(T)$ or $(T, T_k) \in \pi_k$.

According to the definition of $f_o$ in component $C_k$, the value of $o$ at the moment, say $p$, is constrained by $f_o$, written as $(T_k, p) \in f_o$. From $(T, T_k) \in \pi_k$ and $(T_k, p) \in f_o$, it can be seen $(T, p) \in \pi_k \circ f_o$.

For the same network trace $T \in T_d$, the value of $\omega$ at the same moment, say $q$, there should be a relation $g_\omega$, such that $(T, q) \in g_\omega$. The network rule $o \rightarrow \omega \in \mathcal{R}$ and the consistency condition in Def. 4.3 require that $p = q$ at any time, which means $(T, p) \in g_\omega$.

The above says that for any $T \in T_d$, $(T, p) \in \pi_k \circ f_o \Rightarrow (T, p) \in g_\omega$, or $\forall (T, p) \in \pi_k \circ f_o \Rightarrow (T, p) \in g_\omega$, $\pi_k \circ f_o \subseteq g_\omega$ is proved.

If $f_o$ is a function, then $g_\omega$ must be a function. In such cases the trace $T$ in $T_d$ can be mapped by $g_\omega$ as well as $\pi_k \circ f_o$, the formula can be written as:

$$g_\omega = \pi_k \circ f_o$$
If \( F_k = (f_o)_{o \in \Omega_k} \) is the set of component trace functions for component \( C_k \), then we can get a set of network trace functions \( G_k \) by domain extension:

\[
G_k = \{ g_\omega \mid g_\omega = \pi_k \circ (f_o)_{o \in \Omega_k} \text{ and } o \rightarrow \omega \in \mathcal{Y} \}
\]

is the set of network trace functions for the variables in component \( C_k \).

\( F_d = G_1 \cup G_2 \) is the set of network trace functions by extending the domains of all component trace functions.

**4.2.3 The Derivation Relation**

As can be seen, the network traces contain all of the component variables. We denote the derived TFM descriptions of the network as:

\[
CIT_e = (\Sigma', \Omega', F_d)
\]

where \( \Sigma', \Omega' \) are the union of all component inputs and component outputs.

Now the sets \( T_d \) and \( F_d \) for the network are derived from the component TFM descriptions. First we introduce a formula for domain extension.

**Definition 4.3:** the TFM description of a network of two components

If a network \( NoC = (\Delta, \Sigma, \Omega, \mathcal{Y}) \) has two components \( C_1 \) and \( C_2 \) with the TFM interface descriptions being \( CIT_1 = (I_1, O_1, F_1) \) and \( CIT_2 = (I_2, O_2, F_2) \), then the TFM description of the network \( CIT_e \) is a quadruple \( (\Sigma', \Omega', F_d) \), where:

- \( \Sigma' = I_1 \cup I_2 \)
- \( \Omega' = O_1 \cup O_2 \)
- \( F_d = G_1 \cup G_2 \) where \( G_k = \{ g_\omega \mid g_\omega = \pi_k \circ (f_o)_{o \in \Omega_k} \text{ and } o \rightarrow \omega \in \mathcal{Y} \} \)

Sections 4.2.1 and 4.2.2 are the justification of this definition. The domain of the functions in \( F_d \) is \( T_d = \text{range}(U) \).

Using the relation \( g_\omega = \pi_k \circ (f_o)_{o \in \Omega_k} \), one can get a set of equations of the form “\( \omega(T) \equiv \text{a tabular expression} \)” where \( \omega \in \Omega' \) and \( T \) is a network trace.

Such an equation was defined in component \( C_k \) in terms of the input and output history of that component. After the domain extension, the variables (inputs/outputs of that component) are still in the equation. Some of the input variables \( (i) \) may be connected to
the output \( o \) of other components by an interconnector variable \( \gamma \); the domain of the output function for \( o \) \( (\gamma) \) is also extended to network traces. In this case the variable \( i \) can be substitute by the network trace function for \( o \). If there is no loop in the network, this procedure should eventually result in an equation in which the variables are all system inputs.

After the domain extension and the internal variable substitution, for a network without loop we may get a set of output functions for the external output variables in \( \Omega \) and the traces may include the history of the variables in \( \Sigma \) and \( \Omega \) only.

**Definition 4.4:** the TFM description of a constructed component

If a network \( NoC = (\Delta, \Sigma, \Omega, \mathcal{R}) \) has two components \( C_1 \) and \( C_2 \) with the TFM interface descriptions being \( CIT_1 = (I_1, O_1, F_1) \) and \( CIT_2 = (I_2, O_2, F_2) \), then the TFM description of the constructed component \( CIT_d \) is a quadruple \((\Sigma, \Omega, G)\), where:

- \( \Sigma = I_1 \cup I_2 - I_{\text{inter}} \) where \( I_{\text{inter}} = \{i | (i \in I_1 \lor i \in I_2) \land i \leftarrow \gamma \leftarrow o \in \mathcal{R}\} \)
- \( \Omega = O_1 \cup O_2 - O_{\text{inter}} \) where \( O_{\text{inter}} = \{o | (o \in O_1 \lor o \in O_2) \land i \leftarrow \gamma \leftarrow o \in \mathcal{R}\} \)
- \( G = (g_o)_{o \in \Omega} \)

The functions in \( G \) are the network trace \((T_d = \text{range}(U))\) functions where the internal variables are substituted by the trace functions for the variables that are connected to them.

For example for the network in Fig 4.2, one can derive a function \( g_\gamma \) from the function \( f_{o_\gamma} \), and \( g_o \) from the function \( f_{o_2} \). The domain of \( g_\gamma \) and \( g_o \) includes traces with event descriptors extended to include all variables in the network. The range of \( g_o \) is the same as the range of \( f_{o_2} \) and the range of \( g_\gamma \) is the same as the range of \( f_{o_\gamma} \). The variable values in the network can be determined as the following:

- The values of \( \sigma \) that appear in \( T \) are determined outside of \( N \).
- The values of \( o_2 \) and \( \omega \) after the last event in \( T \) are given by \( g_o(T) \).
- The values of \( o_1, \gamma, \) and \( i_2 \) after the last event in \( T \) are given by \( g_{\gamma}(T) \).

Because there are no loops in the network, the final values of each of these variables can be described by an expression that does not mention that variable. By replacing the
intermediate variables with the output variables that are defined as a function of network traces, the values of \( i_1, o_1, \gamma, i_2, o_2, \) and \( \omega \) are all determined by the history of values of \( \sigma \).

### 4.2.4 A Simple Example without Loops

As can be seen, Def. 4.1 is the basis of the domain extension formula in Def. 4.3. If the event descriptors in each component can be enumerated, we can use Def. 4.1 to construct the network traces. If the network traces (predicates on network traces) can be derived, the trace functions are already known. In the next section we provide a simple example to illustrate how to get network traces by using Def. 4.1 directly. In Appendix 1, an example is provided by using Def. 4.1 and 4.3 respectively, and the same results are deducted. For the cases where the event descriptors cannot be enumerated in Chapter 6, Def. 4.3 and Def. 4.4 will always be used.

Suppose components \( C_1 \) and \( C_2 \) in Fig. 4.2 are identical devices that will produce output \( v \) corresponding to input \( i \) as illustrated in Fig. 4.6. We will present how to get network traces (event descriptors) using Def. 4.1.

![Fig. 4.6: The Input/Output Waveforms of a “Flip-flop”](image)

They are connected to produce a constructed component \( C \) as illustrated in Fig. 4.7:

![Fig. 4.7: Two Connected “Flip-flops”](image)

Their TFM interface descriptions are listed in the following.
Component C₁:

Input variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₁</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

Output variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₁</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

In the following table the set of all possible component event descriptors are listed. Each event descriptor describes a set of component state changes that are identical except in time and will appear repeatedly. We denote the descriptor as ce₁k to indicate component state change at time t. Normally the superscript t will be omitted.

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>values of ce₁k</th>
<th>values of ce₁k'</th>
<th>Name of ce₁k'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce₁₁</td>
<td>{(i₁, 0, 1), (v₁, 0, 1)}</td>
<td>{(i₁, 1, 1), (v₁, 1, 1)}</td>
<td>ce₁₁'</td>
</tr>
<tr>
<td>ce₁₂</td>
<td>{(i₁, 1, 0), (v₁, 1, 1)}</td>
<td>{(i₁, 0, 0), (v₁, 1, 1)}</td>
<td>ce₁₂'</td>
</tr>
<tr>
<td>ce₁₃</td>
<td>{(i₁, 0, 1), (v₁, 1, 0)}</td>
<td>{(i₁, 1, 1), (v₁, 0, 0)}</td>
<td>ce₁₃'</td>
</tr>
<tr>
<td>ce₁₄</td>
<td>{(i₁, 1, 0), (v₁, 0, 0)}</td>
<td>{(i₁, 0, 0), (v₁, 0, 0)}</td>
<td>ce₁₄'</td>
</tr>
</tbody>
</table>

v₁(T₁) =

\[
T₁ = \_ \\
'i₁(r(T₁)) = 0 \land i₁'(r(T₁)) = 1 \\
\neg ('i₁(r(T₁)) = 0 \land i₁'(r(T₁)) = 1)
\]

\[v₁(p(T₁))\]

The behaviour of component C₂ is completely the same with C₁, only the variable names are different, as shown below:

Input variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₂</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

Output variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v₂</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>
Possible Event Descriptor sets

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>value of ce_{2k}</th>
<th>value of ce_{2k}'</th>
<th>Name of ce_{2k}'</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_{21}</td>
<td>{(i_2, 0, 1), (v_2, 0, 1)}</td>
<td>{(i_2, 1, 1), (v_2, 1, 1)}</td>
<td>ce_{21}'</td>
</tr>
<tr>
<td>ce_{22}</td>
<td>{(i_2, 1, 0), (v_2, 1, 1)}</td>
<td>{(i_2, 0, 0), (v_2, 1, 1)}</td>
<td>ce_{22}'</td>
</tr>
<tr>
<td>ce_{23}</td>
<td>{(i_2, 0, 1), (v_2, 1, 0)}</td>
<td>{(i_2, 1, 1), (v_2, 0, 0)}</td>
<td>ce_{23}'</td>
</tr>
<tr>
<td>ce_{24}</td>
<td>{(i_2, 1, 0), (v_2, 0, 0)}</td>
<td>{(i_2, 0, 0), (v_2, 0, 0)}</td>
<td>ce_{24}'</td>
</tr>
</tbody>
</table>

\[ v_2(T_2) = \]
\[
\begin{align*}
T_2 &= \_ \\
r(T_2) &= ce_{21} \lor r(T_2) = ce_{23} \\
r(T_2) &= ce_{22} \lor r(T_2) = ce_{24}
\end{align*}
\]

0

\[ \neg v_2(p(T_2)) \]

\[ v_2(p(T_2)) \]

In the above TFM descriptions, we use \( T_1 \) and \( T_2 \) to distinguish the traces in different components and the network trace. According to Def. 4.1, we can find the network trace \( T \) by checking that \( \forall t, E(T)(t) = E(T_1)(t) \cup E(T_2)(t) \) and \( \mathcal{R}_{consistentAt}(t) \), as shown in the following table.

Table 4.1: The relation between component trace pairs and network traces

<table>
<thead>
<tr>
<th>U (&lt;T_1&gt; \times &lt;T_2&gt; \times &lt;T&gt;)</th>
<th>T[0]</th>
<th>T[2]</th>
</tr>
</thead>
<tbody>
<tr>
<td>ce_{11}</td>
<td>ne_{1}</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{12}</td>
<td>\times</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{13}</td>
<td>\times</td>
<td>ne_{3}</td>
</tr>
<tr>
<td>ce_{14}</td>
<td>\times</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{11}'</td>
<td>\times</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{12}'</td>
<td>\times</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{13}'</td>
<td>\times</td>
<td>\times</td>
</tr>
<tr>
<td>ce_{14}'</td>
<td>\times</td>
<td>\times</td>
</tr>
</tbody>
</table>

where:

\[ ne_{1} = ce_{11} \cup ce_{21} = \{(i_1, 0, 1), (v_1, 0, 1), (i_2, 0, 1), (v_2, 0, 1)\} \]
\[ ne_{2} = ce_{11} \cup ce_{23} = \{(i_1, 0, 1), (v_1, 0, 1), (i_2, 0, 1), (v_2, 1, 0)\} \]
\[ ne_{3} = ce_{13} \cup ce_{22} = \{(i_1, 0, 1), (v_1, 1, 0), (i_2, 1, 0), (v_2, 1, 1)\} \]
\[ ne_{4} = ce_{13} \cup ce_{24} = \{(i_1, 0, 1), (v_1, 1, 0), (i_2, 1, 0), (v_2, 0, 0)\} \]
\[ ne_{5} = ce_{12} \cup ce_{21}' = \{(i_1, 1, 0), (v_1, 1, 1), (i_2, 1, 1), (v_2, 1, 1)\} \]
\[ ne_{6} = ce_{12} \cup ce_{23}' = \{(i_1, 1, 0), (v_1, 1, 1), (i_2, 1, 1), (v_2, 0, 0)\} \]
\[ ne_{7} = ce_{14} \cup ce_{22}' = \{(i_1, 1, 0), (v_1, 0, 0), (i_2, 0, 0), (v_2, 1, 1)\} \]
\[ ne_{8} = ce_{14} \cup ce_{24}' = \{(i_1, 1, 0), (v_1, 0, 0), (i_2, 0, 0), (v_2, 0, 0)\} \]
#1: \( ne_1' = ce_{11}' \cup ce_{21}' = \{(i_1, 1, 1), (v_1, 1, 1), (i_2, 1, 1), (v_2, 1, 1)\} \)

#2: \( ne_2' = ce_{11}' \cup ce_{23}' = \{(i_1, 1, 1), (v_1, 1, 1), (i_2, 1, 1), (v_2, 0, 0)\} \)

#3: \( ne_5' = ce_{12}' \cup ce_{21}' = \{(i_1, 0, 0), (v_1, 1, 1), (i_2, 1, 1), (v_2, 1, 1)\} \)

#4: \( ne_6' = ce_{12}' \cup ce_{23}' = \{(i_1, 0, 0), (v_1, 1, 1), (i_2, 1, 1), (v_2, 0, 0)\} \)

#5: \( ne_3' = ce_{13}' \cup ce_{22}' = \{(i_1, 1, 1), (v_1, 0, 0), (i_2, 0, 0), (v_2, 1, 1)\} \)

#6: \( ne_4' = ce_{13}' \cup ce_{24}' = \{(i_1, 1, 1), (v_1, 0, 0), (i_2, 0, 0), (v_2, 0, 0)\} \)

#7: \( ne_7' = ce_{14}' \cup ce_{22}' = \{(i_1, 0, 0), (v_1, 0, 0), (i_2, 0, 0), (v_2, 1, 1)\} \)

#8: \( ne_8' = ce_{14}' \cup ce_{24}' = \{(i_1, 0, 0), (v_1, 0, 0), (i_2, 0, 0), (v_2, 0, 0)\} \)

The Table 4.1 above is a normal function table. The event descriptors at different time instants for \( C_1 \) are denoted as \( ce_{1k} \) \((k=1...4)\) and listed in grid \( T[1] \), and those for \( C_2 \) are denoted as \( ce_{2k} \) \((k = 1...4)\) and listed in grid \( T[2] \). If \( \Re \) consistentAt \((t) \) holds for \( ce_{1i} \) and \( ce_{2j} \), there is a corresponding network event descriptor \( ne_k \) in the corresponding cell \( ij \) of grid \( T[0] \) and \( ne_k = ce_{1i} \cup ce_{2j} \).

In Def. 2.9, we denote the descriptor at time \( t \) as \( E(T)(t) \), therefore the relation \( ne_k = ce_{1i} \cup ce_{2j} \) can be written as \( E(T)(t) = E(T_1)(t) \cup E(T_2)(t) \) and it describes the relation among traces \( T_1, T_2 \) and \( T \), in which the equation holds true for all \( t \). According to Table 4.1, we can write a set of predicates that the traces \( T_1, T_2 \) and \( T \) must satisfy. For instance, the predicate for cell \( 32 \) is:

\[
 r(T_1) = ce_{13} \land r(T_2) = ce_{22} \Rightarrow U(T_1, T_2) = T \text{ s.t. } r(T) = ne_3
\]

From \( ne_3 = ce_{13} \cup ce_{22} = \{(i_1, 0, 1), (v_1, 1, 0), (i_2, 1, 0), (v_2, 1, 1)\} \), it can be seen that the before and after values of the connected variables \( i_2 \) and \( v_1 \) are equal and there is a network event descriptor calculated from the two component event descriptors. If there is a “×” sign in a cell of grid \( T[0] \), then it is impossible to combine these \( T_1 \) and \( T_2 \) into a network trace \( T \), because the \( \Re \) consistentAt \((t) \) does not hold.

By observing Table 4.1 and the calculation formula, it can be seen that if there is a component state change in \( C_1 \), there is always a network event, no matter there is a state change in \( C_2 \) or not. But if there is a state change in \( C_2 \), while no state change in \( C_1 \), there is no consistent network event (the bottom left block in grid \( T[0] \)). This is natural considering the way in which the two components are connected. It is a series connection and if there is no state change in \( C_1 \) yet, it is not possible that anything would change in
In the network trace functions, the values of $v_1$ do not depend on any values from variables of component $C_2$, but the values of $v_2$ depend on the variables of component $C_2$. If the connections are bidirectional, i.e., there is an interconnector from $C_2$ to $C_i$ as well, this area will not be empty. This situation can be observed in the RS flip-flop example in Section 6.1.

We can see that the calculated descriptors for states $(ce_{11}' \cup ce_{21}' = ne_1')$ are exactly the same with those by doing $ne'$ operation on $ne$, as defined in Def. 2.8. This is not coincidental, but a nice feature of Def. 4.1 – the network will get a set of consistent event values and state values. Since there is no event in this area (the gray part in the table, where both components have no event), there is no need to consider it in getting the final network trace functions.

According to the sets of network event descriptors, the network trace functions for the two output variables can be written as:

$$\gamma(T) \equiv \begin{array}{ccc} T = & 0 & \gamma(p(T)) \\ r(T)=ne_1 \lor r(T)=ne_2 \lor r(T)=ne_3 \lor r(T)=ne_4 & \end{array}$$

$$\omega(T) \equiv \begin{array}{ccc} T = & 0 & \omega(p(T)) \\ r(T)=ne_1 \lor r(T)=ne_2 & \end{array}$$

According to the derived TFM descriptions of the network, the waveform of the constructed component is shown in Fig. 4.8.
4.2.5 Does the Network Have Well-defined Behaviours with Delayed Loops?

If there is a delay-free loop in a network, in the variable substitute procedure introduced in the Section 4.2.3, one will keep getting internal variables in the expressions no matter how often the substitution procedure is used. That is why in Chapter 3 the delay-free loops were suggested to be avoided in software design. However, if a loop is with delay, we can show the internal variables will eventually be removed in the equations for a system output.

**Lemma 4.3: delayed loops can lead to defined behaviour**

If there are delayed loops between two components, the constructed component will have well-defined behaviours.

**Proof:**

**Case 1:** if there are delays in both components, the delayed loop can be expressed as bellow:

\[
\gamma_1(T) = F(\gamma_2(p(T))) \quad (1)
\]
\[
\gamma_2(T) = G(\gamma_1(p(T))) \quad (2)
\]

where \( T \) is a network trace.

First convert mutual recursion (1) + (2) (delayed loop) to a single recursion (3) (loop equation):
let: $\gamma_{12}(T) = (\gamma_1(T), \gamma_2(T)) = (F(\gamma_2(p(T))), G(\gamma_1(p(T))))$

$\Rightarrow \gamma_{12}(p(T)) = (\gamma_1(p(T)), \gamma_2(p(T)))$

$\Rightarrow \gamma_{12}(T) = (F(y), G(x))$ where $(x,y) = \gamma_{12}(p(T))$

$\Rightarrow \gamma_{12}(T) = (F(second(\gamma_{12}(p(T))), G(first(\gamma_{12}(p(T)))))$

Case 2: If there is delay in one of the components only:

$\gamma_1(T) = F(\gamma_2(p(T)))$ \hspace{1cm} (4)

$\gamma_2(T) = G(\gamma_1(T))$ \hspace{1cm} (5)

$\Rightarrow \gamma_1(T) = F(G(\gamma_1(p(T)))) = F'(\gamma_1(p(T)))$

$\Rightarrow \gamma_2(T) = G(F(\gamma_2(p(T)))) = G'(\gamma_2(p(T)))$

In the following we show that the loop equation of a delayed loop will converge theoretically.

Since it is a delayed loop, the recursion can be expressed using fold [51]:

$fold(c, h)(\_\_) = c$

$fold(c, h)(p(T).r(T)) = h(r(T), fold(c, h)(p(T)))$

where $c$ is a constant and $h$ is a function

In functional programming, fold is a standard operator that encapsulates a simple pattern of recursion for processing lists. It comes equipped with a proof principle that avoids the need for inductive proofs, and a definition principle that guides the transformation of recursive functions into definitions using the fold [51].

Obviously the scheme can be proceeded to reach the point where the value of an empty trace is defined. For example, if $\gamma$ is defined as:

a) $\gamma(\_\_) = 0$

$\gamma(p(T).r(T)) = F(r(T)) + \gamma(p(T))$

let $f(e, x) = F(e) + x$

$\Rightarrow \gamma(p(T).r(T)) = f(r(T), \gamma(p(T))$

$\Rightarrow \gamma = fold(0, f)$

or:
b) \( \gamma(\_ \_ \_ \_ \_ \_ \_ \_ \_.) = 1 \)

\[
\gamma(p(T), r(T)) = G(r(T)) \cdot \gamma(p(T))
\]

let \( g(e, x) = G(e) \cdot x \)

\[
\Rightarrow \gamma(p(T), r(T)) = g(r(T), \gamma(p(T))
\]

\[
\Rightarrow \gamma = \text{fold}(1, g)
\]

In this manner the function \( \gamma \) on traces (lists) can be defined using \( \text{fold} \). The universal property of \( \text{fold} \) states that for finite lists, the function \( \text{fold}(c, h) \) is not just a solution to its defining equations, but also the unique solution [51].

This says that a network with a delayed loop has well-defined behaviours and the internal variables will eventually be removed.

It completes the proof.

For example, the network in Fig 4.9 shows a network that contains a loop. The network description \( \mathcal{R} \) contains four rules:

\[
\mathcal{R} = \{ \sigma \rightarrow i_{11}, o_1 \rightarrow \gamma_1, \gamma_1 \rightarrow i_2, o_2 \rightarrow \gamma_2, \gamma_2 \rightarrow i_{12}, o_2 \rightarrow \omega \}
\]

It is a delayed loop if at least one of the components introduces a delay, i.e. if either of the following holds:

- The output \( o_1 \) does not immediately depend on \( i_{12} \), i.e. \( (i_{12} \rightarrow \bullet o_1) \notin D_{c_1} \);
- The output \( o_2 \) does not immediately depend on \( i_2 \), i.e. \( (i_2 \rightarrow \bullet o_2) \notin D_{c_2} \);

For the case where the output \( o_2 \) does not immediately depend on \( i_2 \), the value of \( o_2 \) is determined by the prefix of \( T \) that is obtained by removing the last event descriptor of \( T \). In other words, the current value of \( o_2 \) can be calculated knowing only the past values of \( i_2 \). The past values of \( i_2 \) can be calculated from the history values of \( i_{11} \) and \( i_{12} \).

For the case where the output \( o_1 \) does not immediately depend on \( i_{12} \), the values of \( o_1, \gamma_1, i_2, o_2, \) and \( \omega \) can be determined knowing the current and past values of \( \sigma \) and the past values of \( o_2 \).
A concrete example for this case is the RS flip-flop and will be shown in Section 6.1.

If any of the components in the above examples are not deterministic, the relationship among the three relations are described by the commuting diagram in Fig 4.5, and by the formula in Theorem 4.1, i.e., instead of \( g_{\omega} = \pi_k \circ f_\omega \), it would be \( g_{\omega} \supseteq \pi_k \circ f_\omega \).

### 4.2.6 The TFM Descriptions for a Network of more than Two Components

As can be seen from Def. 4.3 and 4.4, the domain extension and the substitution of internal variables depend neither on the number of input/output variables in each component, nor the number of the interconnectors and the directions of the interconnections in a network. This means that the domain extension and variable substitution operation is commutative and associative, it is sufficient to define the TFM descriptions for a network of two components. If there are more than two components in a network, by using the operation in Def. 4.3 repeatedly on two components one can finally get the TFM description of a network with any number of components. It can start from a pair of components \( C_1 \) and \( C_2 \), calculate the \( CIT_e \) and denote the constructed component as \( C \) and combine it with \( C_3 \), calculate the \( CIT_e \) again, until get the \( CIT_e \) of the whole network and then substitute the internal variables to get the \( CIT_d \) for the constructed component \( C \).

Alternatively we can also first calculate the \( CIT_d \) (both the domain extension and variable substitution) and denote the constructed component as \( C \) and combine it with \( C_3 \), calculate the \( CIT_d \) again, until we get the \( CIT_d \) of the whole network.
4.3 Simplifying Expressions in the TFM Descriptions of a Network

After the internal variables in an output function are substituted according to the consistency rules, there may be many expressions that can be simplified by some rules. For example, if $o \rightarrow \omega \in \mathcal{H}$, and $f_o$ is a function, the calculated function $g_{o\omega}$ is $g_{o\omega}(T) = \pi_k f_o(T) = f_o(\pi_k(T))$. Since $f_o$ is given as a table, we may generate another table for $g_{o\omega}$ by substituting component traces with $\pi_k(T)$ in $f_o$. Because the network trace and the corresponding component trace are closely related, expressions of the form such as $'o(r(\pi_k(T)))$ and $'i(r(\pi_i(T)))$ can be simplified.

Lemma 4.4:

For any variable $v$ in a network, we have

\begin{align*}
\text{• } v'(r(\pi_k(T))) &= v'(r(T)) \quad &\text{1)} \\
\text{• } 'v(r(\pi_k(T))) &= \begin{cases} 
'v(r(T)) & \text{if } t(r(\pi_k(T))) = t(r(T)) \\
'v(E(T)(t_0)) & \text{if } t_0 = t(r(\pi_k(T))) < t(r(T))
\end{cases} \quad &\text{2)}
\end{align*}

$\pi_k$ is the projection function to the $k$-th component in the network.

Proof:

The proof follows from Def. 4.1 and 4.3. After the last component state change (e.g. $e_{22}$ in Fig. 4.10), the after value of a variable (e.g. $v_2$) for that component at $e_{22}$ has not been changed even there might be a lot of network events ($e_3$, $e_4$, $e_5$). It is obvious that the after values in the last network event and in the last component state change are the same ($v_2'(e_{22}) = v_2'(e_5)$, here $e_{22}$ is denotation of $r(\pi_2(T))$ and $e_5$ is for $r(T)$, therefore $v_2'(r(\pi_k(T))) = v_2'(r(T))$, equation 1) is proved).

Because the last network event may be the union of a state in one component and events in other components, the before value of a variable $v_2$ in the last network event ($e_3$) is the after value of that variable in $e_{22}$. The before value in the component state change ($v_2$ in $e_{22}$) is missing in the last network event ($e_3$) since this variable (for example $v_2$) keeps redundant after values of the last component state change. In this case $'v(r(\pi_k(T))) = 'v(r(T))$ does not hold.

If there is a component state change (e.g. $e_{14}$) corresponding to the last network event, the before value is remembered in the network event descriptor and $'v(r(\pi_k(T)) = \ldots$
‘v(r(T)) holds; otherwise we have to go back to the point where the component has a state change and copy the value 'v(E(T)(t_2)) at that time t_2, which is 'v(e_{22}) in the diagram.

According to the Lemma 4.4, the expressions in a function table can be modified in some cases. For example:

\[
v_{4d}(T) = (\gamma_4) \begin{array}{c}
\pi_d(T) = _- \\
\pi_d(T) \neq _-
\end{array} 0 \begin{array}{c}
'_{i41}(r(\pi_d(T)))
\end{array}
\]

\[
\begin{array}{c}
\pi_d(T) = _-
\pi_d(T) \neq _-
\end{array} t(r(\pi_d(T))) = t(r(T)) \begin{array}{c}
0
\end{array} \begin{array}{c}
'_{i41}(r(T))
'_{i41}(E(T)(t_0))
\end{array}
\]

In this expression the consistency rules can be used on 'i_{41}(E(T)(t_0)). For example ‘v_3(E(T)(t_0))='i_{41}(E(T)(t_0)). In general we need to define some other auxiliary functions to extract the values like ‘v_3(E(T)(t_0)) from a defined trace function table.

**Lemma 4.5:**

For any variable v and a non-empty trace T in a network, we have

- ‘v(r(T))=v(p(T)), v’(r(T))=v(T), or more generally,
- ‘v(E(T)(t_0))=v(p(pt(t_0, T))), v’(E(T)(t_0))=v(pt(t_0, T))

where pt(t, T) = \{e_1, e_2, \ldots, e_k\}, t(e_k) = t is a trace containing all the event descriptors no later than t and T = \{e_1, e_2, \ldots, e_n\}, t(e_i) < t(e_2) < \ldots < t(e_n).

**Proof:**

pt(t_0, T) \neq _- \Rightarrow ‘v(E(T)(t_0))=v'(r(p(pt(t_0, T)))) \quad (Continuity Condition),
\[
\begin{align*}
\forall \nu(p(t_0, T)) & \quad \text{(Output Condition)} \\
pt(t_0, T) &\Rightarrow \forall (E(T)(t_0)) = \nu(-) \quad \text{(Initialization Condition)} \\
\text{Hence, } \forall (E(T)(t_0)) & = \nu(p(pt(t_0, T))) \\
\nu'(E(T)(t_0)) & = \nu(pt(t_0, T)) \quad \text{(Output Condition)} \\
\text{If } t_0 & = t(r(T)), \forall (E(T)(t_0)) = r(T), \pt(t_0, T) = T \text{ thus } \forall (r(T)) = \nu(p(T)) \text{ and } \nu'(r(T)) = \nu(T) \\
\text{This completes the proof.}
\end{align*}
\]
Chapter 5

Correctness of a Network of Components

If a set of components can fit together to perform some task and the behavioral TFM description of the network is derived, it is desirable to know whether the network (white box) has been designed correctly to provide the required black-box behavior. In this chapter we discuss how to check whether the set of components can work together correctly, i.e., the network will function satisfactorily if each of the components is implemented correctly.

5.1 The Correctness of a White Box

Correctness is not something we can check alone mathematically. We can only check if an implementation satisfies a specification [92]. To do this checking, we need to assume that a system requirement specification is provided.

Here the “implementation” is a “relative white-box” view of a system. “Relative” means that the depth of the details depends on the level we are considering. For example, a program that is actually executed is an implementation of the program’s specification; a set of program functions that is an internal design of a component is an implementation of the component interface specification; and a software design, including component interface descriptions and the connection description of the components, is an implementation of the system requirement specification.

If the implementation is described precisely by functions/relations, an unambiguous requirement specification is provided, and the same description notation and method are used at the requirements and implementation stage, the correctness can be checked based on the relational approach in [90].

Since correctness is a mathematical property that establishes conformance of an implementation and its specifications, it depends largely on the rigorousness and unambiguousness of the requirement specifications as well as the description of the implementation. If the software design documents are written mathematically and
precisely, then conformance realtion can be properly defined and hence the correctness of a design can be proved accordingly.

We will present an approach to check the correctness of a set of software design documents with respect to its requirements specifications. The design documents include component interface TFM descriptions of a set of components and a network description describing the interconnection of these components.

5.2 The Basic Conformance Relation

In the relational approach [90], the behaviour of a system is described by relations between pairs of variables. For instance, the requirements are described by two relations: NAT and REQ, both being relation between the monitored and controlled variables. NAT describes the (natural) boundaries of the system and REQ described the behaviour of the system.

Because the requirements should specify behaviour for all cases that can arise, it should be true that,

\[ \text{domain}(\text{REQ}) \supseteq \text{domain}(\text{NAT}) \]  

(1)

The relation REQ can be considered feasible with respect to NAT if (1) holds and

\[ \text{domain}(\text{REQ} \cap \text{NAT}) = (\text{domain}(\text{REQ}) \cap \text{domain}(\text{NAT})) \]  

(2)

Feasibility, in the above sense, means that nature (as described by NAT) will allow the required behaviour (as described by REQ); it does not mean that the functions involved are computable or that an implementation is practical.

Note that (1) and (2) imply that,

\[ \text{domain}(\text{REQ} \cap \text{NAT}) = \text{domain}(\text{NAT}) \]  

(3)

The basic conformance relation in this thesis is derived from the above relations and the correctness relation in [69] and this basic relation will be used to establish a conformance relation between a TFM description of a network and a system requirements specification.
Definition 5.1: conformance relation

According to relational approach in [69, 90], if an implementation can be described by a relation B, and the requirement is specified by a relation A, the implementation conforms to the specification if $B \subseteq A$ and $\text{domain}(B) \supseteq \text{domain}(A)$.

Where $B \subseteq A$ means that the requirements $A$ allows the implementation $B$, or $B$ refines $A$. While $\text{domain}(B) \supseteq \text{domain}(A)$ means that the implementation should include behaviours of all cases that are specified in the requirements. The second restriction solves the problem of “empty relation refines everything”, raised in [49].

5.3 Verify the Description of the Behaviour of a Network

The black-box behavior of the system denoted as $CIT$; while a TFM description of the white-box behaviors of the network is denoted as $CIT_d$. In this section we show how to check whether the white-box behavioral description, $CIT_d$, conforms to the black-box behavioral description, $CIT$.

Definition 5.2: syntax match between $CIT_d$ and $CIT$

Given a system requirements TFM specification $CIT = (I, O, F)$ and a network $NW = (\Delta, \Sigma, \Omega, \mathcal{H})$ of n components to satisfy the requirements. The derived TFM description $CIT_d = (\Sigma, \Omega, G)$ of the network will syntactically conform to $CIT$ if the inputs, outputs and their relations match in name and type. More specifically:

- The variables in $I$ and $\Sigma$ match in name and type, $I \subseteq \Sigma$
- The variables in $O$ and $\Omega$ match in name and type, $O \subseteq \Omega$
- For any $(f_\omega)_{\omega \in O}$ there must be a $(g_\omega)_{\omega \in \Omega}$.

Def. 5.2 is called syntactical conformance since it considers the match of the variables of the two documents only in name and type. For semantic match we need to define the following function to extract the triples of the matched variables from a network trace in order to get a system trace.

Definition 5.3: relation $\pi_0$ between network traces and system traces

Based on Def. 5.2 ($I \subseteq \Sigma$ and $O \subseteq \Omega$), we can define an abstraction relation that maps a network trace to a system trace, which keeps the system input/output variables (those...
matched with the variables in $I \cup O$ and their values only, and removing the
interconnection details of the network. This relation, denoted as $\pi_0$, is expressed as bellow:

$$\pi_0: <\text{network traces}> \rightarrow <\text{system traces}>$$

$$\pi_0 = \{(T_d, T) | T_d \in T_d \land T \in T \land \forall t, E(T)(t) = e\}$$

where $e = \{(v, p, q) | (v, p, q) \in E(T_d)(t) \land v \in (I \cup O)\}$ is the set of triples whose variable is in
the data structure of the system.

**Remark 5.1** $\pi_0$ is a total function because $T_d$ is the domain space, and according to the
definition, for every network trace $T_d$ in $T_d$, there is one and only one system trace $T$,
such that $(T_d, T) \in \pi_0$.

**Theorem 5.1:**

*Given a system TFM specification $CIT = (I, O, F)$ and a network $NoC = (\Delta, \Sigma, \Omega, \mathcal{R})$
of $n$ components to satisfy this given requirement. The calculated TFM description $CIT_d = (\Sigma, \Omega, G)$ of the network will conform to $CIT$ if the two documents match
syntactically as defined in Def. 5.2 and $\forall \omega \in O, g_\omega \subseteq \pi_0 \circ f_\omega \land \text{range}(\pi_0) \supseteq T$.

**Proof:**

Here $g_\omega$ is the implementation, and $f_\omega$ is the specification. $g_\omega$ and $f_\omega$ cannot be
compared directly as shown in Def. 5.1 since the domain space of $g_\omega$ and $f_\omega$ are
different. However,

relation $\pi_0^{-1}g_\omega: <T> \times \text{Type}(\omega)$ and

relation $f_\omega: <T> \times \text{Type}(\omega)$ have the same domain space. According to Def. 5.1, we
have: $\pi_0^{-1}g_\omega \subseteq f_\omega$ and $\text{domain}(\pi_0^{-1}g_\omega) \supseteq \text{domain}(f_\omega)$

$$\pi_0^{-1}g_\omega \subseteq f_\omega$$

$$\pi_0 \circ \pi_0^{-1}g_\omega \subseteq \pi_0 \circ f_\omega$$

and $\pi_0$ is total so $I_T \subseteq \pi_0 \circ \pi_0^{-1}$

$$I_T \circ g_\omega \subseteq \pi_0 \circ f_\omega$$

$$g_\omega \subseteq \pi_0 \circ f_\omega$$

$$\text{domain}(\pi_0^{-1}g_\omega) \supseteq \text{domain}(f_\omega)$$

$$\text{domain}(\pi_0^{-1}) \supseteq \text{domain}(\pi_0^{-1}g_\omega) \supseteq \text{domain}(f_\omega)$$

$$\text{domain}(\pi_0^{-1}) \supseteq \text{domain}(f_\omega).$$
domain(π₀⁻¹) = range(π₀) and domain(f_ω) = T \Rightarrow range(π₀) \supseteq T

From the above it is proved that ∀ω ∈ O, g_ω ⊆ π₀ ° f_ω \land range(π₀) \supseteq T, therefore CIT_d conforms to CIT.

This completes the proof.

The formula g_ω ⊆ π₀ ° f_ω describes another triangle “commuting diagram” shown in Fig 5.1. For each output variable ω in O there is such a diagram that commutes for every network trace. For any network trace T_d in T_d, the corresponding values of ω in the network are constrained by relations g_ω, i.e., (T_d, ω) ∈ g_ω. On the other hand, the trace T_d can be first mapped to system trace π₀(T_d) = T and T ∈ T, then the value of ω is constrained by f_ω, i.e., (T, ω) ∈ f_ω. This means that for a network trace, the value of ω can be obtained through the pale gray route (g_ω) or the dark gray route (π₀ ° f_ω) while g_ω ⊆ π₀ ° f_ω must hold. Two dark gray lines and two pale gray lines indicate that g_ω and f_ω are relations.

If f_ω is a function, g_ω must be a function and the formula becomes g_ω = π₀ ° f_ω. For an output variable ω, any network trace in T_d should be mapped to the value of ω through either route π₀ ° f_ω or route g_ω.

The condition range(π₀) \supseteq T says that the implementation g_ω, whose domain is also T_d (so is the domain of π₀), includes behaviors of all cases that are specified in the requirements, whose domain is the set of system traces T.

Combining the “commuting diagram” in Fig 5.1 and the “commuting diagram” in Fig. 4.3 together yields the diagram as in Fig 5.2 shown.
For any variable $\omega$ in $O$, there is a defined component trace relation $f_o$ for variable $o$, such that $o \rightarrow \omega \in \mathcal{R}$ and $\pi_k^o f_o \subseteq g_{o^w}$, where $\pi$ is the projection function from network traces to component traces in $C_k$. For every variable $\omega$ in the set of system output variables there exists such a commuting diagram. For a network trace $T_d$ in $T_d$, the output values of $o$ for trace $T_d$ are restricted by relation $\pi_k^o f_o$, i.e., $(T_d, o) \in \pi_k^o f_o$, while the output values of $\omega$ for trace $T_d$ can be restricted by the two relations $\pi_0^o f_{o^w}$ and $g_{o^w}$, i.e., $(T_d, \omega) \in \pi_0^o f_{o^w}$, $(T_d, \omega) \in g_{o^w}$.

Fig. 5.2: The Second Commuting Diagram for Conformance Checking

And these three relations satisfy the following formula (see the proof for Lemma 5.1):

$$\pi_k^o f_o \subseteq g_{o^w} \subseteq \pi_0^o f_{o^w}$$

If $f_{o^w}$ is a function, then $f_o$ and $g_{o^w}$ must be functions and the three routes are equivalent, i.e.

$$\pi_k^o f_o = g_{o^w} = \pi_0^o f_{o^w}$$

For each system output variable, there is such a diagram that commutes for all network traces in $T_d$.

**Lemma 5.1:**

Given a system requirements TFM specification $CIT = (I, O, F)$ and a network $NW = (\Delta, \Sigma, \Omega, \mathcal{R})$ of $n$ components satisfy this requirement. If the components’ TFM descriptions are $CIT_{k=1...n} = (I_k, O_k, F_k)$, then $CIT_{k=1...n}$ and the network description...
together conform to CIT if the two documents match syntactically as in Def. 5.2 and ∀ω∈O, there is an o ∈ Ok and o→ω ∈ℜ s.t. πk°f_o ⊆ π0°f_o ∧ range(π_o) ⊇ T. πk is the projection function from network traces T_d to component traces T_k in component C_k.

**Proof:**

From πk°f_o ⊆ g_o (Theorem 4.1) and g_o ⊆ π0°f_o (Theorem 5.1) it can be seen πk°f_o ⊆ π0°f_o, range(π_o) ⊇ T is proved in the proof of Theorem 5.1.

Lemma 5.1 says that the system requirements TFM specification is an abstraction of behavioral description of the network and the behavioral description of the network is an abstraction of the behavioral description of the interconnected components.

Since in Chapter 3 we required delay-free loop free design and in Chapter 4 it is proved that the components will work together, terminate and lead to a defined behavior (Lemma 4.3), the correctness of such a software design discussed in this chapter is total correctness.

Theorem 5.1 and Lemma 5.1 describe when a set of software design documents conform to the requirements specification. One trace relation in the requirement may need to be compared with a set of trace relations from different component that are supposed to implement the required relation. The comparison is between a composed complex relation and a simple relation. There may be problems such as:

- It is not so easy to come up with a composed tabular expression.
- Even if we can get one tabular expression, it may be too complicated to check whether the two tabular expressions are the same or whether there are any implications between them.

In the next chapter we will investigate how to use this approach to verify a software design by three case studies.
Chapter 6

Case Studies

We have conducted three case studies in this chapter: an RS (Reset Set) flip-flop, a temperature monitor and a flight reservation system, on one hand to develop, improve and test our approach, and on the other hand to demonstrate the way and the procedure of using the method. In each application we focus on some special issues. As long as a system and the constructing components are specified or described as functions of history of inputs and outputs, we can apply the approach to these documents.

For the RS flip-flop example, we will demonstrate two versions of designs. The contribution of this case study lies in defining and formalizing the consistency of the network, and developing the derivation relation and the conformance relation.

In Section 6.1 we will show how to get network traces (predicates on traces) of the RS by using Def. 4.1, which can be used directly only when the event descriptor sets can be enumerated. For this same example we will also try to use the derivation relation defined in Def. 4.3. The complete procedure of determining the behavior of the network and comparing it with the system requirements will be illustrated in Appendix 1. We can see that Def. 4.1 is the basis for defining the relations in Def. 4.3. The same results can be achieved through both definitions. Def. 4.3 can be applied to general cases, such as the second and third case studies in this chapter.

To test and demonstrate the use of the developed theory, we designed two other applications from system requirements analysis, functional decomposition, component interface descriptions, and a network description. By applying the calculation in Def.4.3 and the conformance checking relation to these examples we try to express those aspects that are necessary for a general understanding to the approach. We will also discuss some interesting issues such as program calls and explain how we can deal with a program call as nothing special but another shared variable in the network description.
6.1 RS Flip-flop

An RS flip-flop, one of whose implementations is shown in Fig 6.1, has two input variables, R and S, and two output variables, Q and \( \overline{Q} \). The output Q is always the value that is 'stored' in the flip-flop. The input variables specify whether the value 0 or 1 should be stored in the flip-flop. If S is 1, the state of the flip-flop is 'set' to 1. If R is 1, the state of the flip-flop is 'reset' to 0. If S and R are both 0, the state of the flip-flop does not change, i.e., keeps the immediately previous value; the bit that is previously stored in the flip-flop remains there. R and S are not allowed to be 1 at the same time. It does not make sense to both set and reset the flip-flop at the same time anyway.

The following truth table describes how the circuit should behave. In addition to the input values of S and R, the value 'stored' in the flip-flop also affects the output in the next clock cycle.

<table>
<thead>
<tr>
<th>S</th>
<th>R</th>
<th>Q</th>
<th>( Q_{\text{next}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>*</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>*</td>
<td>x</td>
</tr>
</tbody>
</table>

Here "*" means any value, 0 or 1. "x" means undefined.

**6.1.1 The Requirements TFM Specification of the RS Flip-flop**

In a synchronous system, all events happen at the same clock rate based on a single reference clock. Assuming events happen at time instants \( t, t+c, t+2c \), there is no event between \( t \) and \( t+c \). In TFM, the before value of a variable in an event at time \( t \) equals to the after value of the same variable in an event at time \( t-c \). If an event at time \( t \) is the last event, we should have ‘\( R(r(T)) = R’(r(p(T))\). The value of \( Q_{\text{next}} \) at time \( t \) is decided by looking at the values of S, R, and Q at time \( t-c \), which means that Q(T) depends on ‘\( R(r(T)) \), ‘S(r(T)) and ‘Q(r(T)).

The truth table can be translated into TFM specifications as follows.
### INPUT VARIABLES:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>S</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>R</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

### OUTPUT VARIABLES:

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

### OUTPUT FUNCTIONS:

\[
Q(T) =
\begin{array}{|c|c|}
\hline
T = & 0 \\
\hline
'S(r(T)) = 0 \land Q(p(T)) = 0 & 0 \\
'R(r(T)) = 0 \land Q(p(T)) = 1 & 1 \\
'S(r(T)) = 0 \land 'R(r(T)) = 1 & 0 \\
'S(r(T)) = 1 \land 'R(r(T)) = 0 & 1 \\
'S(r(T)) = 1 \land 'R(r(T)) = 1 & 1 \\
\hline
\end{array}
\]

### 6.1.2 The Network of Components

Fig. 6.1 illustrates one of the many RS implementations. Here an OR gate and an AND gate are used, while in most text books an RS is normally constructed from a cross-coupled NOR or NAND gates by feeding the outputs of two NOR/NAND back to the inputs of another NOR/NAND gates.

The network of components is described as the following according to Def. 3.1 and 3.8.

\[
\text{NoC} = (\Delta, \Sigma, \Omega, \mathcal{R})
\]

\[
\Delta = \{\text{NOT, AND, OR}\}
\]

\[
\Sigma = \{R, S\}\]
\[ \Omega = \{Q\} \]
\[ \Gamma = \{\gamma_1, \gamma_2, \gamma_3\} \]
\[ \mathcal{R} = \{v_1 \rightarrow \gamma_1 \rightarrow i_{22}, v_2 \rightarrow \gamma_2 \rightarrow i_{32}, v_3 \rightarrow \gamma_3 \rightarrow i_{21}, \]
\[ R \rightarrow i_{11}, S \rightarrow i_{31}, \]
\[ v_3 \rightarrow Q\} \]

\[ CIT_{\text{NOT}} = (I_1, O_1, F_1) \]
\[ I_1 = \{i_{11}\} \]
\[ O_1 = \{v_1\} \]
\[ D_1 = \{i_{11} \rightarrow v_1\} \]
\[ F_1 = \{v_1(T)\} \]

\[ CIT_{\text{AND}} = (I_2, O_2, F_2) \]
\[ I_2 = \{i_{21}, i_{22}\} \]
\[ O_2 = \{v_2\} \]
\[ D_2 = \{i_{21} \rightarrow v_2, i_{22} \rightarrow v_2\} \]
\[ F_2 = \{v_2(T)\} \]

\[ CIT_{\text{OR}} = (I_3, O_3, F_3) \]
\[ I_3 = \{i_{31}, i_{32}\} \]
\[ O_3 = \{v_3\} \]
\[ D_3 = \{i_{31} \rightarrow v_3, i_{32} \rightarrow v_3\} \]
\[ F_3 = \{v_3(T)\} \]

6.1.3 Is the Network Complete and Consistent?

From the truth tables of the gates (AND, OR, NOT) it can be seen the devices are supposed to be delay-free. Then we may find that the network is completely connected (Def. 3.2) but not consistent (Def. 3.11), because there is a delay-free loop in the network. Since the output of an RS flip-flop is the value ‘stored’ in the flip-flop and the output value is expected in the next clock cycle after the inputs are set or reset, there should be one bit memory in the circuit. But from the design in Fig. 6.1, and the other implementations such as those by NOR or NAND, the delay effect is not explicitly modelled in the circuit. If each gate is considered as delay free, as suggested in the truth table, the circuit may be unstable.
According to this analysis we have modelled the delay effect explicitly as a delay device in the following diagram. With this delay device we have got a consistent network; and using the formulas in Def. 4.3 we can calculate a TFM description of the network and then compare it with the requirements specification of the RS flip-flop.

The network description of the new network is the following:

$$NT = (\Delta, \Sigma, \Omega, \gamma)$$

$$\Delta = \{\text{NOT, AND, OR, D}\}$$

$$\Sigma = \{R, S\}$$

$$\Omega = \{Q\}$$

$$\Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$$

$$\gamma = \{v_1 \rightarrow \gamma_1 \rightarrow i_{22}, v_2 \rightarrow \gamma_2 \rightarrow i_{32}, v_3 \rightarrow \gamma_3 \rightarrow i_{41}, v_4 \rightarrow \gamma_4 \rightarrow i_{21}\}$$

$$R \rightarrow i_{11}, S \rightarrow i_{31}, v_4 \rightarrow Q\}$$

$CIT_{\text{NOT}}, CIT_{\text{AND}}, CIT_{\text{OR}}$ are the same as described for Fig 6.1.

$$CIT_D = (I_4, O_4, F_4)$$

$$I_4 = \{i_{41}\}$$

$$O_4 = \{v_4\}$$

$$D_4 = \{\}$$

$$F_4 = \{v_4(T)\}$$
6.1.4 The TFM Descriptions of the Constructing Components

According to the truth tables of the constituting components, we can specify their behaviours by TFM.

Def. 4.1 describes how to combine two component traces into a network trace by uniting component event descriptors at each time instant. Although the combination of event descriptors should be done on the whole time space of the traces, we start from checking the value consistency in state changes from two components and then combine the consistent component event descriptors into a network event descriptor. As clarified in Section 4.2.1, we need some special classes of component event descriptors that cover all possible cases. In many systems, similar events/states may repeatedly happen at different times, which means that the before and after values of all variables are the same except for the time. This set of events/states can be described by a class of event descriptors or dummy event descriptors. With these descriptor sets we can combine them into network event descriptor sets if the component state changes can possibly happen at the same time. There is a class for every combination of before and after values of the variables, and each class contained descriptors for all time instants.

This example is suitable for studying the two components with looped connections starting from Def. 4.1 and it helps to develop and formalize the conformance relations. The enumeration of the set of possible component event descriptors makes the determination of the network behaviour and the conformance comparison straightforward.

The following are the TFM descriptions of the constructing components in Fig. 6.2.

1. NOT gate:

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i11</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>
OUTPUT FUNCTIONS:
\[ v_1(T) = \begin{cases} 0 & T = _ \_ \\ \neg i_{11}'(r(T)) & T \neq _ \_ \end{cases} \]

Possible event descriptor sets:

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>'i_{11} 'i_{11} 'v_1 'v_1 for ce'</th>
<th>'i_{11} 'i_{11} 'v_1 'v_1 for ce'</th>
<th>Name of ce'</th>
</tr>
</thead>
<tbody>
<tr>
<td>1e_{12}</td>
<td>0 1 1 0</td>
<td>1 1 0 0</td>
<td>1e_{30}</td>
</tr>
<tr>
<td>1e_{21}</td>
<td>1 0 0 1</td>
<td>0 0 1 1</td>
<td>1e_{03}</td>
</tr>
</tbody>
</table>

For the convenience of manually checking the value consistency, we make the following naming conventions for the event descriptor sets: the first digit indicates in which component the event descriptor is. The footnote of each event descriptor is a series of digits with each digit indicating the value changes of a variable. See the explanation in Fig. 6.3. For instance, from the name 1e_{12} we know it is from component C_{1}, and the two variables i_{11} and v_1 are changed from 0 to 1 and 1 to 0 respectively.

\[ \text{00 – 0} \quad \text{This is the correspondence between before/after values of a variable and the footnote of the event descriptor. The two digits on the left side of the dash is the before and after values of a variable, the digit on the right side of the dash is the corresponding footnote for that variable in an event descriptor.} \]

2. AND gate

OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v_2</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

INPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_{21}</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>i_{22}</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

OUTPUT FUNCTIONS:
\[ v_2(T) = \begin{cases} 0 & T = _ \_ \\ 1 & (T \neq _ \_) \land (i_{21}'(r(T)) = i_{22}'(r(T)) = 1) \\ 0 & (T \neq _ \_) \land \neg (i_{21}'(r(T)) = i_{22}'(r(T)) = 1) \end{cases} \]
### Possible event descriptor sets:

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>'i$<em>{21}$' 'i$</em>{22}$' 'v$_{2}$' for ce'</th>
<th>'i$<em>{21}$' 'i$</em>{22}$' 'v$_{2}$' for ce'</th>
<th>Name of cet</th>
</tr>
</thead>
<tbody>
<tr>
<td>2e$_{010}$</td>
<td>0 0 0 1 0 0</td>
<td>0 0 1 1 0 0</td>
<td>2e$_{030}$</td>
</tr>
<tr>
<td>2e$_{020}$</td>
<td>0 0 1 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>2e$_{000}$</td>
</tr>
<tr>
<td>2e$_{100}$</td>
<td>0 1 0 0 0 0</td>
<td>1 1 0 0 0 0</td>
<td>2e$_{000}$</td>
</tr>
<tr>
<td>2e$_{111}$</td>
<td>0 1 0 1 0 1</td>
<td>1 1 1 1 1 1</td>
<td>2e$_{333}$</td>
</tr>
<tr>
<td>2e$_{120}$</td>
<td>0 1 1 0 0 0</td>
<td>1 1 0 0 0 0</td>
<td>2e$_{000}$</td>
</tr>
<tr>
<td>2e$_{131}$</td>
<td>0 1 1 1 0 1</td>
<td>1 1 1 1 1 1</td>
<td>2e$_{333}$</td>
</tr>
<tr>
<td>2e$_{200}$</td>
<td>1 0 0 0 0 0</td>
<td>0 0 0 0 0 0</td>
<td>2e$_{000}$</td>
</tr>
<tr>
<td>2e$_{210}$</td>
<td>1 0 0 1 0 0</td>
<td>0 0 1 1 0 0</td>
<td>2e$_{030}$</td>
</tr>
<tr>
<td>2e$_{222}$</td>
<td>1 0 1 0 1 0</td>
<td>0 0 0 0 0 0</td>
<td>2e$_{000}$</td>
</tr>
<tr>
<td>2e$_{232}$</td>
<td>1 0 1 1 1 0</td>
<td>0 0 1 1 0 0</td>
<td>2e$_{030}$</td>
</tr>
<tr>
<td>2e$_{233}$</td>
<td>1 1 0 1 0 1</td>
<td>1 1 1 1 1 1</td>
<td>2e$_{333}$</td>
</tr>
</tbody>
</table>

### 3. OR gate:

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v$_{3}$</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i$_{31}$</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>i$_{32}$</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

**OUTPUT FUNCTION:**

\[
v_{3}(T) =
\begin{cases}
0 & \text{if } T \neq _\_ (T \neq _\_ \land (i_{31}'(r(T)) = i_{32}'(r(T)) = 0)) \\
0 & \text{if } T \neq _\_ (T \neq _\_ \land \neg (i_{31}'(r(T)) = i_{32}'(r(T)) = 0)) \\
1 & \text{otherwise}
\end{cases}
\]

Possible event descriptors:

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>'i$<em>{31}$' 'i$</em>{32}$' 'v$_{3}$' for ce'</th>
<th>'i$<em>{31}$' 'i$</em>{32}$' 'v$_{3}$' for ce'</th>
<th>Name of cet</th>
</tr>
</thead>
<tbody>
<tr>
<td>3e$_{011}$</td>
<td>0 0 0 1 0 1</td>
<td>0 0 1 1 1 1</td>
<td>3e$_{033}$</td>
</tr>
<tr>
<td>3e$_{022}$</td>
<td>0 0 1 0 1 0</td>
<td>0 0 0 0 0 0</td>
<td>3e$_{000}$</td>
</tr>
<tr>
<td>3e$_{101}$</td>
<td>0 1 0 0 0 1</td>
<td>1 1 0 0 1 1</td>
<td>3e$_{303}$</td>
</tr>
<tr>
<td>3e$_{111}$</td>
<td>0 1 0 1 0 1</td>
<td>1 1 1 1 1 1</td>
<td>3e$_{333}$</td>
</tr>
<tr>
<td>3e$_{123}$</td>
<td>0 1 1 0 1 1</td>
<td>1 1 0 0 1 1</td>
<td>3e$_{303}$</td>
</tr>
<tr>
<td>3e$_{133}$</td>
<td>0 1 1 1 1 1</td>
<td>1 1 1 1 1 1</td>
<td>3e$_{333}$</td>
</tr>
<tr>
<td>3e$_{202}$</td>
<td>1 0 0 0 1 0</td>
<td>0 0 0 0 0 0</td>
<td>3e$_{000}$</td>
</tr>
<tr>
<td>3e$_{213}$</td>
<td>1 0 0 1 1 1</td>
<td>0 0 1 1 1 1</td>
<td>3e$_{033}$</td>
</tr>
<tr>
<td>3e$_{222}$</td>
<td>1 0 1 0 1 0</td>
<td>0 0 0 0 0 0</td>
<td>3e$_{000}$</td>
</tr>
<tr>
<td>3e$_{233}$</td>
<td>1 0 1 1 1 1</td>
<td>0 0 1 1 1 1</td>
<td>3e$_{033}$</td>
</tr>
</tbody>
</table>
4. Delay device:

**OUTPUT VARIABLES:**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>v4</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES:**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i41</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
</tbody>
</table>

There is one input and one output variable for a delay device and the output value is the input value delayed by one event. The after value of the output variable at an event is the after value of the input variable at the previous event. Since the before value of a variable in an event always equals to the after value of the same variable in the previous event, the delay effect can be expressed in a single event. In any event the after value of the output variable is always the before value of the input variable (\(i_{41}(e) = v_4(e)\)). From this observation the output can be specified as:

**OUTPUT FUNCTION:**

\[
v_4(T) = \begin{cases} 0 & T = \_ \\ i_{41}(r(T)) & T \neq \_ \end{cases}
\]

Possible event descriptors:

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>‘i41’ i41 i41 ’v4’ v4 for ce^l</th>
<th>‘i41’ i41 ’v4’ v4 for ce^r</th>
<th>Name of ce^r</th>
</tr>
</thead>
<tbody>
<tr>
<td>4e10</td>
<td>0 1 0 0</td>
<td>1 1 0 0</td>
<td>4e30</td>
</tr>
<tr>
<td>4e02</td>
<td>0 0 1 0</td>
<td>0 0 0 0</td>
<td>4e00</td>
</tr>
<tr>
<td>4e12</td>
<td>0 1 1 0</td>
<td>1 1 0 0</td>
<td>4e30</td>
</tr>
<tr>
<td>4e23</td>
<td>1 0 1 1</td>
<td>0 0 1 1</td>
<td>4e03</td>
</tr>
<tr>
<td>4e31</td>
<td>1 1 0 1</td>
<td>1 1 1 1</td>
<td>4e33</td>
</tr>
<tr>
<td>4e21</td>
<td>1 0 0 1</td>
<td>0 0 1 1</td>
<td>4e03</td>
</tr>
</tbody>
</table>

**6.1.5 Calculate the TFM Descriptions of the Network**

Knowing the TFM interface descriptions of each device, using Def. 4.1 we can calculate the set of network event descriptors and hence the behavioural TFM description of the flip-flop.
1) Calculate \( CIT_{12} \) from \( CIT_{\text{NOT}} \) and \( CIT_{\text{AND}} \),
2) Calculate \( CIT_{123} \) from \( CIT_{12} \) and \( CIT_{\text{OR}} \),
3) Calculate \( CIT_{1234} \) from \( CIT_{123} \) and \( CIT_{D} \),
4) Calculate \( CIT \) from \( CIT_{1234} \) and \( CIT_{\text{NOT}} \),

Finally we can get the TFM description \( CIT \) for the network of all components.

Since we consider full synchronous systems, the consistency checking condition \( \rho_{\text{consistentAt}}(t) \) will include the checking of whether there are component state changes at the same time and whether the two event descriptors are consistent in values of connected variables.

Here we explain the conventions and rules in the calculation of \( CIT_{12} \). The other steps can be seen from Appendix 1.

\[
U(<T_1> \times <T_2> \times <T>) \equiv \text{T[2]}
\]

\[
\begin{array}{cccccccccccccccc}
000 & 020 & 100 & 111 & 120 & 131 & 200 & 210 & 222 & 232 & 311 & 322 & 000 & 030 & 300 & 333 \\
\hline
12 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
21 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
03 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
30 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
\end{array}
\]

\( T[1] \)

This is a normal function table, the numbers \( ij \) in grid \( T[1] \) and \( kmn \) in grid \( T[2] \) are the subscript in predicates, such as \( r(T_1) = 1e_{ij} \) and \( r(T_2) = 2e_{kmn} \). To save space, only the subscripts are listed.

The subscript \( ij \) is used to indicate the value change of \( i_{11} \) and \( v_1 \) and the subscript \( kmn \) is used to indicate the change of \( i_{21} \), \( i_{22} \) and \( v_2 \). From the network rules and the consistency conditions we should have \( j=m \). \( (i_{22} \leftarrow \gamma_1 \leftarrow v_1 \Rightarrow 'i_{22} = 'v_1, i_{22}' = v_1' \Rightarrow j=m) \)

All the consistent traces should satisfy the condition \( j=m \) for the corresponding event descriptors.

The ‘1’s in grid \( T[0] \) is used to indicate that there is a consistent trace because the corresponding predicates in \( T[1] \) and \( T[2] \) are evaluated to true.

The table above is read as:

\[
\forall t, E(T_1)(t) = 1e_{ij}^{1} \land E(T_2)(t) = 2e_{kmn}^{1} \land j=m
\]

\[
\Rightarrow U(T_1, T_2) = T_{12} \text{ s.t. } E(T_{12})(t) = e_{ijkmn}^{1}
\]
It can be converted into:
\[ r(T_1) \in 1e_{ij} \land r(T_2) \in 2e_{kmn} \land j = m \]
\[ \Rightarrow U(T_1, T_2) = T_{12} \text{ s.t. } r(T_{12}) \in e_{ijkmn} \]

Since \( j = m \), we omit one of the digits, using \( e_{ijkn} \) to describe the network event descriptor. In the sequel this convention is always applied.

From the above relation table we can get the possible event descriptor sets for the network consisting of a NOT and an AND gate. There is a network event descriptor when there is a “1” in grid \( T[0] \). After getting all the event descriptor sets for the network, by doing operation \( e' \) on each event descriptor \( e \), the dummy event descriptor sets (those descriptors of which all digits in the footnotes are “0”s and “3”s) can be obtained. As described in Chapter 4, obtaining the \( e' \) is necessary only for the intermediate calculation. In the final result this is not needed.

Possible event/dummy event descriptor sets for the network of NOT and AND gates:

<table>
<thead>
<tr>
<th>Name of ( e ) set</th>
<th>( i_1 )</th>
<th>( \gamma_1 )</th>
<th>( i_{21} )</th>
<th>( v_2 ) for ( ce' )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( e_{1200} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( e_{1210} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( e_{1222} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{1232} )</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{2100} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( e_{2111} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( e_{2120} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( e_{2131} )</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0311} )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0322} )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0301} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( e_{0302} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0309} )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0331} )</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( e_{0000} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( e_{0031} )</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

6.1.6 Is the TFM Description of the Behaviour of the Network Correct?

The complete deriving procedure and the final results are in Appendix 1. Here we list the final results only for the comparison with the requirements.
Input/output variables of the network of components including interconnectors:

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_1 (R)</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>i_3 (S)</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>v_5 (−Q)</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>γ_1</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>γ_2</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>γ_3</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>γ_4 (Q)</td>
<td>&lt;boolean&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

The possible network event descriptors for the complete network are listed in Table 6.1.

Table 6.1: The possible network event descriptor sets

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>i_11(R)</th>
<th>γ_1</th>
<th>i_21</th>
<th>γ_2</th>
<th>i_31(S)</th>
<th>γ_3</th>
<th>v_4(Q) for ce</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_1200110</td>
<td>0 1 1 0 0 0 0 0 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1200000</td>
<td>0 1 1 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1210221</td>
<td>0 1 1 0 1 0 0 1 0 1 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1210331</td>
<td>0 1 1 0 0 1 0 0 1 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1232023</td>
<td>0 1 1 0 1 1 1 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1232133</td>
<td>0 1 1 0 1 1 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1232233</td>
<td>0 1 1 0 1 1 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_1233333</td>
<td>0 1 1 0 1 1 1 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2100110</td>
<td>1 0 0 1 0 0 0 0 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2100000</td>
<td>1 0 0 1 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2111231</td>
<td>1 0 0 1 0 1 0 1 1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2111331</td>
<td>1 0 0 1 0 1 0 1 1 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2120112</td>
<td>1 0 0 1 1 0 0 0 0 1 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2120002</td>
<td>1 0 0 1 1 0 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2131233</td>
<td>1 0 0 1 1 0 1 1 0 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_2131333</td>
<td>1 0 0 1 1 0 1 1 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3011231</td>
<td>0 0 1 1 0 1 0 1 1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_30311331</td>
<td>0 0 1 1 0 1 0 1 1 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3010221</td>
<td>1 1 0 0 0 1 0 0 1 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3010331</td>
<td>1 1 0 0 0 1 0 0 1 1 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3020112</td>
<td>1 1 0 0 1 0 0 0 0 1 0 1 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3020002</td>
<td>1 1 0 0 1 0 0 0 0 0 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3000110</td>
<td>0 0 1 1 0 0 0 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3033333</td>
<td>0 0 1 1 1 1 1 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3033233</td>
<td>0 0 1 1 1 1 1 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3000100</td>
<td>1 1 0 0 0 0 0 0 1 0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_3030223</td>
<td>1 1 0 0 1 1 0 1 0 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
As described in Section 4.1, this is an internal view of the system, i.e. the values of the internal variables are all observable. By observing the possible event descriptor sets in Table 6.1, the predicates on event descriptors can be written as the following and compared with the requirements specification in Section 6.1.1.

1) Requirements for \( R=0 \) and \( S=0 \):
\[
\begin{align*}
\text{and Q} & \quad R(r(T)) = 8 S(r(T)) = 0 \land Q(p(T)) = 0 \Rightarrow Q(T) = 0 \\
\text{and Q} & \quad R(r(T)) = 8 S(r(T)) = 0 \land Q(p(T)) = 1 \Rightarrow Q(T) = 1 \\
\end{align*}
\]
In the implementation, from Table 6.1 it can be seen that for all possible traces in the TFM description:
\[
\begin{align*}
\text{and Q} & \quad R(r(T_d)) = 8 S(r(T_d)) = 0 \land Q(p(T_d)) = 0 \Rightarrow Q(T_d) = 0 \\
\text{and Q} & \quad R(r(T_d)) = 8 S(r(T_d)) = 0 \land Q(p(T_d)) = 1 \Rightarrow Q(T_d) = 1, \text{ consistent with requirement.} \\
\end{align*}
\]

2) Requirements for \( R = 0 \) and \( S=1 \):
\[
\begin{align*}
\text{and Q} & \quad R(r(T)) = 0 \land S(r(T)) = 1 \Rightarrow Q(T) = 1 \\
\end{align*}
\]
In the implementation, for all traces in the description,
\[
\begin{align*}
\text{and Q} & \quad R(r(T_d)) = 0 \land S(r(T_d)) = 1 \Rightarrow Q(T_d) = 1, \text{ consistent with requirement.} \\
\end{align*}
\]

3) Requirements for \( R=1 \) and \( S=0 \):
\[
\begin{align*}
\text{and Q} & \quad R(r(T)) = 1 \land S(r(T)) = 0 \Rightarrow Q(T) = 0 \\
\end{align*}
\]
The implementation, for all traces in the description,
\[
\begin{align*}
\text{and Q} & \quad R(r(T_d)) = 1 \land S(r(T_d)) = 0 \Rightarrow Q(T_d) = 0, \text{ consistent with the requirement.} \\
\end{align*}
\]

4) Requirements for \( R=1 \) and \( S=1 \):
\[
\begin{align*}
\text{and Q} & \quad R(r(T)) = 1 \land S(r(T)) = 1 \Rightarrow Q \text{ is undefined} \\
\end{align*}
\]
The implementation, for all traces in the description,
\[
\begin{align*}
\text{and Q} & \quad R(r(T_d)) = 1 \land S(r(T_d)) = 1 \Rightarrow Q(T_d) = 1 \\
\end{align*}
\]
If “undefined” means “can be anything”, the implementation satisfies the specification.

It is proved that the circuit in Fig. 6.2 is a correct design for the requirement of an RS flip-flop and the TFM description of the behavior of the network is correct. The same result is obtained in Appendix 1, by using the symbolic calculation as defined in Def. 4.3 and Def. 4.4.
6.2 Temperature Monitor System

A Temperature Monitor System (TMS) has one noisy sensor that measures temperature. Because of the noise, it delivers several values to a display. We now consider the case where there are three output variables: the currently read temperature, the average temperature over the last 20 minutes and that over the past hour.

6.2.1 The Goal of the System and its Requirements Specification

As shown in Fig 6.4, the inputs are the current temperature and a clock tick that is a periodic pulse used as a trigger to read the temperature sensor and to refresh the display. The outputs are the currently read temperature, the average temperature of the past 20 minutes and the average temperature of the past one hour.

![Fig. 6.4: The Black Box of a Temperature Monitor System](image)

Considering that there is some time delay needed for temperature storage and average calculation, the system may need a delay of three clock ticks for displaying temperatures. If every clock tick allows an event, the displayed “current temperature” is the value read three clock ticks ago.

Alternatively, the clock input “c” can be replaced by a PGM variable. The temperature sampling process and display refreshing procedure are controlled by program calls. The specifications of the component TFM descriptions in the corresponding networks, the deriving procedure of the behaviours of the networks, and the conformance comparisons are all illustrated in Appendix 2.

6.2.2 The Network of Components for the Task

The temperature monitor system can be decomposed into five components: sensor reader, temperature storage, average calculator, temperature display, and scheduler.
Sensor Reader (SR) reads values from the temperature sensor at each sampling time instant and converts them into temperature. It hides the secret of the way of communicating with the sensor and how to convert the sensor reading values into temperature.

Temperature Storage (TS) receives the values of the temperature from the SR and stores them. It provides current and previous temperature. It hides the secret of the format for storing the data.

Average Calculator (AV) reads temperatures from the SR and calculates the average values, and sends them to display. It hides the secret of the calculating algorithm, for example how to take advantage of iterative calculations.

Temperature Display (TD) gets the average and the current temperatures from the AV and displays them on the screen. It hides the secret of displaying format.

The Scheduler has a clock tick as input. It periodically and interleavably invokes the SR and the TD to read temperatures and to refresh the display screen of the temperatures. It hides the secret of the frequencies of invoking the temperature sensor reader and refreshing the display device, and the sequence of doing these.

The network of components implementing the temperature monitor is shown in Fig 6.5. The description of the component network is as the following:

\[ NT = (\Delta, \Sigma, \Omega, \Gamma) \]
\[ \Delta = \{TS, AV, Scheduler, SR, TD, Unifier\} \]
\[ \Sigma = \{PGM, tp\} \]
\[ \Omega = \{avg1, avg2, tmp\} \]
\[ \Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9, \gamma_{10}, \gamma_{11}, \gamma_{12}\} \]
\[ \mathcal{R} = \{\sigma_{41} \rightarrow \gamma_1 \rightarrow i_{11}, \sigma_{21} \rightarrow \gamma_2 \rightarrow i_{12}, \sigma_{22} \rightarrow \gamma_3 \rightarrow i_{13}, \]
\[ o_{11} \rightarrow \gamma_4 \rightarrow i_{21}, o_{12} \rightarrow \gamma_5 \rightarrow i_{22}, o_{13} \rightarrow \gamma_6 \rightarrow i_{23}, \]
\[ i_{51} \rightarrow \gamma_6 \rightarrow o_{23}, o_{24} \rightarrow \gamma_7 \rightarrow i_{52}, o_{13} \rightarrow \gamma_8 \rightarrow i_{53}, \]
\[ cp_{6} \rightarrow \gamma_9 \rightarrow PGM_1, \]
\[ cp_{51} \rightarrow \gamma_{10} \rightarrow PGM_2, \]
cp_{31} \rightarrow \gamma_{11} \rightarrow \text{PGM}_4,

\text{cp}_{32} \rightarrow \gamma_{12} \rightarrow \text{PGM}_5,

tp \rightarrow i_{41}, \text{PGM} \rightarrow i_{31},

o_{51} \rightarrow \text{avg}_1, o_{52} \rightarrow \text{avg}_2, o_{53} \rightarrow \text{tmp}\}

\text{Fig. 6.5: A Network of Components for the Temperature Monitor System}

\text{Legend:}
Thick arrows: network inputs/outputs
Thin arrows: interconnectors/component internal inputs/outputs
Boxes: components
Letters with subscript inside/outside the boxes: component input/output variables
Dashed lines inside the box: Delay-free dependencies

\text{CIT}_1 = (I_1, O_1, F_1)

I_1 = \{i_{11}, i_{12}, \text{PGM}_1\}

O_1 = \{o_{11}, o_{12}, o_{13}\}

D_1 = \{i_{12} \rightarrow o_{11}, i_{13} \rightarrow o_{12}\}

F_1 = (o_{11}(T), o_{12}(T), o_{12}(T))

\text{CIT}_2 = (I_2, O_2, F_2)
\(I_2 = \{i_{21}, i_{22}, i_{23}, \text{PGM}_2\}\)

\(O_2 = \{o_{21}, o_{22}, o_{23}, o_{24}, \text{cp}_2\}\)

\(D_2 = \{\}\)

\(F_2 = (o_{21}(T), o_{22}(T), o_{23}(T), o_{24}(T), \text{cp}_2(T))\)

\(CIT_3 = (I_3, O_3, F_3)\)

\(I_3 = \{i_{31}\}\)

\(O_3 = \{\text{cp}_{31}, \text{cp}_{32}\}\)

\(D_3 = \{\}\)

\(F_3 = (\text{cp}_{31}(T), \text{cp}_{32}(T))\)

\(CIT_4 = (I_4, O_4, F_4)\)

\(I_4 = \{i_{41}, \text{PGM}_4\}\)

\(O_4 = \{o_{41}, \text{cp}_4\}\)

\(D_4 = \{i_{41} \rightarrow o_{41}\}\)

\(F_4 = (o_{41}(T), \text{cp}_4(T))\)

\(CIT_5 = (I_5, O_5, F_5)\)

\(I_5 = \{i_{51}, i_{52}, i_{53}, \text{PGM}_5\}\)

\(O_5 = \{o_{51}, o_{52}, o_{53}, \text{cp}_{51}, \text{cp}_{52}\}\)

\(D_5 = \{i_{51} \rightarrow o_{51}, i_{52} \rightarrow o_{52}, i_{53} \rightarrow o_{53}\}\)

\(F_5 = (o_{51}(T), o_{52}(T), o_{53}(T), \text{cp}_{51}(T), \text{cp}_{52}(T))\)

6.2.3 Is the Network Complete and Consistent?

The TFM descriptions of the components in the network are presented in Appendix 2. It can be checked that this network is completely connected (Def. 3.2) and consistent (Def. 3.11). There are delayed loops between TS and AV but no delay-free loops. Therefore the constructed component should have a well defined behaviour and the description of the behaviour can be derived according to Def. 4.3. Before doing the calculation we need to discuss some issues such as program calls among components and the multiplexer/splitter component.
6.2.4 Program Calls

Program calls can be fully described by TFM [90] and the network rules. We treat program calls as connections between global variables. Program calls just place an identifier for a program on some variable that a component reads. A PGM variable is an input to a component with callable programs, and the program is executed if the value of the PGM variable is the name of that program. If a component calls programs from other components, it has one or more PGM variables as outputs. The PGM output variables can be specified by TFM in the same way as other variables. Then the programs calls can be expressed by connecting a PGM output variable to an input PGM variable.

For example, in the above network description, the network rule $cp_{32} \rightarrow \gamma_{12} \rightarrow PGM_5$ indicates that component “Scheduler”, who has an output PGM variable $cp_{32}$, calls programs in component “TD”, who has an input variable PGM$_5$, and the values such as “REFRESH” and “DISPLAY” will be passed from “Scheduler” to “TD”, which means that program “REFRESH” or “DISPLAY” in component TD is called.

Each component may have zero, one or more PGM input variables. If there are access programs in a component that are called from different components, we can choose to have more than one PGM input variables, each for a caller component to be connected with; or to have only one PGM input variable. Similarly, each component may have zero, one or more output PGM variables. If the component needs to call programs from different other components, we can choose to use one or more PGM output variables. Based on how many PGM variables used, some special components may need to be introduced to the network to keep its consistency.

6.2.5 Multiplexer and Splitter Component

Multiplexer

The completeness of a network requires that for every input variable, there is one and only one variable connected to it. If there is more than one other components calling programs within this component, and if there is only one PGM input variable in this component, a multiplexer component should be used to combine the flow of values from different components into one single output and then connect this output to the intended input. In the component interface TFM descriptions, the behaviour of the combined
program calls should be specified, if they behave differently from the case where they are called separately.

For example, the component “Unifier” in Fig 6.5 is to unify the three inputs to an output in order to be connected to the input variable PGM₁ in the temperature storage component. The values of these variables are power sets of the set of access programs of temperature storage component. It can be seen that the output value of cp₆ is the union of the three inputs to the unifier: cp₆(T) ≡ cp₂(T) ∪ cp₄(T) ∪ cp₅₁(T), where T is a network trace.

Splitter

The completeness condition requires that, if different values of one output variable are sent to different components, this output variable should be split into several output variables, each being connected to one component. The “Splitter” component in the next section (see Figure 6.6) is such an example. Although in the examples the multiplexer and unifier components are for unifying or splitting PGM variables, they can be used for other kinds of variable.

6.2.6 Calculate the Behavioural Descriptions of the Network

According to Def. 4.3, the formula \( g_{ωk} = π_k \circ f_{ωk} \) is used to extend all the component trace functions to network trace functions. To give an intuitive image about this point, the converting of one component trace function into a network trace function is shown here. The complete and detailed derivation procedure of this system is presented in Appendix 2.

\[
o_{41\delta}(T) = π_4 \circ o_{41} = o_{41}(π_4(T)) ≡ \\
\begin{array}{l}
π_4(T) = _-
\quad 0.0 \\
π_4(T) ≠ _-
\quad i_{41}'(r(π_4(T))) \\
\end{array}
\]

≡ (π₄(T) = _ - T = _ and π₄(T) ≠ _ - T ≠ _ by definition of cp₃₁₄)

\[
\begin{array}{|c|c|}
\hline
T & \text{PGM₄(r(T))} = \{\text{READ_SENSOR}\} \quad 0.0 \\
\hline
T \neq _ & \text{PGM₄(r(T))} = \{\} \\
\hline
\end{array}
\]

≡ (PGM₄ ← γ₁₁ ← cp₃₁ ∈ y ⇒ PGM₄(r(T)) = cp₃₁(r(T)) = cp₃₁(T) by Lemma 4.5)

\[
\begin{array}{|c|c|}
\hline
T & 0.0 \\
\hline
T \neq _ & \neg \text{tick(T)} \\
\hline
\end{array}
\]
After finishing these processes we will have a set of network trace functions that is comparable with the requirements TFM specifications.

The output functions of the network outputs after substituting the internal variables with their trace functions and the network rules are as follows.

\[ o_{51d}(T) = \pi^o_5 o_{51} = o_{51}(\pi^o_5(T)) \equiv \]

<table>
<thead>
<tr>
<th>L(T) &lt; 3</th>
<th>L(T) ≥ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \to tick(T) )</td>
<td>( \neg tick(T) )</td>
</tr>
<tr>
<td>( \to phas(T,0) )</td>
<td>( \neg phas(T,0) )</td>
</tr>
<tr>
<td>( L(R(T)) \leq N_{20} )</td>
<td>( L(R(T)) &gt; N_{20} )</td>
</tr>
<tr>
<td>( -phas(T,0) )</td>
<td>( -phas(T,0) )</td>
</tr>
</tbody>
</table>

\[ o_{52d}(T) = \pi^o_5 o_{52} = o_{52}(\pi^o_5(T)) \equiv \]

<table>
<thead>
<tr>
<th>L(T) &lt; 3</th>
<th>L(T) ≥ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \to tick(T) )</td>
<td>( \neg tick(T) )</td>
</tr>
<tr>
<td>( \to phas(T,0) )</td>
<td>( \neg phas(T,0) )</td>
</tr>
<tr>
<td>( L(R(T)) \leq N_{60} )</td>
<td>( L(R(T)) &gt; N_{60} )</td>
</tr>
<tr>
<td>( -phas(T,0) )</td>
<td>( -phas(T,0) )</td>
</tr>
</tbody>
</table>

\[ o_{53d}(T) = \pi^o_5 o_{53} = o_{53}(\pi^o_5(T)) \equiv \]

<table>
<thead>
<tr>
<th>L(T) &lt; 3</th>
<th>L(T) ≥ 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \to tick(T) )</td>
<td>( \neg tick(T) )</td>
</tr>
<tr>
<td>( \to phas(T,0) )</td>
<td>( \neg phas(T,0) )</td>
</tr>
<tr>
<td>( tp'(r(R(T))) )</td>
<td>( avg_3(p(T)) )</td>
</tr>
</tbody>
</table>

6.2.7 Is the TFM Description of the Behaviour of the Network Correct?

According to Lemma 5.1, we need to check whether the following formulas hold true:

- \( \pi^o_5 o_{51} \subseteq \pi^o_0 \text{avg} 1 \)
- \( \pi^o_5 o_{52} \subseteq \pi^o_0 \text{avg} 2 \)
- \( \pi^o_5 o_{53} \subseteq \pi^o_0 \text{current}_t \)
To show that \( \pi_k f_\theta \subseteq \pi_0 f_\omega \), every row in the network function table (\( \pi_k f_\theta \)) should imply a row or some rows in the requirement table (\( \pi_0 f_\omega \)) with the same output values. If so we can conclude that the network is an implementation of the requirement according to the conformance relation.

\[
\pi_0 \cdot \text{avg1} \Leftrightarrow \text{avg1}(\pi_0(T)) \equiv
\]

<table>
<thead>
<tr>
<th>( \pi_0(T) )</th>
<th>( \pi_0(T) \neq )</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick(( \pi_0(T) ))</td>
<td>phas(( \pi_0(T), 0 ))</td>
</tr>
<tr>
<td>( \pi_0(T) \neq )</td>
<td>( \leq N20 )</td>
</tr>
<tr>
<td>( \pi_0(T) \neq )</td>
<td>( &gt; N20 )</td>
</tr>
</tbody>
</table>

\( \equiv \) (the event frequency in the network is the same with that in the system)

<table>
<thead>
<tr>
<th>( T )</th>
<th>( T \neq )</th>
</tr>
</thead>
<tbody>
<tr>
<td>tick(T)</td>
<td>phas(T, 0)</td>
</tr>
<tr>
<td>( \pi_0(T) \neq )</td>
<td>( \leq N20 )</td>
</tr>
<tr>
<td>( \pi_0(T) \neq )</td>
<td>( &gt; N20 )</td>
</tr>
</tbody>
</table>

Compare this table with the (\( \pi_5 o_{51} \)) table in Section 6.2.6, we can see:

In implementation table \( \pi_5 o_{51} \):

**Row 1)**

\( L(T) < 3 \land o_{51} = 0.0 \) is subsumed by row 1) and row 5) in the requirement table \( \pi_0 \cdot \text{avg1} \)

**Row 2) and 5)** is subsumed by row 2) and 5) in the requirement table \( \pi_0 \cdot \text{avg1} \), respectively.

**Row 3)**

If the last event of \( T \) is in the third phase, the number of reads will be \( L(T)/3 \), i.e. \( L(T)/3 = L(R(T)) \). So row 3) in table \( \pi_5 o_{51} \) is subsumed by row 3) in the requirement table \( \pi_0 \cdot \text{avg1} \).

**Row 4)**
As defined in \( cp_3 \), read always happens in the first phase of a period. In phase 3 the most recent read is the third recent event, \( ri(3,T) = r(R(T)) \). So row 4) in table \( \pi_5 \circ o_{s1} \) is subsumed by row 4) in the requirement table \( \pi_6 \circ avg1 \).

The other two pairs of tables can be compared similarly (see Appendix 2). Through this checking it can be ensured that this design satisfies its requirements in the sense that if each of the components is implemented accordingly, we will get a correctly behaved system.

### 6.3 Flight Reservation System

The systems in the above two case studies operate continuously when started and the events happen regularly at some instants of a fixed frequency, which makes the simplification of the derived TFM descriptions relatively simple. In this example we consider an application whose events happen irregularly with time, i.e., at the instants that are invoked by users at any time. It illustrated how the composition operation and the conformance relation work for such a web-based application analyzed and designed via the object-oriented methodology.

#### 6.3.1. Overview of the System

A **Flight Reservation System (FRS)** is a software application to assist airlines to store and retrieve information and conduct transactions related to air travel, including flight scheduling, ticket reserving, cancelling and so on. We will not try to design such a complete and real system. Instead, we will make considerable simplifications just for the purpose of demonstrating how our conformance theory works for a web-based application. The system is for travel agents to reserve/cancel tickets for customers, and the flight information would be entered and managed by airline services.

A flight reservation system that can be used to book air tickets from many airlines has two basic functionalities:

1. **Manage flight schedule details**, including entering, retrieving, and editing flight information. This is operated by the airlines scheduling operators.

There are classes of flights with the same flight number, starting time and city, capacity, destination city and arrival time but on different days. We call such a class of flights with
the same flight number a “generic flight”. A particular flight is one of the generic flights on a particular day, which is called “specific flight” here.

At certain time points the operator reviews the flight schedule for some time period (6 months for instance) from now on, creates new generic flights of a certain flight number if there is none yet, adds new specific flights, modifies the flight information and so on.

2. Reserve/cancel seats in a particular flight for a customer (operated by customer service agents)

A sale agent is allowed to browse the seat availability of specific flights and the booking situations through interfaces provided by the flight information management.

Since a more complex system does not show anything more in illustrating how our method works, for simplification we make the following assumptions:

a). Omit the step “flight information lookup” and assume the desired flight number is already known, so that a customer can book tickets directly by giving flight number and departure date.

b). Only direct flights will be displayed, i.e., no interchanges are considered.

c). A customer can book seats for several passengers, and one or more seats booked by one customer in one specific flight are considered as one reservation.

d). It reserves or cancels seats without caring about seat position information.

e). Once a generic flight is created, the flight number will not change, and we use it as a key to identify a GenericFlight.

f). Once a specific flight is created, the flight number and departure date will not be changed, and they will be used as a joint key to identify the specific flight.

The specification of the system requirements is given in Appendix 3.

6.3.2. The Network of Components for the Task

The system can be divided into five components, each manipulating some objects of specific types. The secrets of the five information hiding components are:
Component “AH” (Agent Hiding) hides how the system interacts with the agent, and the secret is all aspects of communicating with the agent including language, sequence of information provision, etc.

The secret of “GFAM” (Generic Flight information Access Management) is the data structure used to store the generic flight information and the algorithms for storing, changing and retrieving that information.

The secret of “SFAM” (Specific Flight information Access Management) is the data structure used to store the specific flight information, the booked seats information for each flight, and the algorithms for checking available seats, adding and removing a booking.

The secret of “CIAM” (Customer Information Access Management) is the data structure used to store the customer information and the algorithms for storing, changing and retrieving that information.

Component “PGM splitter” hides how the different values of the PGM variable in the system are sent to different components. The secret is the algorithm for splitting the PGM variable.

All of the components operate only when invoked.

Here is the network description of the FRS:

\[ NT = (\Delta, \Sigma, \Omega, \mathcal{R}) \]

\[ \Delta = \{AH, GFAM, SFAM, CIAM, Splitter\} \]

\[ \Sigma = \{gf\text{flight}, sf\text{light}, customer, cid, fno, d, PGM\} \]

\[ \Omega = \{custid, flitno, ddate, bdate, bst\} \]

\[ \Gamma = \{\gamma_1, \gamma_2, \gamma_3, \gamma_4, \gamma_5, \gamma_6, \gamma_7, \gamma_8, \gamma_9, \gamma_{10}, \gamma_{11}, \gamma_{12}, \gamma_{13}, \gamma_{14}\} \]

\[ \mathcal{R} = \{res_3 \rightarrow \gamma_1 \rightarrow i_{14}, fno_1 \rightarrow \gamma_2 \rightarrow i_{22}, \]
\[ gft_2 \rightarrow \gamma_3 \rightarrow i_{32}, fno_1 \rightarrow \gamma_4 \rightarrow i_{33}, ddate_1 \rightarrow \gamma_5 \rightarrow i_{34}, snum_4 \rightarrow \gamma_6 \rightarrow i_{35}, \]
\[ cid_1 \rightarrow \gamma_7 \rightarrow i_{42}, cp_{51} \rightarrow \gamma_8 \rightarrow PGM_1, \]
\[ cp_{52} \rightarrow \gamma_9 \rightarrow PGM_{21}, cp_{51} \rightarrow \gamma_{10} \rightarrow PGM_{22}, \]
\[ cp_{53} \rightarrow \gamma_{11} \rightarrow PGM_{31}, cp_1 \rightarrow \gamma_{12} \rightarrow PGM_{32}, \]
\[ cp_{54} \rightarrow \gamma_{13} \rightarrow PGM_{41}, cp_{32} \rightarrow \gamma_{14} \rightarrow PGM_{42}, \]
$AH = (I, O, F)$

$I = \{i_{11}, i_{12}, i_{13}, i_{14}, PGM_1\}$

$O = \{cid_1, fno_1, ddate_1, bdate_1, bst_1, cp_1\}$

$D = \{i_{11} \rightarrow \text{cid}_1, i_{12} \rightarrow \text{fno}_1, i_{13} \rightarrow \text{ddate}_1, i_{14} \rightarrow \text{bst}_1\}$

$F = (\text{cid}_1(T), \text{fno}_1(T), \text{ddate}_1(T), \text{bdate}_1(T), \text{bst}_1(T), \text{cp}_1(T))$

$GFAM = (I_2, O_2, F_2)$

$I_2 = \{i_{21}, i_{22}, PGM_{21}, PGM_{22}\}$

$O_2 = \{gft_2\}$
\[ D_2 = \{ i_{21} \mapsto \text{gft}_2 \} \]
\[ F_2 = (\text{gft}_2(T)) \]
\[ SFAM = (I_3, O_3, F_3) \]
\[ I_3 = \{ i_{31}, i_{32}, i_{33}, i_{34}, i_{35}, \text{PGM}_{31}, \text{PGM}_{32} \} \]
\[ O_3 = \{ \text{res}_3, \text{cp}_31, \text{cp}_32 \} \]
\[ D_3 = \{ \} \]
\[ F_3 = (\text{res}_3(T), \text{cp}_31(T), \text{cp}_32(T)) \]
\[ CIT_4 = (I_4, O_4, F_4) \]
\[ I_4 = \{ i_{41}, i_{42}, \text{PGM}_{41}, \text{PGM}_{42} \} \]
\[ O_4 = \{ \text{snum}_4 \} \]
\[ D_4 = \{ i_{41} \mapsto \text{o}_41 \} \]
\[ F_4 = (\text{snum}_4(T)) \]
\[ CIT_5 = (I_5, O_5, F_5) \]
\[ I_5 = \{ \text{PGM}_5 \} \]
\[ O_5 = \{ \text{cp}_{51}, \text{cp}_{52}, \text{cp}_{53}, \text{cp}_{54} \} \]
\[ D_5 = \{ \} \]
\[ F_5 = (\text{cp}_{51}(T), \text{cp}_{52}(T), \text{cp}_{53}(T), \text{cp}_{54}(T)) \]

### 6.3.3 Complete and Simplified Network Descriptions

In the first section of Appendix 3, we define some data types as modules that have operations revealing or changing the state of each object of this type. Since a component is a collection of programs that has clear interfaces for communication with its environment, each variable or object of such a data type is a component. In this sense, the variables such as “gflight”, “sflight” in Fig 6.6 are all components and there should be more blocks in the network diagrams.

To simplify a network, we distinguish the component that takes care of all variables of a certain type from the components that use variables of such a type. If there is a data type and a component for creating and changing the values of variables of this type, we can have other components that have internal storage which contains variables of that type.

The first type of components includes program to store information, alter the information stored in an object, and compute the value of the output variables of that object, which is
called VCAD (Variable Create, Alter, Destroy) components\(^3\). They do not compute information using or affecting more than one object in any operation. The second type of components includes the storage used for those variables and includes programs that stores information of many objects in the component and retrieves information of certain object from this component.

Since we view a system as communicating a set of components, we need to show the components and (1) the connection of an output variable to a single interconnector and (2) the connection of the interconnectors to component inputs. A network illustrates all components and all connections. All outputs are passed to VCAD components which are then connected to a component that receives the information as input. All communications between components other than the VCAD components are via one of the VCAD components.

To simplify a network description we neglect all VCAD components and their interconnections. Instead of showing the VCAD component, we indicate only the type of the variable. A VCAD component is associated with a certain type which is implicit in the network description since the variable will be stored in that VCAD component. Because the VCAD component does not store any hidden information, it is not necessary to use it in computing the input output function of the network. A network description without VCAD components is called a simplified network description. As a contrast, the above described network description including all VCAD components is called complete network description.

Figure 6.6 is a simplified network diagram in the sense that all variables that are connected to the components be of the types we have defined, while the VCAD components are not shown in the diagram.

**6.3.4 Is the Network Complete and Consistent?**

The TFM descriptions of the constructing components in the network are provided in Appendix 3. From Def. 3.2 and Def. 3.11 it can be checked that the network is completely connected and it is consistent, and there are no delay-free loops in the

---
\(^3\) The paragraphs about VCAD are from David Parnas by personal communication.
network. We can derive a TFM description of the behaviour of the network from the TFM descriptions of the constructing components.

6.3.5 Is the TFM Description of the Behavior of the Network Correct?

The detailed derivation procedure of the TFM descriptions of the network is given in Appendix 3. Here we compare the final results only. By Lemma 5.1, to show the conformance of the behaviour of the description of a network and the system requirements, we need to check whether the following relationships hold:

- $\pi^0_1 \circ cid_1 \subseteq \pi^0_0 \circ custid$
- $\pi^0_1 \circ fno_1 \subseteq \pi^0_0 \circ flitno$
- $\pi^0_1 \circ ddate_1 \subseteq \pi^0_0 \circ ddate$
- $\pi^0_1 \circ bdate_1 \subseteq \pi^0_0 \circ bdate$
- $\pi^0_1 \circ bst_1 \subseteq \pi^0_0 \circ bst$

1) $\pi^0_1 \circ cid_1 \subseteq \pi^0_0 \circ custid$

$$
\pi_1(T) = _- \Leftrightarrow T = _- \vee \neg \text{ex}(PGM()=\text{makResv} \vee PGM()=\text{canResv})(T),\text{ therefore,} \\
\pi_1(T) = _- \wedge \text{cid}_1 = \phi_n \text{ is subsumed by row 1) and row 3) in the requirement table } \pi^0_0 \circ custid

Row 1)

$\pi_1(T) = _- \Leftrightarrow T = _- \vee \neg \text{ex}(PGM()=\text{makResv} \vee PGM()=\text{canResv})(T)$, therefore,

$\pi_1(T) = _- \wedge \text{cid}_1 = \phi_n$ is subsumed by row 1) and row 3) in the requirement table $\pi^0_0 \circ custid$

Row 2) and 3) are subsumed by row 2) and 3) in the requirement table $\pi^0_0 \circ custid$, respectively.

From the above we conclude that $\pi^0_1 \circ cid_1 \subseteq \pi^0_0 \circ custid$ holds true.

The $\pi^0_1 \circ fno_1 \subseteq \pi^0_0 \circ flitno$, $\pi^0_1 \circ ddate_1 \subseteq \pi^0_0 \circ ddate$ and $\pi^0_1 \circ bdate_1 \subseteq \pi^0_0 \circ bdate$ can be proved similarly.
2) $\pi_i^o bst_1 \subseteq \pi_0^o bst$

$\pi_i^o bst_1 \equiv bst_{1d}(T) \equiv bst_1(\pi_i(T)) \equiv$

- $\pi_1(T) = _-$  
  - $\pi_1(T) \neq _-$ \quad $\neg (\neg (PGM(r(T))=makResv \lor PGM(r(T))=canResv))$  
  - $\neg (PGM(r(T))=makResv \lor PGM(r(T))=canResv)$

$\pi_0^o bst \equiv bst(\pi_0(T)) \equiv$

- $T = _-$
  - $T \neq _-$ \quad $PGM(r(T))=makResv \land e = \phi_0$  
    - $\neg (e = \phi_0) \land sflight.fst(e) \geq n(cid'(r(T)),T)$  
    - $sflight.fst(e) < n(cid'(r(T)),T)$
  - $PGM(r(T))=canResv$

Where $e' \equiv s(i_{33}(r(\pi_3(T))), i_{34}(r(\pi_3(T))), \pi_3(T)) \equiv$

- $\pi_1(T) = _-$
  - $\pi_1(T) \neq _-$ \quad $PGM(r(T))=addSflight \land fno'(r(T))=sflight.fno'(r(T)) \land d'(r(T))=sflight.depdate'(r(T))$  
    - $\neg (PGM(r(T))=addSflight \land fno'(r(T))=sflight.fno'(r(T)) \land d'(r(T))=sflight.depdate'(r(T)) \land s(fno'(r(T)), d'(r(T)), p(\pi_3(T))))$

In the network function table $\pi_i^o bst_1$:

**Row 1:** $\pi_i(T) = _- \iff T = _- \lor \neg \exists (PGM()=makResv \lor PGM()=canResv)(T)$

“$\pi_i(T) = _- \land bst_1 = empty$” is subsumed by row 1) and 6) in the requirement table $\pi_0^o bst$.

**Row 2:** From the definition of $e'$ and $e$ it can be seen $e' = \phi_0 \Rightarrow e = \phi_0$, because:

1) $\pi_0(T) = _- \Rightarrow T = _- \lor \neg \exists (PGM()=addSflight)(T)$
2) $\pi_0(T) \neq _- \Rightarrow T \neq _-$
3) $\pi_i(T) \neq _- \land \pi_3(T) \neq _- \Rightarrow T = _-$
From 1), 2) and 3), row 2) "\(\pi_1(T) \neq \_ \land \pi_3(T) \neq \_ \land PGM(r(T)) = makResv \land e' = \phi_0 \land bst_1 = empty\)" in the network function table is subsumed by row 2) and 6) in the requirement table. (if a \(\Rightarrow\) c then a\(\land\)b \(\Rightarrow\) c\(\land\)b).

**Row 3):** \(\neg(e'=\phi_0) \land sflight.fst(e') \geq snum_4'(r(T)) \Rightarrow \neg(e=\phi_0) \land sflight.fst(e) \geq n(cid'(r(T)),T)\) because:

\[
sflight.fst(r(\pi_3(T))) = sflight.fst(r(T)) \quad \text{(sflight changed only in SFAM)}
\]
\[
\pi_3(T) \neq \_ \land (PGM(r(T)) = addSflight \land fno'(r(T)) = sflight.fno'(r(T)) \land \\
d'(r(T)) = sflight.depdate'(r(T)) \land sflight.fst(r(\pi_3(T))) \geq n(cid'(r(T)),T) \Rightarrow \\
T \neq \_ \land (PGM(r(T)) = addSflight \land fno'(r(T)) = sflight.fno'(r(T)) \land \\
d'(r(T)) = sflight.depdate'(r(T)) \land sflight.fst(r(T)) \geq n(cid'(r(T)),T)
\]
Therefore row 3) in the network table is subsumed by row 3) in the requirement table.

**Row 4):** similar to the above, \(\neg(e'=\phi_0) \land sflight.fst(e') < snum_4'(r(T)) \Rightarrow \neg(e=\phi_0) \land sflight.fst(e) < n(cid'(r(T)),T)\), so that row 4) in the network table is subsumed by row 4) in the requirement table.

**Row 5):** \(\pi_1(T) \neq \_ \land \pi_3(T) \neq \_ \land PGM(r(T)) = canResv \land bst_1 = cancelled\) is subsumed by row 5) in the requirement table \(\pi_0^\circ bst\) because \(\pi_1(T) \neq \_ \land \pi_3(T) \neq \_ \Rightarrow T \neq \_
\)

(if a \(\Rightarrow\) c then a\(\land\)b \(\Rightarrow\) c\(\land\)b)

**Row 6)** \(\pi_1(T) \neq \_ \land \neg(PGM(r(T)) = makResv \lor PGM(r(T)) = canResv) \land bst_1 = bst_{id}(p(T))\) is subsumed by row 6) in the requirement table \(\pi_0^\circ bst\) because \(\pi_1(T) \neq \_ \Rightarrow T \neq \_
\)

From the above we can conclude that \(\pi_1^\circ bst_1 \subseteq \pi_0^\circ bst\).
Chapter 7
Conclusions and Future Work

The development of software systems based on components is error-prone. Benefiting from the rigorous mathematical design documents, the completeness and consistency of the design documents can be verified mathematically in the early development phase. If a set of components are supposed to work together to finish a task, it is important that a description of the component network can be used to describe the communication paths among the components, since it is the necessary information for constructing the components back into a system and for checking the consistency and correctness of the precise documentation in the requirement and design phase of the development.

7.1 Summary and Conclusion

In this thesis we have defined a network of components and discussed the rules for checking when the network is completely connected and consistent. These rules are:

- Each network output variable will be connected to exactly one component output variable.
- Network input variables can be connected to any number of component inputs.
- If \((o \rightarrow \gamma \rightarrow i) \in R\), the data types of the three variables \(o\), \(\gamma\) and \(i\) must be consistent: the set of possible values of \(o\) must be a subset of the set of possible values of \(\gamma\), and the set of possible values of \(\gamma\) must be a subset of the set of possible values of \(i\).
- If there is a loop in the network, there should be at least a delay in the loop.

If a network is complete and consistent as described above, the connected components could work together. We have provided a formula for computing the TFM description of a complete and consistent network, and then illustrated how the behaviour can be derived from the network description and the TFM descriptions of the components. Some techniques on how to simplify the derived trace function expressions are also developed.
If the network behaviour computed in this way satisfies the requirements for the network, one can proceed with the implementation of the internal components in confidence that if the components are implemented correctly, the network will function satisfactorily. For this purpose we have established the conformance relation between a design and its requirements.

If the behavior of a set of components is described by the trace function method in [92], we can check the compatibility of the set of TFM descriptions to see whether the components can work together properly. If the requirements of the software system are also given by the TFM, we can then check the conformance between the behavior of the network of components and the system requirements. As a result of this checking procedure, one can get a set of reliable TFM descriptions of components, so that if each component is implemented according to its specification, the system constructed by these components will behave correctly. This process gives developers the assurance that a design can correctly implement the behavior specified in the requirements.

The building up of a mathematics-based requirements and design documents, the definition of a network of components, and the logical checking of these design documents are a rational way to design trustworthy software. This thesis illustrates how to use our approach to document and to check the documents of a software system before implementation.

The proposed approach has been illustrated by using three case studies. The first one has only input/output connection lines in each device and the other two are general software systems with access programs in each component. This approach is suitable for software systems with many components working simultaneously together. It assures the compatibility of the interconnected components and the conformance between an implementation and its requirements in a hierarchical way. Theoretically the approach can decrease the likelihood of critical errors at the earlier phase of the software development process and increase the trustworthiness of the developed system.

7.2 Contributions

In a complex system, although the role of each individual instruction and the behaviour of each individual component are clearly defined, their interaction through shared data
and devices may make it difficult to understand the behaviour of the full system. Therefore, to check whether the integrated complex system works properly is a difficult task. This thesis makes some initial attempts towards this goal.

The research work in this thesis resulted in the following contributions to the studies of inter-component consistencies and the correctness of mathematical design documents:

(1) A network description has been proposed, which specifies the inter-component dependencies based on variables; and delay-free dependencies among variables within a component are defined to describe the inner-component dependency. This enables the completeness and consistency checking among the interconnected components and the derivation of a network behavior of a set of components. Although the ideas of describing interconnection among components are not new, as stated in Chapter 1 and reviewed in Chapter 3, we have shown how the simple “wire-connection” that is designed traditionally for hardware and for processes [75] works for software components.

(2) The completeness and consistency checking of a network is developed, which ensures that the set of components can work together harmoniously to lead to a well defined behavior. The completeness of a network describes how the connections among components are restricted, and the network consistency describes how the data types are matched between the connected variables and in which cases there exist well defined behaviors for the network. With the network description and the variable dependency description, delay-free loops can be detected, and the network with such a kind of loop is excluded from our consideration of correctness checking.

(3) The formulas for determining a TFM description of a network from component interface TFM descriptions and the component interconnections are developed, which is the key step towards the conformance checking between a network behavior and the system requirements. Based on the properties of the formulas for a network of two components, the derivation formulas have been theoretically extended to networks consisting of any number of components.

(4) The conformance relations using the relational approach has been built up, which suit the occasion of comparing the white-box behavior of a network of components and the
black-box behavior of the system requirements. Some projection functions are defined to enable the comparison of the TFM descriptions at design level and the specifications at the requirements level, since the domain of the relations at different levels of development are different. The commuting diagrams for derivation operation and correctness checking are developed.

(5) Three case studies have been presented, which provide some intuitive backgrounds of using the approach developed in this thesis and have demonstrated the feasibility of the approach. We tried to choose the case studies in the real world with considerable simplifications, and in each application we focus on some special issues. We presented the three applications from system requirement analysis, functional decomposition, component interface description, network description, and then applied the derivation formulas and the conformance checking relation to these documents.

7.3 Limitations of the work
Although the feasibility of the derivation operation and the conformance checking are illustrated by several case studies, they were all done manually. Without the support of a tool, this is possible only for relatively small systems.

If the number of components in the network is very large, the resultant tabular expression after the internal variable substitution might be complicated and the comparison might not be so straightforward. However, whether or not the method scales up depends largely on the quality of the design. In a good design, each component abstracts from the details of the processing within it. In such cases, the functional descriptions of the component will be simple - no matter how many subcomponents there are. A poorly designed system with a small number of components will be harder to analyse than a well designed system with many more components.

7.4 Future Work
To show the real applicability of the approach, investigations towards the automation of the usage of the approaches are certainly necessary. Since the procedure is logic- and mathematics-oriented, it is possible to develop a software tool to 1) extend the component trace functions to network trace functions and replace the intermediate
variable by its function definition, and 2) check the correctness of the mathematical
documents for a design.

Employing an appropriate theorem prover and converting the descriptions and
specifications to the corresponding format can be investigated. It is possible that the
conformance checking between a description of the behaviour of a network and the
requirements can be realized by a theorem prover.

As mentioned in Chapter 5, in the conformance checking, one trace relation in the
requirements may need to be compared with a set of trace relations from different
components that are supposed to implement the required relation. The comparison is
between a composed complex relation and a simple relation. There may be problems due
to the fact that either it is not easy to come up with a composed tabular expression, or it
may be complicated to check whether the two tabular expressions are the same or
whether there are any implications between them even if we can get one tabular
expression.

Another point where the work can be extended is that the domain coverage for the
conformance checking is not considered in the case studies. More work is needed to
investigate the simplification of the deriving procedure of getting the TFM description of
the behaviour of a network and checking the domain coverage of the conformance
relation. The simplification of converting the trace relations into the same domain is also
interesting and may need more formalization. In the case studies we used some
application-related properties to simplify the function tables, which is a weakness for
using an automation tool to realize the procedure.

For conformance checking between a set of design documents and its requirements, we
have used the basic relation of conformance between an implementation and its
specification. Since the implementation mentioned in this thesis is a general concept,
which could be a design, a detailed design or coding, the relation can be extended to
checking the conformance between an implementation and its specification in other
levels of refinement, as long as the specification/implementation are specified/described
by defining the function between the output values and the history of the input/output
values.
Appendix 1

The Reset/Set Flip-flop

A1.1 The Enumerate Derivation

1. Derive $CIT_{12}$ from $CIT_{NOT}$ and $CIT_{AND}$ by Def. 4.1 (see section 6.1)

2. Derive $CIT_{12,3}$ from $CIT_{12}$ and $CIT_{OR}$ by Def. 4.1

Table A1:

<table>
<thead>
<tr>
<th>$U_{12,3}$ ($&lt;T_{12}&gt; \times &lt;T_{3}&gt; \times &lt;T&gt;$)</th>
<th>$T[2]$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$011$</td>
<td>$022$</td>
</tr>
<tr>
<td>1200</td>
<td>1</td>
</tr>
<tr>
<td>1210</td>
<td>1</td>
</tr>
<tr>
<td>1222</td>
<td>1</td>
</tr>
<tr>
<td>1232</td>
<td>1</td>
</tr>
<tr>
<td>2100</td>
<td>1</td>
</tr>
<tr>
<td>2111</td>
<td>1</td>
</tr>
<tr>
<td>2120</td>
<td>1</td>
</tr>
<tr>
<td>2131</td>
<td>1</td>
</tr>
<tr>
<td>0311</td>
<td>1</td>
</tr>
<tr>
<td>0322</td>
<td>1</td>
</tr>
<tr>
<td>3010</td>
<td>1</td>
</tr>
<tr>
<td>3020</td>
<td>1</td>
</tr>
<tr>
<td>0300</td>
<td>1</td>
</tr>
<tr>
<td>0333</td>
<td>1</td>
</tr>
<tr>
<td>3000</td>
<td>1</td>
</tr>
<tr>
<td>3030</td>
<td>1</td>
</tr>
</tbody>
</table>

The blocks on the top right and bottom left (the pale grey parts) of grid $T[0]$ correspond to the points when one of the components has state change while the other has none. Because of the looped connections, there can be network events in such cases. If this is the final step of calculation, there is no need to check the right bottom block (dark grey part). But as an intermediate step this state values will be used in the next step of calculation.


The subscript $ijkn$ in grid $T[1]$ is used to indicate the value change of $i_1$, $\gamma_3$, $i_2$, and $\nu_2$.

The subscript $opq$ in grid $T[2]$ is used to indicate the change of $i_3$, $i_2$, and $\nu_3$.

From the network rules and the consistency conditions we should have $n=p$. ($i_{32} \leftarrow \gamma_2 \leftarrow \nu_2 \Rightarrow 'i_{32} = ' \nu_2 ', i_{32}' = \nu_2 ' \Rightarrow n=p)$

All the consistent traces should satisfy the condition $n=p$ for the corresponding event descriptors.

Table A1 reads as:
∀t, \( E(T_{12})(t) = e_{ijkn} \land E(T_3)(t) = 3e_{opq} \land n=p \)

\( \Rightarrow U_{12,3}(T_{12}, T_3) = T_{12,3} \) s.t. \( E(T_{12,3})(t) = e_{ijknopq} \)

It can be converted to:

\( r(T_{12}) = e_{ijkn} \land r(T_3) = 3e_{opq} \land n=p \Rightarrow U_{12,3}(T_{12}, T_2) = T_{12,3} \) s.t. \( r(T_{12,3}) \in e_{ijknoq} \)

Possible descriptors in \( CIT_{123} \) (network consists of NOT, AND, OR gates):

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>( i_1 )</th>
<th>( \gamma_1 )</th>
<th>( i_2 )</th>
<th>( \gamma_2 )</th>
<th>( i_3 )</th>
<th>( v_3 ) for e e</th>
</tr>
</thead>
<tbody>
<tr>
<td>e_120011</td>
<td>0 1 1 0 0 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_120022</td>
<td>0 1 1 0 0 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_120000</td>
<td>0 1 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_120033</td>
<td>0 1 1 0 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_121011</td>
<td>0 1 1 0 1 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_121022</td>
<td>0 1 1 0 1 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_121000</td>
<td>0 1 1 0 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_121033</td>
<td>0 1 1 0 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_122202</td>
<td>0 1 1 0 1 1 1 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_122213</td>
<td>0 1 1 0 1 1 1 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_122222</td>
<td>0 1 1 0 1 1 1 1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_122233</td>
<td>0 1 1 0 1 1 1 1 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_123202</td>
<td>0 1 1 1 0 0 1 1 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_123213</td>
<td>0 1 1 1 0 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_123222</td>
<td>0 1 1 1 1 0 1 1 0 1 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_123233</td>
<td>0 1 1 1 1 1 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_210011</td>
<td>1 0 1 0 0 0 0 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_210022</td>
<td>1 0 1 0 0 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_210000</td>
<td>1 0 1 0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_210033</td>
<td>1 0 1 0 0 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_211101</td>
<td>1 0 1 0 1 0 1 0 0 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_211111</td>
<td>1 0 1 0 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_211123</td>
<td>1 0 1 0 1 0 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_211133</td>
<td>1 0 1 0 1 0 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_212011</td>
<td>1 0 1 1 0 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_212022</td>
<td>1 0 1 1 0 0 0 1 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_212000</td>
<td>1 0 1 1 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_212033</td>
<td>1 0 1 1 0 0 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_213101</td>
<td>1 0 1 1 0 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_213111</td>
<td>1 0 1 1 0 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_213123</td>
<td>1 0 1 1 0 1 1 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_213133</td>
<td>1 0 1 1 0 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_031101</td>
<td>0 0 1 1 0 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_031111</td>
<td>0 0 1 1 0 1 0 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_031123</td>
<td>0 0 1 1 0 1 0 1 1 0 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_031133</td>
<td>0 0 1 1 0 1 1 1 1 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_032202</td>
<td>0 0 1 1 1 0 1 0 0 0 1 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_032213</td>
<td>0 0 1 1 1 0 1 0 0 1 1 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>e_032222</td>
<td>0 0 1 1 1 0 1 0 1 1 1 0 1 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

113
3. Derive $CIT_{123,4}$ from $CIT_{123,3}$ and $CIT_D$ by Def. 4.1

Table A2
$U_{123,4} \left( <T_{123}> \times <T_4> \times <T> \right) \equiv T[2]$

$$
\begin{array}{cccccccc}
& 4e_{10} & 4e_{02} & 4e_{12} & 4e_{23} & 4e_{31} & 4e_{21} & 4e_{30} & 4e_{33} \\
\text{c_{120001}} & 1 & & & & & & & \\
\text{c_{120022}} & & 1 & & & & & & \\
\text{c_{120000}} & & & & & & & & \\
\text{c_{20033}} & & & & & & & & \\
\text{c_{121011}} & & & & & & & & \\
\text{c_{121022}} & & & & & & & & \\
\text{c_{121000}} & & & & & & & & \\
\text{c_{121033}} & & & & & & & & \\
\text{c_{122202}} & & & & & & & & \\
\text{c_{122213}} & & & & & & & & \\
\text{c_{122222}} & & & & & & & & \\
\text{c_{122233}} & & & & & & & & \\
\text{c_{123202}} & & & & & & & & \\
\text{c_{123213}} & & & & & & & & \\
\text{c_{123222}} & & & & & & & & \\
\end{array}
$$
Appendix 1 The Reset/Set Flip-flop
In Table A2, the subscript $ijknoq$ is used to indicate the value change of $i_{11}$, $\gamma_1$, $i_{21}$, $\gamma_2$, $i_{31}$ and $v_3$. The subscript $rs$ is used to indicate the value change of $i_{41}$ and $v_4$.

From the network rules and the consistency conditions we should have $q=r$ and $k=s$, because:

\[
i_{41} \leftarrow \gamma_3 \leftarrow v_3 \Rightarrow 'i_{41} = 'v_3, i_{41}' = v_3' \Rightarrow q=r
\]
\[
i_{21} \leftarrow \gamma_4 \leftarrow v_4 \Rightarrow 'i_{21} = 'v_4, i_{21}' = v_4' \Rightarrow k=s
\]

The subscript of all the consistent event descriptors should satisfy the condition $q=r$ and $k=s$.

The table reads as:

\[
\forall t, \ E(T_{12_3})(t) = e_{ijknoq} \land E(T_4)(t) = 4ers \land q=r \land k=s
\]

\[
\Rightarrow U_{123_4}(T_{12_3}, T_4) = t \text{ s.t. } E(T)(t) = eeijknopq
\]

It can be converted to:

\[
r(T_{12_3}) = e_{ijknoq} \land r(T_4) \in 4er \land q=r \land k=s \Rightarrow U_{123_4}(T_{12_3}, T_4) = t \text{ s.t. } r(T) = e_{ijknopq}
\]

Possible event descriptors for the complete network consisting of NOT, AND, OR gates and delay device:

<table>
<thead>
<tr>
<th>Name of e set</th>
<th>$i_{11}(R)$</th>
<th>$\gamma_1$</th>
<th>$i_{21}$</th>
<th>$\gamma_2$</th>
<th>$i_{31}(S)$</th>
<th>$\gamma_3$</th>
<th>$v_4(Q)$ for ce</th>
</tr>
</thead>
</table>
A1.2 The Conformance Checking
See Section 6.1.6.

A1.3 The Symbolic Derivation
The TFM interface descriptions of the four components in RS flip-flop circuit are shown in Section 6.1.
Here we use the formula $\pi_k \circ f_k = f_k(\pi_k(T))$ to get the functions of network traces:

$v_{1d}(T) = \pi_1 \circ v_1 = v_1(\pi_1(T)) =
\begin{array}{c|c}
\pi_1(T) = & 1 \\
\pi_1(T) \neq & -i_{11}'(r(\pi_1(T))) \\
\end{array}
\equiv (Lemma 4.4)
\begin{array}{c|c}
\pi_1(T) = & 0 \\
\pi_1(T) \neq & -i_{11}'(r(T)) \\
\end{array}

$v_{2d}(T) = \pi_2 \circ v_2 = v_2(\pi_2(T)) =
\begin{array}{c|c|c|c|c}
\pi_2(T) = & 0 \\
\pi_2(T) \neq & 0 & i_{21}'(r(T))=1 \\
& 1 & i_{21}'(r(\pi_2(T)))=1 \\
& 0 & i_{21}'(r(\pi_2(T)))=0 \\
& 1 & i_{22}'(r(\pi_2(T)))=0 \\
& 0 & i_{22}'(r(\pi_2(T)))=1 \\
\end{array}
\equiv (Lemma 4.4)
\begin{array}{c|c|c|c|c}
\pi_2(T) = & 0 \\
\pi_2(T) \neq & 0 & i_{21}'(r(T))=1 \\
& 1 & i_{21}'(r(T))=0 \\
& 1 & i_{22}'(r(T))=1 \\
& 0 & i_{22}'(r(T))=0 \\
\end{array}

$v_{3d}(T) = \pi_3 \circ v_3 = v_3(\pi_3(T)) =
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\pi_3(T) = & 0 \\
\pi_3(T) \neq & 0 & i_{31}'(r(T))=1 \\
& 1 & i_{31}'(r(\pi_3(T)))=0 \\
& 0 & i_{32}'(r(T))=1 \\
& 0 & i_{32}'(r(\pi_3(T)))=0 \\
\end{array}
\equiv (Lemma 4.4)
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\pi_3(T) = & 0 \\
\pi_3(T) \neq & 0 & i_{31}'(r(T))=1 \\
& 0 & i_{31}'(r(T))=0 \\
& 0 & i_{32}'(r(T))=1 \\
& 1 & i_{32}'(r(T))=0 \\
\end{array}
\begin{align*}
\pi_4(T) \neq \_ & \land
\begin{array}{l}
i_{31}'(r(T)) = 0 \land i_{32}'(r(T)) = 0 \\
i_{31}'(r(T)) = 0 \land i_{32}'(r(T)) = 1 \\
i_{31}'(r(T)) = 1 \land i_{32}'(r(T)) = 0 \\
i_{31}'(r(T)) = 1 \land i_{32}'(r(T)) = 1
\end{array}
\begin{array}{l}
0 \\
1 \\
1 \\
1
\end{array}
\end{align*}

\[v_{4d}(T) = \pi_4 \circ v_4 = v_4(\pi_d(T)) \equiv
\begin{array}{|c|c|}
\hline
\pi_d(T) = \_ & i_{41}'(r(\pi_d(T))) = 0 \\
\pi_d(T) \neq \_ & i_{41}'(r(\pi_d(T))) \\
\hline
\end{array}
\equiv (\text{Lemma 4.4})

\begin{align*}
\pi_4(T) = \_ & \land
\begin{array}{l}
t(r(\pi_d(T))) = t(r(T)) \\
t_0 = t(r(\pi_d(T))) < t(r(T)) \\
\end{array}
\begin{array}{l}
i_{41}(r(T)) \\
i_{41}(E(T)(t_0))
\end{array}
\begin{array}{l}
0 \\
1
\end{array}
\end{align*}

Since \(t_0\) is the time of the last event in the delay device, after \(t_0\) there is no component state changes, but there might be \(n \geq 1\) network events including the last one at \(t_1 = t(r(T))\), see Fig. 4.10. For the delay device we can have the following:

\[v_4'(r(T)) = 'v_4(r(T))\] (no event at \(t_1\) in the delay device so the values of \(v_4\) is unchanged)

\[i_{41}'(r(\pi_4(T))) = 'i_{41}(r(\pi_4(T))) \Rightarrow i_{41}'((E(T)(t_0))) = 'i_{41}((E(T)(t_0)))\] (otherwise there must be a component event after \(t_0^*\))

\[v_4'(r(\pi_d(T))) = 'i_{41}(r(\pi_d(T)))\] (from the specification) \Rightarrow

\[v_4'(E(T)(t_0)) = 'i_{41}((E(T)(t_0)))\]

\[v_4'(E(T)(t_0)) = 'v_4(r(T))\] (From \(t_0\) to \(t_1\) no event in the delay device)

\[i_{41}'(E(T)(t_0)) = 'i_{41}(r(T))\] (as above)

\[\Rightarrow v_4'(r(T)) = 'i_{41}(E(T)(t_0)) = 'i_{41}(r(T))\] (1)

If \(i_{41}'((E(T)(t_0)) \neq 'i_{41}((E(T)(t_0)))\), according to the property of the one step delay device in a synchronous system, after one clock tick the value of \(v_4\) will be changed so that there is a component state change at \(t_0 + t_c\), \(t_c\) is the time of a clock tick. Since there is no other component state changes between \(t_0\) and \(t_1\), and \(t_1 - t_0 = n\times t_c\ (n \geq 1)\), we can conclude that there is no input change in the last component event.

After the simplification we can start to use the consistency rules to replace the internal variables.

\[v_{4d}(T) = \pi_4 \circ v_4 = (\text{equation 1})\]

\[\begin{array}{|c|c|}
\hline
\pi_4(T) = \_ & i_{41}'(r(T)) = 0 \\
\pi_4(T) \neq \_ & i_{41}'(r(T)) \\
\hline
\end{array}
\equiv (i_{41} \leftarrow \gamma_3', v_3 \in \mathcal{R} \Rightarrow 'i_{41}(r(T)) = 'v_3(r(T)) = v_3d(p(T))\ by \text{Lemma 2})

\begin{align*}
\pi_4(T) = \_ & \land
\begin{array}{l}
p(T) = \_ \\
p(\pi_d(T)) = \_ \\
p(T) \neq \_ \\
p(\pi_d(T)) \neq \_
\end{array}
\begin{array}{l}
0 \\
0 \\
0
\end{array}
\end{align*}

\equiv (\text{Lemma 4.4 again and continuity condition } i_{31}'(r(p(T))) = 'i_{31}(r(T)))

\begin{align*}
\pi_4(T) = \_ & \land
\begin{array}{l}
p(T) = \_ \\
p(\pi_d(T)) = \_ \\
p(T) \neq \_ \\
p(\pi_d(T)) \neq \_
\end{array}
\begin{array}{l}
0 \\
0 \\
0
\end{array}
\end{align*}
Appendix 1 The Reset/Set Flip-flop

\begin{align*}
\pi_4(T) = \_ & \quad 0 \\
\pi_4(T) \neq \_ & \quad 0 \\
\pi_3(T) = \_ & \quad 0 \\
\pi_3(T) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_4(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_4(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_3(r(T)) = \_ & \quad 0 \\
\pi_3(r(T)) \neq \_ & \quad 0 \\
\pi_3(r(T)) = \_ & \quad 0 \\
\pi_2(r(T)) = \_ & \quad 0 \\
\pi_2(r(T)) \neq \_ & \quad 0 \\
\pi_2(r(T)) = \_ & \quad 0 \\
\pi_2(r(T)) \neq \_ & \quad 0 \\
\pi_1(r(T)) = \_ & \quad 0 \\
\pi_1(r(T)) \neq \_ & \quad 0 \\
\end{align*}

\(\equiv (\pi_4(p(T)) = \_ \text{ is added since it's implied by } p(T) = \_, \text{ then row 2 and 3 can be merged. } p(T) \neq \_ \text{ is implied by } \pi_3(p(T)) = \_ \text{ and can be removed from row 3 below})\)

\begin{align*}
\pi_4(T) = \_ & \quad 0 \\
\pi_4(T) \neq \_ & \quad 0 \\
\pi_3(T) = \_ & \quad 0 \\
\pi_3(T) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(r(T)) = \_ & \quad 0 \\
\pi_1(r(T)) \neq \_ & \quad 0 \\
\pi_1(r(T)) = \_ & \quad 0 \\
\pi_1(r(T)) \neq \_ & \quad 0 \\
\pi_0(r(T)) = \_ & \quad 0 \\
\pi_0(r(T)) \neq \_ & \quad 0 \\
\pi_0(r(T)) = \_ & \quad 0 \\
\pi_0(r(T)) \neq \_ & \quad 0 \\
\end{align*}

\(\equiv (i_{32} \leftarrow \gamma_2 \leftarrow v_2 \in \mathcal{R} \Rightarrow 'i_{32}(r(T)) = 'v_2(r(T)) \Rightarrow 'i_{32}(r(T)) = v_2(p(T)) \text{ by Lemma 4.5})\)

\(i_{21} \leftarrow \pi_2(p(T)) = \_\ (\pi(p(T)) = 'i_{21}(r(T)) \ ) \text{ by projection definition and CC}\)

\begin{align*}
\pi_4(T) = \_ & \quad 0 \\
\pi_4(T) \neq \_ & \quad 0 \\
\pi_3(T) = \_ & \quad 0 \\
\pi_3(T) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(p(T)) \neq \_ & \quad 0 \\
\end{align*}

\(\equiv (\text{re-arrange the rows and columns})\)

\begin{align*}
\pi_4(T) = \_ & \quad 0 \\
\pi_4(T) \neq \_ & \quad 0 \\
\pi_3(T) = \_ & \quad 0 \\
\pi_3(T) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_3(p(T)) = \_ & \quad 0 \\
\pi_3(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_2(p(T)) = \_ & \quad 0 \\
\pi_2(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(p(T)) \neq \_ & \quad 0 \\
\pi_1(p(T)) = \_ & \quad 0 \\
\pi_1(p(T)) \neq \_ & \quad 0 \\
\end{align*}

\(\equiv i_{21} \leftarrow \gamma_4 \leftarrow v_4 \in \mathcal{R} \Rightarrow 'i_{21}(r(T)) = 'v_4(r(T)), i_{22} \leftarrow \gamma_1 \leftarrow v_1 \in \mathcal{R} \Rightarrow 'i_{22}(r(T)) = 'v_1(r(T)) = v_1(p(T))\)
\[\begin{array}{|c|c|c|}
\hline
\pi_3(p(T)) & \pi_5(p(T)) & i_{31}(r(T)) = 0 \\
\neq \land & \neq \land & 1 \\
\hline
\pi_2(p(T)) & \pi_5(p(T)) & i_{31}(r(T)) = 1 \\
\neq \land & \neq \land & 0 \\
\hline
\end{array}\]

\[\equiv (\text{rearrange the rows and columns})\]

\[\begin{array}{|c|c|c|}
\hline
\pi_3(p(T)) = \_ & \pi_5(p(T)) = \_ & 0 \\
\land & \land & 1 \\
\hline
\pi_2(p(T)) & \pi_5(p(T)) = \_ & 1 \\
\neq \land & \neq \land & 0 \\
\hline
\pi_2(p(T)) \neq \_ \\
\neq \land & \neq \land & 0 \\
\hline
\end{array}\]

\[\equiv (\text{remove rows by (}a \land b) \lor (a \land \neg b) \equiv a)\]
Appendix 1 The Reset/Set Flip-flop

\[ i_{31}(r(T)) = 1 \land i_{11}(r(T)) = 1 \]

\[ \equiv (i_{11} \leftarrow R, i_{31} \leftarrow S, v_4 \rightarrow Q \in \mathcal{F}) \]

\[
\begin{array}{c|c|c}
\pi_4(T) & \pi_4(p(T)) & \pi_4(p(T)) \\
\hline
\_ & \_ & \_ \\
\hline
\_ & \_ & \_ \\
\hline
\_ & \_ & \_ \\
\end{array}
\]

\[ \pi_4 \circ f V \equiv Q(\pi_4(T)) \]

\[
\begin{array}{c|c|c}
\pi_4(T) & \pi_4(p(T)) & \pi_4(p(T)) \\
\hline
\_ & \_ \land \_ & \pi_4(p(T)) \\
\hline
\_ & \_ \land \_ & \_ \\
\hline
\_ & \_ \land \_ & \_ \\
\end{array}
\]

A1.4 The Conformance Checking

To show that \( \pi_4 \circ f o \subseteq \pi_4 \circ f e \), we need to demonstrate the following:

- \( t(r(\pi_4(T))) = t(r(T)) \), so that the values are from the corresponding events.
- Every row in the network function table implies a row or some rows in the requirement table.

From the connections of the variables and the trace functions (see the analysis below) it can be seen that all the value changes on internal variables will be reflected to external variables.

The set of network variables: \( V = \{i_{11}, v_1, i_{21}, i_{22}, v_2, i_{31}, i_{32}, v_3, i_{41}, v_4\} \)

The set of network rules:

\[ \mathcal{F} = \{i_{22} \leftarrow \gamma_1 \leftarrow v_1, i_{32} \leftarrow \gamma_2 \leftarrow v_2, i_{41} \leftarrow \gamma_3 \leftarrow v_3, i_{21} \leftarrow \gamma_4 \leftarrow v_4, \\
i_{11} \leftarrow R, i_{31} \leftarrow S, v_4 \rightarrow Q\} \]

The corresponding value changes of internal and external variables:

- external: \( v_1/i_{22}, v_2/i_{32}, v_3/i_{41} \)
- internal: \( R, Q \rightarrow i_{21}, S \leftarrow v_4 \)

\( v_1/i_{22} \Rightarrow R \) - means if \( v_1/i_{22} \) were changed, \( R \) must have been changed at the same time (NOT gate).
v_2/i_{32} \Rightarrow R \text{ or } Q \text{ or both (AND gate)}

i_{21} \Rightarrow Q \text{ (connection)}

v_{3}/i_{41} \Rightarrow S \text{ or } v_2 \text{ or both } \Rightarrow S \text{ or } R \text{ or Q (OR gate and connections)}

This means that every network event has a corresponding system event. Therefore \( t(r(\pi_0(T)))=t(r(T)) \) and for any system variable \( v \), \( v'(r(\pi_0(T)))=v'(r(T)) \). Thus the requirement table can be simplified as:

\[
\pi_0 \circ f_v = Q(\pi_0(T)) =
\begin{array}{c|c}
\pi_0(T) = & 0 \\
\pi_0(T) \neq _\land & \begin{array}{c}
\pi_0(T) = _\land & \begin{array}{c|c}
\text{S}(r(T)) = 0 & \text{Q}(r(T)) = 0 \\
\text{R}(r(T)) = 0 & \text{Q}(r(T)) = 1 \\
\end{array}
\end{array}
\end{array}
\]

For the grey blocks in both tables (the simplified \( \pi_4 \circ v_4 \) and \( \pi_0 \circ f_v \)), it is obvious that each row in \( \pi_4 \circ v_4 \) implies a row in table \( \pi_0 \circ f_v \). For the first seven rows it can be shown that each row can imply one or several rows in the requirement table for the same output value.

The before value of an output variable in the very first event in a component can take the value of an empty trace defined for that variable, as stated in the “Initialization Condition”.

For each input variable, the before value in the first event can take the initial value of this variable. Looking back at the output value of an empty trace for each variable, we may see that the component TFM descriptions are written under the assumption of \( R_{init}=0 \) \( S_{init}=0 \), this value remains unchanged until the first event happens in the component.

The following is the analysis of the implementation table \( \pi_k \circ f_k \) row by row.

**Row 1).** \( \pi_k(T)= _ \Rightarrow T= _ \text{ or } T \neq _ \Rightarrow 

a) \text{ When } T= , \pi_k(T)=_, \text{ implies the first row in the requirement table.}

b) \text{ When } T \neq , 'Q(r(T)) = Q(\pi_k(p(T))) \text{ (Lemma 4.5)}

\[ = Q( _ )=0 \text{ (IC and } \pi_k(T)=_ \Rightarrow \pi_k(p(T))= _ ) \]

If 'S(r(T))=1, Q'(r(T)) = 'i_{41}(r(T)) = 'v_3(r(T))=1 =Q(T), contradicts itself (because in the first line of the network trace table, Q(T) =0 therefore 'S(r(T)) cannot be 1.

'S(r(T))=0, 'Q(r(T))=0 and Q(T) =0 implies the second or the fourth row in the requirement table.

**Row 2).** \( \pi_k(T) \neq _ \land \pi_k(p(T))= _ \Rightarrow p(T)= _ \text{ or } p(T) \neq _ \Rightarrow 

a) \text{ When } p(T)= , 'Q(r(T))=0 (Q_{init}), 'S(r(T))=0 (S_{init})

b) \text{ When } p(T) \neq , 'Q(r(T))=Q'(r(p(T))) \text{ (Continuity Condition)}

\[ = 'i_{41}(r(p(T))) \text{ (definition of } v_4) \]

\[ = 'v_3(r(p(T))) \text{ (Consistency Condition)} \]

\[ = v_3(r(p(T))) \text{ (definition of } v_3) \]

\[ = v_3(p(T)) \text{ (Output condition)} \]

\[ = v_3( _ )=0 \text{ (definition of } v_3) \]
Appendix 1 The Reset/Set Flip-flop

\[ S(r(T)) = S_{\text{init}} = 0 \quad (\pi_d(p(T)) \neq \_ \Rightarrow \text{no event in OR gate before } t(r(T))) \]

\[ S(r(T)) = 0, \quad Q(r(T)) = 0 \Rightarrow \text{no event in OR gate before } t(r(T)) \]

**Row 3.** \( \pi_s(T) \neq \_ \land \pi_s(p(T)) \neq \_ \land \pi_s(p(T)) = \_ \land S(r(T)) = 0 \)

\[ \pi_d(p(T)) \neq \_ \land \pi_d(p(T)) = \_ \Rightarrow p(T) \neq \_ \Rightarrow \]

\[ Q(r(T)) = \overline{Q}'(r(p(T))) \quad \text{(Continuity Condition)} \]

\[ = Q'(r(\pi_s(p(T)))) \quad \text{(projection definition and } v_4 \rightarrow Q \land i_{21} \leftarrow \gamma_4 \leftarrow v_4) \]

\[ = Q(\pi_s(p(T)) \quad \text{(Output condition)} \]

\[ = Q(\_ ) = 0 \quad \text{(Trace function definition)} \]

\[ S(r(T)) = 0, \quad Q(r(T)) = 0 \Rightarrow \text{no event in OR gate before } t(r(T)) \]

**Row 4.** \( \pi_s(T) \neq \_ \land \pi_s(p(T)) \neq \_ \land \pi_s(p(T)) = \_ \land S(r(T)) = 0 \)

\[ \Rightarrow p(T) \neq \_ \Rightarrow \]

\[ S(r(T)) = 1 \land Q(r(T)) = 0 \Rightarrow \]

\[ R(r(T)) = R_{\text{init}} = 0, \text{ because } \pi_i(p(T)) = 0 \Rightarrow \text{no event in NOT gate before } t(r(T)) \]

**Row 5.** \( \pi_s(T) \neq \_ \land \pi_s(p(T)) \neq \_ \land \pi_s(p(T)) = \_ \land S(r(T)) = 0 \land Q(r(T)) = 0 \)

\[ \Rightarrow p(T) \neq \_ \Rightarrow \]

\[ S(r(T)) = 0 \land Q(r(T)) = 0 \Rightarrow \]

\[ R(r(T)) = R_{\text{init}} = 0, \text{ because at most one event in NOT gate} \]

**Row 6.** \( \pi_s(T) \neq \_ \land \pi_s(p(T)) \neq \_ \land \pi_s(p(T)) = \_ \land S(r(T)) = 1 \land Q(r(T)) = 0 \)

\[ \Rightarrow p(T) \neq \_ \Rightarrow \]

\[ R(r(T)) = R_{\text{init}} = 0, \text{ because at most one event in NOT gate} \]

**Row 7.** \( \pi_s(T) \neq \_ \land \pi_s(p(T)) \neq \_ \land \pi_s(p(T)) = \_ \land Q(r(T)) = 1 \)

\[ \Rightarrow p(T) \neq \_ \Rightarrow \]

\[ R(r(T)) = R_{\text{init}} = 0, \text{ because at most one event in NOT gate} \]

\[ R(r(T)) = 0, \quad Q(r(T)) = 1 \land Q(T) = 1 \Rightarrow \text{no event in OR gate before } t(r(T)) \]

Since the component TFM descriptions used to derive the network TFM description are under the assumption of \( R_{\text{init}} = 0 \) and \( S_{\text{init}} = 0 \), we may conclude that \( \pi_k \circ f_{K_0} \subseteq \pi_0 \circ f_i \) from the above.
Appendix 2

The Temperature Monitor System

A2.1 The TFM Specification of the System Requirements

The input \( c \) is a periodic pulse to trigger the reading of the temperature sensor and refreshing of the display screen.

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( c )</td>
<td>(&lt;\text{boolean}&gt;)</td>
</tr>
<tr>
<td>( tp )</td>
<td>(&lt;\text{float}&gt;)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{avg1} )</td>
<td>(&lt;\text{float}&gt;)</td>
</tr>
<tr>
<td>( \text{avg2} )</td>
<td>(&lt;\text{float}&gt;)</td>
</tr>
<tr>
<td>( \text{current_t} )</td>
<td>(&lt;\text{float}&gt;)</td>
</tr>
</tbody>
</table>

Auxiliary predicates:

\( \text{phas}(T, i) \equiv (L(T) \mod 3 = i) \) describes the phase of the last event of \( T \).

For example, \( \text{phas}(T, 1) \) means that the last event is in phase 1.

- phase 1: read sensor and write to storage.
- phase 2: update the values needed for calculating the average,
- phase 3: calculate the average, and display the required values.

\( \text{tick}(T) \equiv \lnot (c(r(T)) \neq c'(r(T))). \) Although there may be some value changes, they are not considered as events if the value changes are not on clock tick. Events happen only when \( c(r(T)) \neq c'(r(T)). \)

Auxiliary function definition:

\( \text{ri}(i, T) \): the \( i \)th-recent event descriptor in \( T \) (\(<\text{integer}\> \times \langle\text{trace}\rangle \rightarrow \langle\text{event descriptor}\rangle\))

\[
\text{ri}(i, T) =
\begin{cases} 
  i < 1 \lor i > L(T) \lor L(T) = 0 \\
  1 \leq i \leq L(T) \quad r(pn(i, T)) 
\end{cases}
\]

**OUTPUT FUNCTIONS**

\( \text{avg1}(T) =
\begin{array}{|c|c|}
\hline
T=_{}\quad 0.0 \\
T\neq_{} & \quad \begin{array}{|c|c|c|}
\hline
\neg\text{tick}(T) & \text{phas}(T, 0) & L(T)/3 \leq N20 \\
\text{tick}(T) & \hline
\hline
\sum tp'(\text{ri}(i*3, T)) / (L(T)/3) \\
i=1 & \\
\hline
\end{array}
\end{array}
\)
A2.2 The TFM Descriptions of the Constructing Components

The following are some constants, data types and auxiliary functions that will be used in component TFM descriptions.

**CONSTANTS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SAM_INT</td>
<td>&lt;real&gt;</td>
<td>2</td>
<td>the sample or sensor reading interval in minutes</td>
</tr>
<tr>
<td>N20</td>
<td>&lt;int&gt;</td>
<td>10</td>
<td>20/SAM_INT, number of reads within 20 minutes</td>
</tr>
<tr>
<td>N60</td>
<td>&lt;int&gt;</td>
<td>30</td>
<td>60/SAM_INT, number of reads within 60 minutes</td>
</tr>
</tbody>
</table>

**Data type**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>℘(pgs)</td>
<td>pgs is a set of program names and ℘ is power set of pgs</td>
</tr>
</tbody>
</table>

**Auxiliary functions:**

- `mr(pg, pgm, T)`:  
  
  (<program name> × <PGM variable> × <trace> × <event descriptor>)

  It returns the most recent event descriptor in which the program pgm is invoked.

  \[
  \text{mr(pg, pgm, T)} \equiv \\
  \begin{array}{ll}
  \text{T = -} & 0.0 \\
  \text{T ≠ -} : \text{−tick(T)} & \text{tp'(ri(3, T))}
  \\
  \text{tick(T) : phas(T,0)} & \text{current_t(p(T))}
  \\
  \text{−(phas(T,0))} & \text{−(phas(T,0))}
  \end{array}
  \]

For any T ≠ -:

- **Phase:** 1 2 3

- **Tasks:**
  - PGM₁: READ_SENSOR
  - PGM₁: WRITE
  - PGM₁: UPDATE
  - PGM₁: LATEST
  - 20M_OLD
  - PGM₂: REFRESH
  - PGM₂: DISPLAY1
  - PGM₂: DISPLAY2
  - PGM₂: DISPLAY3
  - PGM₃: AVG1
Note: In the above table, the programs in grey are called by other programs within the network; and those in bold are called from outside of the network.
The network of components of this design is shown in Fig. 6.6.

1. Temperature Storage (TS):

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_{11}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>i_{12}</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>i_{13}</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
<tr>
<td>PGM_{1}</td>
<td>{\phi}(pg_{1})</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_{11}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>o_{12}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>o_{13}</td>
<td>&lt;float&gt;</td>
</tr>
</tbody>
</table>

**ACCESS PROGRAMS:**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITE</td>
<td>o_{11}</td>
<td>i_{12}</td>
<td>(PGM_{1};{WRITE}, 'i_{11}, i_{11}', o_{12}', t)</td>
</tr>
<tr>
<td>20M_OLD</td>
<td>o_{12}</td>
<td>i_{13}</td>
<td>(PGM_{1};{20M_OLD}, i_{12}, o_{12}', t)</td>
</tr>
<tr>
<td>60M_OLD</td>
<td>o_{13}</td>
<td></td>
<td>(PGM_{1};{60M_OLD}, i_{13}, o_{13}', t)</td>
</tr>
<tr>
<td>LATEST</td>
<td></td>
<td></td>
<td>(PGM_{1};{LATEST}, o_{13}', t)</td>
</tr>
</tbody>
</table>

pg_{1} = \{WRITE, 20M_OLD, 60M_OLD, LATEST\}

Auxiliary function definition:

\( W(T) := \langle \text{trace} \rangle \rightarrow \langle \text{trace} \rangle \)

\( \text{et}(\text{WRITE} \in \text{PGM}_{1}(T)) \) - returns a trace that contains the events from \( T \) satisfying \( \text{WRITE} \in \text{PGM}_{1}(T) \) in the order that they appear in \( T \).

**OUTPUT FUNCTIONS:**

\( o_{11}(T) = \)

| L(W(T))=0 \lor \neg \text{ex}(20M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
| L(W(T))\geq1 \land \text{ex}(20M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
| \text{ex}(20M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
| \text{ex}(20M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |

\( o_{12}(T) = \)

| T= \_ \lor \neg \text{ex}(60M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
| \text{ex}(60M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
| 60M\_OLD \in \text{PGM}_{1}(T) \land \neg \text{ex}(60M\_OLD \in \text{PGM}_{1}(T)) \) | 0.0 |
Appendix 2 The Temperature Monitor System

<table>
<thead>
<tr>
<th>o₁₀(T) = (latest_temperature)</th>
</tr>
</thead>
<tbody>
<tr>
<td>¬(60M_OLD ∈ PGM₁(τ(T))) o₁₂(p(T))</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{L}(W(T)) = & 0 \lor -\text{ex}(\text{LATEST} \in \text{PGM₁}(T)) \\
\text{L}(W(T)) \geq & 1 \land \\
\text{ex}(\text{LATEST} \in \text{PGM₁}(T)) & \equiv (\text{latest_temperature})
\end{align*}
\]

\*₁¹ the value in this case is not deterministic, either ‘i₁₁(τ(T)), i₁₁’(τ(T))

From the specifications we can see that if WRITE and LATEST are called together, the output value is not deterministic. This is the reason why the scheduler scheduled WRITE and LATEST interleaved, so that the outputs are deterministic and this tables will become proper function tables in the network.

2. Average Calculator Module (AC)

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i₂₁</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>i₂₂</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>i₂₃</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
<tr>
<td>PGM₂</td>
<td>ψ(pg₂)</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>o₂₁ (index)</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>o₂₂ (index)</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>o₂₃ (avg₁)</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>o₂₄ (avg₂)</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>cp₂</td>
<td>ψ(pg₁)</td>
</tr>
</tbody>
</table>

**ACCESS PROGRAMS:**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value in Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPDATE</td>
<td>(PGM₂:{UPDATE}, ‘i₂₁, i₂₁’, ‘i₂₂, i₂₂’, ‘i₂₃, i₂₃’, o₂₁, o₂₂)</td>
</tr>
<tr>
<td>AVG1</td>
<td>(PGM₂:{AVG1}, i₂₁, i₂₃, ‘o₂₃, o₂₃’)</td>
</tr>
<tr>
<td>AVG2</td>
<td>(PGM₂:{AVG2}, i₂₂, i₂₃, ‘o₂₄, o₂₄’)</td>
</tr>
</tbody>
</table>

pg₂ = {UPDATE, AVG1, AVG2}

Auxiliary function definitions

\[ U(T): (<\text{trace}> \rightarrow <\text{trace}>) \equiv \]

\[
\eta(UPDATE \in PGM₂(T)) \text{ returns a trace that contains the events from } T \text{ satisfying } UPDATE \in PGM(e) \text{ in the order that they appear in } T. \\
\]

\[
L(U(T)) \\
\sigma_1(T) \equiv \sum_{i=1}^{i₂₃'}(\text{rk}(i,U(T)))/L(U(T)) \\
L(U(p(T))) \\
\sigma_1^* \equiv \{(\sum_{i=1}^{i₂₃'}(\text{rk}(i,U(p(T)))+*₂¹)/L(U(T)),)
\]

127
\[ i = 1 \]

*\[21\] is either \(i_{23}'(r(T))\) or \(i'_{23}(r(T))\)

\[
\sigma_2(o, i, n, T) \equiv o(p(T)) + (i_{23}'(mr(UPDATE, PGM_2, p(T))) - i'(mr(UPDATE, PGM_2, p(T))))/n
\]

\[
\sigma_2^*(o, n) \equiv o(p(T)) + (*_{22})/n
\]

\[ *_{22} \text{ could be any combination of } i_{23}'(r(T))/i'_{23}(r(T)) - i'(r(T))/i(r(T)) \]

### OUTPUT FUNCTIONS

\[
o_{21}(T) \equiv o_{21}(T) \equiv
\begin{align*}
L(U(T)) &= 0 \quad 0 \\
L(U(T)) \geq 1 & \Rightarrow (UPDATE \in PGM_2(r(T)) \quad N20) \\
& \quad \neg (UPDATE \in PGM_2(r(T))) \quad 0
\end{align*}
\]

\[
o_{22}(T) =
\begin{align*}
L(U(T)) &= 0 \quad 0 \\
L(U(T)) \geq 1 & \Rightarrow (UPDATE \in PGM_2(r(T)) \quad N60) \\
& \quad \neg (UPDATE \in PGM_2(r(T))) \quad 0
\end{align*}
\]

\[
o_{23}(T) =
\begin{align*}
L(U(T)) &= 0 \lor \neg (AVG_1 \in PGM_2())(T) \quad 0.0 \\
L(U(T)) \geq 1 \land \neg (AVG_1 \in PGM_2())(T) & \Rightarrow (UPDATE \in PGM_2(r(T)) \quad L(U(T)) \leq N20) \\
& \quad (AVG_1 \in PGM_2(r(T)) \quad L(U(T)) > N20) \\
& \quad \neg (UPDATE \in PGM_2(r(T)) \quad L(U(T)) \leq N20) \\
& \quad \neg (AVG_1 \in PGM_2(r(T)) \quad L(U(T)) > N20)
\end{align*}
\]

\[
o_{24}(T) =
\begin{align*}
L(U(T)) &= 0 \lor \neg (AVG_1 \in PGM_2())(T) \quad 0.0 \\
L(U(T)) \geq 1 \land \neg (AVG_1 \in PGM_2())(T) & \Rightarrow (UPDATE \in PGM_2(r(T)) \quad L(U(T)) \leq N60) \\
& \quad (AVG_2 \in PGM_2(r(T)) \quad L(U(T)) > N60) \\
& \quad \neg (UPDATE \in PGM_2(r(T)) \quad L(U(T)) \leq N60) \\
& \quad \neg (AVG_2 \in PGM_2(r(T)) \quad L(U(T)) > N60)
\end{align*}
\]

\[
cp_2(T) = \quad \text{(programs called by this component)}
\begin{align*}
T &= \_ \quad \{} \\
T &\neq \Rightarrow (UPDATE \in PGM_2(r(T)) \quad \{\text{LATEST, 20M_OLD, 60M_OLD}\}) \\
& \quad \neg (UPDATE \in PGM_2(r(T))) \quad \{}
\end{align*}
\]

### 3. Scheduler (SC):

\begin{center}
\text{INPUT VARIABLES}
\end{center}

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i_{31})</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

\begin{center}
\text{OUTPUT VARIABLES}
\end{center}
<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
<th>Variable Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp31</td>
<td>$\wp(pg_4)$</td>
<td>set of program names called by this module</td>
</tr>
<tr>
<td>cp32</td>
<td>$\wp(pg_5)$</td>
<td>set of program names called by this module</td>
</tr>
</tbody>
</table>

**OUTPUT FUNCTIONS:**

Variables cp31 and cp32 can be viewed as one variable cp3 with type $\wp(pg_4 \cup pg_5)$:

$$
\text{cp3}(T) \equiv
\begin{align*}
&T=\_ \quad \{\}\nonumber \\
&T\neq \_ \quad \neg \text{tick}(T) \\
&\text{tick}(T) \quad \{\text{READ\_SENSOR}\} \\
&\quad \{\text{REFRESH}\} \\
&\quad \{\text{DISPLAY1, DISPLAY2, DISPLAY3}\} \\
&\quad \text{phas}(T,2) \\
&\quad \text{phas}(T,0)
\end{align*}
$$

Because it will be sent to two different components, a distributor splits it to the following variables:

**cp31(T) ≡**

$$
\text{cp31}(T) \equiv
\begin{align*}
&T=\_ \quad \{\}\nonumber \\
&T\neq \_ \quad \neg \text{tick}(T) \\
&\text{tick}(T) \quad \{\text{READ\_SENSOR}\} \\
&\quad \{\}\nonumber
\end{align*}
$$

**cp32(T) ≡**

$$
\text{cp32}(T) \equiv
\begin{align*}
&T=\_ \quad \{\}\nonumber \\
&T\neq \_ \quad \neg \text{tick}(T) \\
&\text{tick}(T) \quad \{\text{READ\_SENSOR}\} \\
&\quad \{\}\nonumber
\end{align*}
$$

4. **Sensor Reader (SR):**

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i41</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>PGM_4</td>
<td>$\wp(pg_4)$</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
</tbody>
</table>

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>o41</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>cp4</td>
<td>$\wp(pg_4)$</td>
</tr>
</tbody>
</table>

**ACCESS PROGRAMS:**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>in</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>READ_SENSOR</td>
<td>&lt;float&gt;</td>
<td>i41</td>
<td>PGM_4: {READ_SENSOR}, i41, 'o41, o41', t</td>
</tr>
</tbody>
</table>

pg_4 = \{READ\_SENSOR\}

**OUTPUT FUNCTIONS:**
\[ cp_4(T) = \]
\[ \begin{align*}
T &= -_\{} \\
T &\neq _\{} \{ \text{WRITE} \}
\end{align*} \]
\[ o_{41}(T) = \]
\[ \begin{align*}
T &= -_\{} 0.0 \\
T &\neq _i_{41}'(r(T))
\end{align*} \]

5. Temperature Display (TD):

INPUT VARIABLES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_{51}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>i_{52}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>i_{53}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>PGM_5</td>
<td>\varphi(pg_5)</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
</tbody>
</table>

OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>o_{51}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>o_{52}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>o_{53}</td>
<td>&lt;float&gt;</td>
</tr>
<tr>
<td>cp_{51}</td>
<td>\varphi(pg_2)</td>
</tr>
<tr>
<td>cp_{52}</td>
<td>\varphi(pg_1)</td>
</tr>
</tbody>
</table>

ACCESS PROGRAMS:

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>REFRESH</td>
<td>o_{51}</td>
<td>PGM_5:{REFRESH}, 'i_{51}, i_{51}', 'i_{52}, i_{52}', 'i_{53}, i_{53}', t</td>
<td></td>
</tr>
<tr>
<td>DISPLAY1</td>
<td>o_{51}</td>
<td>PGM_5:{DISPLAY1}, i_{51}, 'o_{51}, o_{51}', t</td>
<td></td>
</tr>
<tr>
<td>DISPLAY2</td>
<td>o_{52}</td>
<td>PGM_5:{DISPLAY2}, i_{52}, 'o_{52}, o_{52}', t</td>
<td></td>
</tr>
<tr>
<td>DISPLAY3</td>
<td>o_{53}</td>
<td>PGM_5:{DISPLAY3}, i_{53}, 'o_{53}, o_{53}', t</td>
<td></td>
</tr>
</tbody>
</table>

pg_5 = \{REFRESH, DISPLAY1, DISPLAY2, DISPLAY3\}

Auxiliary function/relation definition:

\[ cr: (<\text{calling program name}> \times <\text{called program name}>) = \]
\[ \{(\text{REFRESH, UPDATE}), (\text{DISPLAY1, AVG1}), (\text{DISPLAY2, AVG2}), \text{DISPLAY}\} \]

\[ \text{Img}(cr)(A) \equiv \{ b \mid \exists a \in A \text{ s.t. } (a, b) \in R \} \] is a set of called programs by the corresponding calling programs in set \( A \).

Problems in this specification:

OUTPUT FUNCTIONS:

\[ o_{51}(T) = \]
\[ T = -_\{} \lor \neg \text{ex}(\text{DISPLAY1} \in \text{PGM}_5())(T) \]
\[ 0.0 \]
### Appendix 2 The Temperature Monitor System

#### \( o_{52}(T) = \)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
<th>( o_{53}(p(T)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = _\lor \neg \text{ex(DISPLAY1 } \in \text{ PGM5}())(T) )</td>
<td>0.0</td>
<td>( \neg \text{DISPLAY1 } \in \text{ PGM5}() )</td>
</tr>
<tr>
<td>( \text{ex(DISPLAY1 } \in \text{ PGM5}())(T) )</td>
<td>( \text{DISPLAY2 } \in \text{ PGM5}() )</td>
<td>( i_{52}'(r(T)) )</td>
</tr>
<tr>
<td>( \text{DISPLAY2 } \in \text{ PGM5}() )</td>
<td>( \neg \text{DISPLAY2 } \in \text{ PGM5}() )</td>
<td>( o_{52}(p(T)) )</td>
</tr>
</tbody>
</table>

#### \( o_{53}(T) = \)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
<th>( o_{53}(p(T)) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = _\lor \neg \text{ex(DISPLAY1 } \in \text{ PGM5}())(T) )</td>
<td>0.0</td>
<td>( \neg \text{DISPLAY1 } \in \text{ PGM5}() )</td>
</tr>
<tr>
<td>( \text{ex(DISPLAY1 } \in \text{ PGM5}())(T) )</td>
<td>( \text{DISPLAY3 } \in \text{ PGM5}() )</td>
<td>( i_{53}'(r(T)) )</td>
</tr>
<tr>
<td>( \text{DISPLAY3 } \in \text{ PGM5}() )</td>
<td>( \neg \text{DISPLAY3 } \in \text{ PGM5}() )</td>
<td>( o_{53}(p(T)) )</td>
</tr>
</tbody>
</table>

#### \( c_{p51}(T) \equiv \)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = _ )</td>
<td>{ }</td>
</tr>
<tr>
<td>( T \neq _ )</td>
<td>{ }</td>
</tr>
</tbody>
</table>

#### \( c_{p52}(T) \equiv \)

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>( T = _ )</td>
<td>{ }</td>
</tr>
<tr>
<td>( T \neq _ )</td>
<td>{ }</td>
</tr>
</tbody>
</table>

#### “Composer” \( C_6: \)

We can see that the output value of \( c_{p6} \) is the union of the three inputs to the composer.

\[ c_{p6}(T) \equiv c_{p2}(T) \cup c_{p4}(T) \cup c_{p51}(T), \] T is a network trace.
A2.3 The Derivation of the Network Behavioural Description

The network trace functions can be calculated by \( \pi^k f_k = f_k(\pi_\delta(T)) \) as follows

\[
\begin{align*}
\text{cp}_{31d}(T) &= \text{cp}_{31}(\pi_3(T)) \\
\pi_3(T) &= _{} \\
\pi_3(T) \neq &\quad -\text{tick}(\pi_3(T)) \\
&\quad \text{tick}(\pi_3(T)) \\
&\quad \text{phas}(\pi_3(T), \, 1) \\
&\quad \text{phas}(\pi_3(T), \, 2) \\
&\quad \text{phas}(\pi_3(T), \, 0) \\
&\equiv (C_3 \text{ is a scheduler so the event rate is the network event rate}) \\
T &= _{} \\
T \neq &\quad -\text{tick}(T) \\
&\quad \text{tick}(T) \\
&\quad \text{phas}(T, \, 1) \\
&\quad \text{phas}(T, \, 2) \\
&\quad \text{phas}(T, \, 0) \\
\text{cp}_{32d}(T) &= \text{cp}_{32}(\pi_3(T)) \\
\pi_3(T) &= _{} \\
\pi_3(T) \neq &\quad -\text{tick}(\pi_3(T)) \\
&\quad \text{tick}(\pi_3(T)) \\
&\quad \text{phas}(\pi_3(T), \, 1) \\
&\quad \text{phas}(\pi_3(T), \, 2) \\
&\quad \text{phas}(\pi_3(T), \, 0) \\
&\equiv \\
\text{o}_{41d}(T) &= \text{o}_{41}(\pi_4(T)) \\
\pi_4(T) &= _{} \\
\pi_4(T) \neq &\quad 0.0 \\
&\quad i_{41}'(r(\pi_4(T))) \\
&\equiv (\text{in a network event READ SENSOR is either called or not called,}
\pi_4(T) = \Leftrightarrow T= \text{ and } \pi_4(T) \neq \Leftrightarrow T \neq \text{ by definition of cp}_{31d}) \\
T &= _{} \\
T \neq &\quad -\text{tick}(T) \\
&\quad \text{tick}(T) \\
&\quad \text{phas}(T, \, 1) \\
&\quad \text{phas}(T, \, 2) \\
&\quad \text{phas}(T, \, 0) \\
&\equiv (\text{PGM}_4 \leftarrow \gamma_{41} \leftarrow \text{cp}_{31} \in \mathcal{H} \Rightarrow \text{PGM}_4(r(T)) = \text{cp}_{31}(r(T)) = \text{cp}_{31}(T) \text{ by Lemma 4.5}) \\
T &= _{} \\
T \neq &\quad -\text{tick}(T) \\
&\quad \text{tick}(T) \\
&\quad \text{phas}(T, \, 1) \\
&\quad \text{phas}(T, \, 1) \\
&\quad i_{41}'(r(T)) \\
&\quad i_{41}(r(T))
Appendix 2 The Temperature Monitor System

\[ cp_{4d}(T) = cp_{41}(\pi_d(T)) \]

\[
\begin{align*}
\pi_d(T) &= _\sim \{ \} \\
\pi_d(T) \neq _\sim \{ \text{WRITE} \}
\end{align*}
\]

= (in a network event READ_SENSOR is either called or not)

\[
\begin{array}{c|c|c}
T &= \_ \{ \} \\
T \neq \_ & \{ \} \\
\text{tick}(T) & \{ \text{WRITE} \} \\
\neg \text{phas}(T, 1) & \{ \}
\end{array}
\]

\[ \equiv (\text{PGM}_4 \leftarrow \gamma_{11} \leftrightarrow cp_{21} \in \mathcal{R} \Rightarrow \text{PGM}_4(r(T)) = cp_{31}(r(T))) \text{ by Lemma 4.5} \]

\[ o_{31a}(T) = o_{31}(\pi_d(T)) = \]

\[
\begin{align*}
\pi_d(T) &= _\sim \\
\neg \text{ex}(\text{DISPLAY}_1 \in \text{PGM}_3())(\pi_d(T)) & \text{0.0} \\
\neg \text{ex}(\text{DISPLAY}_1 \in \text{PGM}_3(\pi_d(T))) & i_{31}'(r(T)) \\
\neg \text{DISPLAY}_1 \in \text{PGM}_3(\pi_d(T)) & o_{31}(p(\pi_d(T)))
\end{align*}
\]

\[ \equiv (\text{PGM}_5 \leftarrow \gamma_{12} \leftrightarrow cp_{32} \in \mathcal{R} \Rightarrow \text{PGM}_5(r(T)) = cp_{32}(T)) \text{ by Lemma 4.5} \]

\[ o_{23a}(T) = o_{23}d(T) \]

\[
\begin{align*}
L(T) &< 3 \\
L(T) &\geq 3 \\
\neg \text{tick}(T) & \text{i}_{31}'(r(T)) \\
\text{tick}(T) & o_{31a}(p(T)) \\
\neg \text{phas}(T, 0) & o_{23a}(T) \\
\text{phas}(T, 0) & o_{51a}(p(T))
\end{align*}
\]

\[ \equiv (i_{31} \leftarrow \gamma_{6} \leftrightarrow o_{23} \in \mathcal{R} \Rightarrow i_{31}'(r(T)) = o_{23}'(r(T)) = o_{23a}(T)) \]

\[ \equiv (L(T) < 2 \leftrightarrow L(U(T)) = 0), \text{ and the false rows are deleted directly} \]

For every \( T \), we check only \text{tick}(T) to see whether the last one is an event. All the previous events are valid events, e.g. \text{tick}(p(T)) is always true)

\[
\begin{array}{c|c}
L(T) & 0.0 \\
L(T) < 3 & \text{L}(W(T)) \leq N_{20} \\
L(T) \geq 3 & \sum_{i=1}^{441}(ri(W(T)))/L(W(T)) \\
\text{tick}(T) & L(W(T)) > N_{20} \\
\text{phas}(T, 0) & o_{23a}(p(T)) + (i_{44}'(mr(WRITE, PGM_1, T))-
\end{array}
\]

133
Comparing the above table with the following one, and noticing the fact \( L(T) < 2 \iff L(U(T)) = 0 \)
we can see that \( o_{51d}(T) \) is equivalent to \( o_{23d}(T) \). Therefore \( o_{23d}(p(T)) \) in the forth row can be replaced by \( o_{51d}(p(T)) \). This is understandable since the 5th component (TD) is just a display device.

\[
o_{23d}(T) =
\begin{align*}
L(T) < 3 & : \neg \text{tick}(T) : \neg \text{phas}(T,0) : L(W(T)) \leq N_{20} \\
L(T) \geq 3 & : \neg \text{tick}(T) : \text{phas}(T,0) : \sum_{i=1}^{i_{41}} \frac{\text{ri}(i,W(T))}{L(W(T))} \leq N_{20} \\
& : \text{L}(W(T)) > N_{20} \\
& : \text{L}(W(T)) \geq L(W(T))
\end{align*}
\]

\[
o_{51d}(T) =
\begin{align*}
L(T) < 3 & : \neg \text{tick}(T) : \neg \text{phas}(T,0) : L(W(T)) \leq N_{20} \\
L(T) \geq 3 & : \text{tick}(T) : \text{phas}(T,0) : \sum_{i=1}^{i_{41}} \frac{\text{ri}(i,W(T))}{L(W(T))} \leq N_{20} \\
& : \text{L}(W(T)) > N_{20} \\
& : \text{L}(W(T)) \geq L(W(T))
\end{align*}
\]

\[\equiv (\text{WRITE and READ SENSOR is always at the same phase, } i_{41} \text{'}(mr(WRITE,PGM1,T)) = i_{41} \text{'}(mr(READ SENSOR,PGM4,T)) = i_{41} \text{'}(r(R(T))), \]
\[i_{41} \text{'}(ri(i,W(T))) = i_{41} \text{'}(ri(i,R(T))))
\]

\[
L(W(T)) = L(R(T))
\]

Similarly:
\[o_{52d}(T) = o_{55}(\pi_5(T)) =
\begin{align*}
L(T) < 3 & : \neg \text{tick}(T) : \neg \text{phas}(T,0) : L(R(T)) \leq N_{60} \\
L(T) \geq 3 & : \text{tick}(T) : \text{phas}(T,0) : \sum_{i=1}^{i_{41}} \frac{\text{ri}(i,R(T))}{L(R(T))} \leq N_{20} \\
& : \text{L}(R(T)) > N_{20} \\
& : \text{L}(R(T)) \geq L(R(T))
\end{align*}
\]
Appendix 2 The Temperature Monitor System

\[ \sum_{i=1}^{41} \left( r_i(R(T)) \right) / L(R(T)) \]
\[ o_{52d}(p(T)) + (i_{41} r(R(T)) - i_{41} r(N60,R(T))) / N60 \]
\[ o_{52d}(p(T)) \]

\[ o_{53d}(T) = o_{53}(\pi(T)) = \]
\[ \pi(T) = _\_ \vee \neg \text{ex(DISPLAY1} \in \text{PGM}_3(\pi(T))) \]
\[ \text{ex(DISPLAY1} \in \text{PGM}_3(\pi(T))) : \text{DISPLAY3} \in \text{PGM}_3(\pi(T)) \]
\[ -(\text{DISPLAY3} \in \text{PGM}_3(\pi(T))) \]

\[ \equiv (\text{By definition of } \text{cp32}, \text{TD starts to have events from phase 2, } \pi(T) = _\_ \Rightarrow L(T) < 2 \]
\[ \neg \text{ex(DISPLAY3} \in \text{PGM}_3(\pi(T))) \Rightarrow L(T) < 3 \]
\[ \text{PGM}_3(\pi(T)) = \text{PGM}_3(\pi(T)) \text{ by Lemma 4.4 and } \text{PGM}_3(\pi(T))=\text{cp32d}(T) \]
\[
\equiv (\text{phas}(T,0) \iff \text{phas}(p(T),2), \ L(T) \geq 3 \iff \ni_{41}(\text{mr}(\text{WRITE, PGM}_1, p(T))) = \ni_{41}(\text{mr}(\text{WRITE, PGM}_1, T)) \text{ and delete the false rows})
\]

\[
\begin{array}{|c|c|}
\hline
L(T) < 3 & 0.0 \\
\hline
L(T) \geq 3 & \neg \text{tick}(T) \\
\hline
\text{tick}(T) & \text{phas}(T,0) \\
\hline
\neg \text{phas}(T,0) & \ni_{41}(\text{mr}(\text{WRITE, PGM}_1, T)) \\
\hline
\end{array}
\]

\[
\equiv \ni_{41}(\text{mr}(\text{WRITE, PGM}_1, T)) = \ni_{41}(\text{mr}(\text{READ SENSOR, PGM}_4, T)) = \ni_{41}(\tau(R(T)),
\]

\[
\begin{array}{|c|c|}
\hline
L(T) < 3 & 0.0 \\
\hline
L(T) \geq 3 & \neg \text{tick}(T) \\
\hline
\text{tick}(T) & \text{phas}(T,0) \\
\hline
\neg \text{phas}(T,0) & \ni_{41}(\tau(R(T))) \\
\hline
\end{array}
\]

\[
\text{cp}_{51d}(T) = \text{cp}_{51}(\pi_5(T)) =
\]

\[
\begin{array}{|c|}
\hline
\pi_5(T) = _{} & \{\} \\
\hline
\pi_5(T) \neq _{} & \text{Img}(\text{cr})(\text{PGM}_4(\tau(\pi_5(T)))) \\
\hline
\end{array}
\]

\[
\equiv (\text{PGM}_4(\tau(\pi_5(T)) = \text{PGM}_4(\tau(T)) \text{ by Lemma 4.4})
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \text{Img}(\text{cr})(\text{PGM}_4(\tau(T))) \\
\hline
\end{array}
\]

\[
\equiv (\text{PGM}_5 \leftarrow \gamma_{12} \leftarrow \text{cp}_{52} \in \mathcal{Y} \Rightarrow \text{PGM}_5(\tau(T)) = \text{cp}_{52}(\tau(T)) = \text{cp}_{52d}(T) \text{ by Lemma 4.5})
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \text{REFRESH} \in \text{PGM}_5(\tau(\pi_5(T))) \\
\hline
\neg (\text{REFRESH} \in \text{PGM}_5(\tau(\pi_5(T)))) & \text{\{LATEST\}} \\
\hline
\end{array}
\]

\[
\equiv (\text{TD has events at phase 2, PGM}_4(\tau(\pi_5(T)) = \text{PGM}_4(\tau(T)) \text{ by Lemma 4.4})
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \text{REFRESH} \in \text{PGM}_4(\tau(T)) \\
\hline
\neg (\text{REFRESH} \in \text{PGM}_4(\tau(T))) & \text{\{LATEST\}} \\
\hline
\end{array}
\]

\[
\equiv (\text{PGM}_5 \leftarrow \gamma_{12} \leftarrow \text{cp}_{52} \in \mathcal{Y} \Rightarrow \text{PGM}_5(\tau(T)) = \text{cp}_{52}(\tau(T)) = \text{cp}_{52d}(T) \text{ by Lemma 4.5})
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \text{REFRESH} \in \text{PGM}_5(\tau(T)) \\
\hline
\neg (\text{REFRESH} \in \text{PGM}_5(\tau(T))) & \text{\{LATEST\}} \\
\hline
\end{array}
\]

\[
\text{cp}_{2d}(T) =
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \neg \text{tick}(T) \\
\hline
\end{array}
\]

\[
\text{cp}_{2d}(T) =
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \neg \text{tick}(T) \\
\hline
\end{array}
\]

\[
\text{cp}_{2d}(T) =
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \neg \text{tick}(T) \\
\hline
\end{array}
\]

\[
\text{cp}_{2d}(T) =
\]

\[
\begin{array}{|c|}
\hline
L(T) < 2 & \{\} \\
\hline
L(T) \geq 2 & \neg \text{tick}(T) \\
\hline
\end{array}
\]

\[
\text{cp}_{2d}(T) =
\]
### Appendix 2 The Temperature Monitor System

#### cp₄δ(T) =

<table>
<thead>
<tr>
<th>T =</th>
<th>{WRITE}</th>
</tr>
</thead>
<tbody>
<tr>
<td>T ≠ _ ∧ ¬tick(T)</td>
<td>{WRITE}</td>
</tr>
<tr>
<td>tick(T)</td>
<td>{LATEST}</td>
</tr>
<tr>
<td>¬phas(T, 1)</td>
<td>{}</td>
</tr>
</tbody>
</table>

#### cp₅δ(T) =

| L(T) < 2 | {} |
| L(T) ≥ 2 | {} |
| ¬tick(T) | {} |
| tick(T) | {LATEST} |
| ¬phas(T, 1) | {} |

#### cp₆δ(T) = cp₂δ(T) ∪ cp₄δ(T) ∪ cp₅δ(T) =

| L(T) = 0 | {} |
| L(T) ≥ 1 | {} |
| ¬tick(T) | {} |
| tick(T) | {LATEST} |
| ¬phas(T, 1) | {} |

#### o₁₁δ(T) = o₁₁(π₁(T)) =

$$\pi_1(T) = _- \lor \neg \text{ex}(20\text{M}_\text{OLD} \in \text{PGM}_5())(\pi_1(T))$$

$$\text{ex}(20\text{M}_\text{OLD} \in \text{PGM}_5())(\pi_1(T)) \land 20\text{M}_\text{OLD} \in \text{PGM}_1(\tau(\pi_1(T))) \Rightarrow L(\text{W}(\pi_1(T)) < i_{12}(\tau(\pi_1(T))))$$

$$L(\text{W}(\pi_1(T)) \geq i_{12}(\tau(\pi_1(T))))$$

$$\neg (20\text{M}_\text{OLD} \in \text{PGM}_1(\tau(\pi_1(T))))$$

$$\equiv (\text{PGM}_1(\tau(\pi_1(T))) = \text{PGM}_1(\tau(T)), i_{12}(\tau(\pi_1(T))) = i_{12}(\tau(T))) \text{ by Lemma 4.4}$$

$$L(\text{W}(\pi_1(T))) = L(\text{W}(T)) \text{ by definition of W}$$

$$\neg \text{ex}(20\text{M}_\text{OLD} \in \text{PGM}_5())(\pi_1(T)) \Leftrightarrow L(T) < 2$$

$$\pi_1(T) = \Leftrightarrow T = \_ \text{ and } \pi_1(T) \neq \Leftrightarrow T \neq \_ \text{ by definition of cp₄₁δ)$$

#### L(T) < 2

| 20\text{M}_\text{OLD} \in \text{PGM}_1(\tau(T)) | 0.0 |
| : L(\text{W}(\pi_1(T)) < i_{12}(\tau(\pi_1(T))) | 0.0 |
| : L(\text{W}(\pi_1(T)) \geq i_{12}(\tau(\pi_1(T))) | 0.0 |

$$\equiv (\text{PGM}_1 \leftarrow _\gamma_9 \leftarrow \text{cp}_0 \in \mathcal{R} \Rightarrow \text{PGM}_1(\tau(T)) = \text{cp}_0(\tau(T)))$$

#### L(T) ≥ 2

| 20\text{M}_\text{OLD} \in \text{PGM}_1(\tau(T)) | 0.0 |
| : L(\text{W}(\pi_1(T)) < i_{12}(\tau(\pi_1(T))) | 0.0 |
| : L(\text{W}(\pi_1(T)) \geq i_{12}(\tau(\pi_1(T))) | 0.0 |

$$\equiv (i_{11} \leftarrow _\gamma_1 \leftarrow o_{41} \in \mathcal{R} \Rightarrow i_{11} \leftarrow (\text{rn}(i, \text{W}(T)))) = o_{11\delta}(\text{rn}(i, \text{W}(T))) \text{ by Lemma 4.5}$$
\[ i_{12} \leftarrow \gamma_{2} \leftarrow o_{21} \in \mathcal{H} \implies i_{12}(r(T)) = o_{21d}(T) \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(T) &lt; 2 )</td>
<td>(-\text{tick}(T) )</td>
<td>( \neg \text{phas}(T, 2) )</td>
</tr>
<tr>
<td>( L(T) \geq 2 ) &amp; ( \text{tick}(T) ) &amp; ( \text{phas}(T, 2) ) &amp; ( L(W(T)) &lt; N^{20} ) &amp; ( L(W(T)) \geq N^{20} ) &amp; ( \neg \text{phas}(T, 2) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ (L(T) < 2) (L(W(T)) < N^{20}) \]

\[ o_{12d}(T) = o_{12}(\pi_{1}(T)) = \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(T) &lt; 2 )</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>( L(T) \geq 2 ) &amp; ( \neg \text{tick}(T) ) &amp; ( \text{phas}(T, 2) ) &amp; ( L(W(T)) &lt; N^{60} ) &amp; ( L(W(T)) \geq N^{60} ) &amp; ( \neg \text{phas}(T, 2) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ o_{13d}(T) = \pi_{i} \circ o_{13} = o_{13}(\pi_{1}(T)) = \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(W(\pi_{1}(T))) = 0 \lor \neg \text{ex}(\text{LATEST} \in \text{PGM}<em>{1}(\pi</em>{1}(T))) )</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>( L(W(\pi_{1}(T))) \geq 1 \land \text{ex}(\text{LATEST} \in \text{PGM}<em>{1}(\pi</em>{1}(T))) ) &amp; ( \text{LATEST} \in \text{PGM}<em>{1}(\pi</em>{1}(T)) ) &amp; ( \text{WRITE} \in \text{PGM}<em>{1}(\pi</em>{1}(T)) ) &amp; ( \neg (\text{WRITE} \in \text{PGM}<em>{1}(\pi</em>{1}(T))) ) &amp; ( \neg (\text{LATEST} \in \text{PGM}<em>{1}(\pi</em>{1}(T))) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ (L(W(\pi_{1}(T))) = L(W(T)), \text{mr}(\text{WRITE}, \text{PGM}_{1}, \pi_{1}(T)) = \text{mr}(\text{WRITE}, \text{PGM}_{1}, p(T)) \]

\[ \text{PGM}_{1}(\pi_{1}(T)) = \text{PGM}_{1}(\pi(T)) \]

\[ (PGM_{1} \leftarrow \gamma_{9} \leftarrow c_{p_{6}} \in \mathcal{H} \Rightarrow \text{PGM}_{1}(r(T)) = c_{p_{6}}(T) \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(T) &lt; 2 )</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>( L(T) \geq 2 ) &amp; ( \neg \text{tick}(T) ) &amp; ( \text{phas}(T, 2) ) &amp; ( \neg \text{phas}(T, 1) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ (i_{11} \leftarrow \gamma_{1} \leftarrow o_{41} \in \mathcal{H} \Rightarrow i_{11}(\text{mr}(\text{WRITE}, \text{PGM}_{1}, T)) = o_{41}(\text{mr}(\text{WRITE}, \text{PGM}_{1}, T)) \]

<table>
<thead>
<tr>
<th>Condition</th>
<th>True</th>
<th>False</th>
</tr>
</thead>
<tbody>
<tr>
<td>( L(T) &lt; 2 )</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>( L(T) \geq 2 ) &amp; ( \neg \text{tick}(T) ) &amp; ( \text{phas}(T, 2) ) &amp; ( \neg \text{phas}(T, 2) )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[ \text{PGM}_{1}(\pi(T)) = \text{PGM}_{1}(\pi(T)) \]

\[ (\text{PGM}_{1} \leftarrow \gamma_{9} \leftarrow c_{p_{6}} \in \mathcal{H} \Rightarrow \text{PGM}_{1}(r(T)) = c_{p_{6}}(T) \]
Appendix 2 The Temperature Monitor System

\[ o_{21d}(T) = o_{21}(\pi_2(T)) \equiv \]
\[ \begin{array}{c}
L(U(\pi_2(T))) = 0 \\
L(U(\pi_2(T))) \geq 1
\end{array} \]

\[ (\text{UPDATE} \in \text{PGM}_2(\tau(\pi_2(T)))) \]

\[ \neg(\text{UPDATE} \in \text{PGM}_2(\tau(\pi_2(T)))) \]

\[ \equiv (L(U(\pi_2(T))) = 0 \iff L(T) < 2 \land L(U(\pi_2(T))) \geq 1 \iff L(T) \geq 2 \text{ by definition of } \text{cp}_{51d}) \]

\[ L(T) < 2 \]

\[ L(T) \geq 2 \]

\[ \neg(\text{tick}(T)) \]

\[ \text{tick}(T) : \text{phas}(T, 2) \]

\[ \neg\text{phas}(T, 2) \]

\[ o_{22d}(T) = o_{22}(\pi_2(T)) \equiv \]

\[ \begin{array}{c}
L(U(\pi_2(T))) = 0 \\
\land \exists \text{AVG}1 \in \text{PGM}_2(\pi_2(T)) \land \text{ex}(\text{AVG}1 \in \text{PGM}_2(\pi_2(T)))
\end{array} \]

\[ \neg(\text{AVG}1 \in \text{PGM}_2(\pi_2(T))) \]

\[ \equiv (L(U(\pi_2(T))) = 0 \iff L(T) < 2 - \text{definition of } U \text{ and } \text{cp}_{52} \]

\[ \exists \text{AVG}1 \in \text{PGM}_2(\pi_2(T)) \iff L(T) \geq 3 \text{ definition of } \text{cp}_{52} \]

\[ \text{PGM}_2(\tau(\pi_2(T))) = \text{PGM}_2(\tau(T)) - \text{Lemma 4.4}, \]

\[ \sigma_1(\pi_2(T)) = \sigma_1(T), \sigma_2(\pi_{23}, \pi_{21}, \pi_{20}, \pi_2(T)) = \sigma_2(\pi_{23}, \pi_{21}, \pi_{20}, \pi(T)) - \sigma_1, \sigma_2 \text{ definition} \]

\[ L(T) < 3 \]

\[ L(T) \geq 3 \]

\[ \text{AVG}1 \in \text{PGM}_2(\pi_2(T)) \]

\[ \text{UPDATE} \in \text{PGM}_2(\tau(\pi_2(T))) \]

\[ L(U(T)) \leq \text{N}_20 \]

\[ L(U(T)) > \text{N}_20 \]

\[ \neg(\text{UPDATE} \in \text{PGM}_2(\tau(\pi_2(T)))) \]

\[ L(U(T)) \leq \text{N}_20 \]

\[ L(U(T)) > \text{N}_20 \]

\[ \neg(\text{AVG}1 \in \text{PGM}_2(\tau(T))) \]

\[ \equiv (\text{PGM}_2 \leftarrow \gamma_{10} \leftarrow \text{cp}_{51} \in \mathcal{H} \Rightarrow \text{PGM}_2(\tau(T)) = \text{cp}_{51}(\tau(T)) = \text{cp}_{51d}(T) \text{ by Lemma 4.5} \]

\[ L(U(T)) = 0 \iff L(T) < 2) \]

\[ L(T) < 3 \]

\[ 0.0 \]

\[ L(T) \geq 3 \]

\[ 0.0 \]
\[ L(U(T)) \geq 3 \Rightarrow \text{tick}(T) \text{ phas}(T, 0) \]
\[ L(U(T)) \leq N20 \]
\[ L(U(T)) > N20 \]
\[ \neg \text{phas}(T, 0) \]

\[ \neg \text{tick}(T) \]

\[ \sigma 1(T) \]
\[ \sigma 2(o_{23}, t_{11}, N20, T) \]
\[ o_{23d}(p(T)) \]

\[ L(T) < 3 \]
\[ \text{tick}(T) \text{ phas}(T, 0) \]
\[ L(U(T)) \leq N20 \]
\[ L(U(T)) > N20 \]
\[ \neg \text{phas}(T, 0) \]
\[ \neg \text{tick}(T) \]

\[ 0.0 \]

\[ \sigma 1(T) \]
\[ \sigma 2(o_{23}, t_{11}, N20, T) \]
\[ o_{23d}(p(T)) \]

\[ L(U(T)) \]
\[ \sigma 1(T) \]
\[ \sigma 2(o_{23}, t_{11}, N20, T) \]
\[ o_{23d}(p(T)) \]

\[ \sum_{i=1}^{i23} (ri(i, U(T))) / L(U(T)) \]

\[ \sum_{i=1}^{i41} (ri(i, W(T)))/ L(W(T)) \]

Because: \( i_{23} \leftarrow \gamma_6 \leftarrow o_{13} \in \mathcal{R} \Rightarrow i_{23}'(ri(i, U(T))) = o_{13}'(ri(i, U(T))) \)
\[ L(T) \mod 3 = 0 \Rightarrow L(U(T)) = L(W(T)) \]
\[ o_{13}'(ri(i, U(T))) = i_{41}'(ri(i, W(T))) \]

For any \( U(T), L(U(T)) \mod 3 = 2 \) – definition of \( \text{cp}_{32} \)

\[ \sigma 2(o_{23}, t_{11}, N20, T) \equiv (i_{23} \leftarrow \gamma_4 \leftarrow o_{11}, i_{23} \leftarrow \gamma_6 \leftarrow o_{13}) \]
\[ o_{23d}(p(T)) + (i_{23}'(mr(UPDATE, PGM2, p(T)))-i'(mr(UPDATE, PGM2, p(T))))/ N20 \]
\[ = o_{23d}(p(T)) + (i_{41}'(mr(WRITE, PGM1, p(T)))-i_{41}'(ri(N20, W(T)))/ N20 \]

\[ o_{23d}(T) = o_{23d}(\pi_2(T)) \equiv \]

\[ L(W(T)) \]
\[ \sum_{i=1}^{i41} (ri(i, W(T)))/ L(W(T)) \]

\[ o_{23d}(T) = o_{23d}(\pi_2(T)) \equiv \]

\[ o_{24d}(T) = o_{24d}(\pi_2(T)) \equiv \]
Appendix 2 The Temperature Monitor System

### A2.4 The result comparison

The output functions of the network outputs:

By network rules $i_{41} \leftarrow t_p, o_{51} \rightarrow a v g_1, o_{52} \rightarrow a v g_2$ and $o_{53} \rightarrow c u r r e n t_t$:

$$o_{51d}(T) = \pi_{5^0} q_{51} = o_{51}(\pi_5(T))$$

- **Case 1:** $L(T) < 3$
  - $L(T) \geq 3$
    - tick(T) : phas(T,0) : $L(R(T)) \leq N_{20}$
      - $L(R(T)) > N_{20}$
        - $\neg$phas(T,0)

$$o_{52d}(T) = \pi_{5^0} q_{52} = o_{52}(\pi_5(T))$$

- **Case 1:** $L(T) < 3$
  - $L(T) \geq 3$
    - tick(T) : phas(T,0) : $L(R(T)) \leq N_{60}$
      - $L(R(T)) > N_{60}$
        - $\neg$phas(T,0)
According to Lemma 5.1, we need to check whether \( \pi_5^o o_{51} = \pi_5^o o_{53} = 0.0 \)  

To show that \( \pi_5^o f_{50} \subseteq \pi_5^o f_e \), every row in the network function table should imply a row or some rows in the requirements table.

\[ \pi_5^o \text{avg1} \equiv \text{avg1}(\pi_5^o(T)) \equiv \]

Compare with the following table:

\[ o_{51d}(T) = \pi_5^o o_{51} = 0.0 \]
Row 3)
If the last event of T is in the third phase, the number of reads will be \(L(T)/3\), i.e. \(L(T)/3 = L(R(T))\), so row 3) in table \(\pi_5^{o_51}\) is subsumed by row 3) in the requirement table \(\pi_0^{o}\text{avg1}\).

Row 4)
As defined in cp3, read is always happen in the first phase of a period, in phase 3 the most recent read is in the third recent event, \(ri(3,T) = ri(R(T))\). So row 4) in table \(\pi_5^{o_51}\) is subsumed by row 4) in the requirement table \(\pi_0^{o}\text{avg1}\).

Similarly \(\pi_5^{o_52} \subseteq \pi_0^{o}\text{avg2}\), \(\pi_5^{o_53} \subseteq \pi_0^{o}\text{current}\_t\) can be proved.
Appendix 3
The Flight Reservation System

A3.1 The Definition and Specification of Data Types

A3.1.1 Definitions of some simple data types

At first we introduce some simple types and constants for the type checking in advance to avoid some run-time error. For example, a flight number is a string but not an arbitrary string; it can be fixed as a 2 alphabetic and 4 digit combination such as CA0966. A customer id can also be of such type.

**TYPES:**

<table>
<thead>
<tr>
<th>Type Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;num&gt;</td>
<td>&lt;string&gt;, the combination of two characters and 4 digits</td>
</tr>
<tr>
<td>&lt;city&gt;</td>
<td>&lt;string&gt;, 3 alphabetic combination</td>
</tr>
<tr>
<td>&lt;date&gt;</td>
<td>&lt;string&gt;, format: &lt;dd&gt;/&lt;mm&gt;/&lt;yyyy&gt;</td>
</tr>
<tr>
<td>&lt;time&gt;</td>
<td>&lt;string&gt;, format: &lt;hh&gt;:&lt;min&gt;</td>
</tr>
<tr>
<td>&lt;status&gt;</td>
<td>{empty, reserved, no enough seats, cancelled}</td>
</tr>
</tbody>
</table>

Here the definitions of day and month are rough. For example “30/02/2011” or “31/04/2010” are considered as valid dates.

**CONSTANTS**

<table>
<thead>
<tr>
<th>Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>φ₀</td>
<td>&lt;string&gt;, φ₀ = “”</td>
</tr>
<tr>
<td>φₖ</td>
<td>&lt;city&gt;, φₖ = “OOO”</td>
</tr>
<tr>
<td>φₙ</td>
<td>&lt;num&gt;, φₙ = “OO0000”</td>
</tr>
<tr>
<td>φₐ</td>
<td>&lt;date&gt;, φₐ = “00/00/0000”</td>
</tr>
<tr>
<td>φₜ</td>
<td>&lt;time&gt;, φₜ = “00:00”</td>
</tr>
</tbody>
</table>

These constants will be used to specify the values of empty traces in the trace functions according to the type of the defined variables.

A3.1.2 Specifications of some complicated data types

To hide the representation of some data structures, we specify them as modules that have operations revealing or changing the state of each object of this type.

1 GenericFlight

This module hides the representation of a GenericFlight object. An object of this type describes the general schedule of a class of flights with the same flight number.
Appendix 3 The Flight Reservation System

We assume that all objects with the same flight number will also have the same departure city, departure time, capacity, etc. In this case, for simplification there is only SET and GET operations except in the creation and deletion of programs. The SET operation will add all the data when the flight is created and do not allow alteration. To alter, one should delete this object and create a new one.

Output Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.fno</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.depcity</td>
<td>&lt;city&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.descity</td>
<td>&lt;city&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.deptime</td>
<td>&lt;time&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.capacity</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>in2</td>
<td>&lt;city&gt;</td>
</tr>
<tr>
<td>in3</td>
<td>&lt;city&gt;</td>
</tr>
<tr>
<td>in4</td>
<td>&lt;time&gt;</td>
</tr>
<tr>
<td>in5</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>in6</td>
<td>&lt;GenericFlight&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Oname</th>
<th>in</th>
<th>value</th>
<th>Abbreviated Event descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>creatGflight</td>
<td>&lt;id&gt;</td>
<td>in6</td>
<td></td>
<td>(PGM:creatGflight)</td>
</tr>
<tr>
<td>deletGflight</td>
<td>&lt;id&gt;</td>
<td>in6</td>
<td></td>
<td>(PGM:deletGflight)</td>
</tr>
<tr>
<td>get</td>
<td>&lt;id&gt;</td>
<td></td>
<td>fno, depcity, descity, deptime, capacity</td>
<td>(PGM:get, fno’, depcity’, descity’, deptime’, capacity’)</td>
</tr>
</tbody>
</table>

OUTPUT FUNCTIONS:

fno(T) ≡

\[
\begin{array}{c|l|c}
T = \_ & PGM(\tau(T)) = creatGflight & \phi_s \\
T \neq \_ & PGM(\tau(T)) = set & in_1'(\tau(T)) \\
& PGM(\tau(T)) = deletGflight & fno(p(T)) \\
& PGM(\tau(T)) = get & \\
\end{array}
\]

depcity(T) ≡

\[
\begin{array}{c|l|c}
T = \_ & PGM(\tau(T)) = creatGflight & \phi_s \\
T \neq \_ & PGM(\tau(T)) = set & in_2'(\tau(T)) \\
& PGM(\tau(T)) = deletGflight & \\
\end{array}
\]

145
One object of this type characterizes a class of flights with the same flight number, start time and city, capacity, end time and city but on different days. It will be created, set and deleted by service operators.

2 SpecificFlight

This module hides the representation of a SpecificFlight object. It is an instance of a generic flight for a specific day.

Output Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.fno</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.date</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.fst</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.valid</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>in2</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>in3</td>
<td>&lt;boolean&gt;</td>
</tr>
</tbody>
</table>
### Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Oname</th>
<th>in</th>
<th>value</th>
<th>Abbreviated Event descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>creatSflight</td>
<td>&lt;id&gt;</td>
<td>in4,</td>
<td></td>
<td>(PGM:creatSflight)</td>
</tr>
<tr>
<td>deletSflight</td>
<td>&lt;id&gt;</td>
<td>in5</td>
<td></td>
<td>(PGM:deletSflight)</td>
</tr>
<tr>
<td>set</td>
<td>&lt;id&gt;</td>
<td>in1, in2, in3</td>
<td></td>
<td>(PGM:set, fno', 'in1, in1', 'in2, in2', valid', 'in3, in3')</td>
</tr>
<tr>
<td>setFseats</td>
<td>&lt;id&gt;</td>
<td>in4</td>
<td></td>
<td>(PGM:setFseats, fst', 'in4, in4')</td>
</tr>
<tr>
<td>unsetFseats</td>
<td>&lt;id&gt;</td>
<td>in4</td>
<td></td>
<td>(PGM:unsetFseats, fst', 'in4, in4')</td>
</tr>
<tr>
<td>get</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td>(PGM:get, fno', date', fst', valid')</td>
</tr>
</tbody>
</table>

### OUTPUT FUNCTIONS:

#### fno(T) ≡

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T = _ ∨ PGM(r(T))=creatSflight</td>
<td></td>
</tr>
<tr>
<td>T ≠ _ ∧</td>
<td>PGM(r(T))=set</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight</td>
</tr>
<tr>
<td>¬(PGM(r(T))=set ∧</td>
<td>PGM(r(T))=creatSflight ∧</td>
</tr>
<tr>
<td>PGM(r(T))=deletSflight)</td>
<td>PGM(r(T))=set ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight</td>
</tr>
<tr>
<td></td>
<td>¬(PGM(r(T))=set ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=creatSflight ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight)</td>
</tr>
<tr>
<td></td>
<td>fno(p(T))</td>
</tr>
</tbody>
</table>

#### date(T) ≡

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T = _ ∨ PGM(r(T))=creatSflight</td>
<td></td>
</tr>
<tr>
<td>T ≠ _ ∧</td>
<td>PGM(r(T))=set</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight</td>
</tr>
<tr>
<td>¬(PGM(r(T))=set ∧</td>
<td>PGM(r(T))=creatSflight ∧</td>
</tr>
<tr>
<td>PGM(r(T))=deletSflight)</td>
<td>PGM(r(T))=set ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight</td>
</tr>
<tr>
<td></td>
<td>¬(PGM(r(T))=set ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=creatSflight ∧</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=deletSflight)</td>
</tr>
<tr>
<td></td>
<td>date(p(T))</td>
</tr>
</tbody>
</table>

#### valid(T) ≡

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T = _</td>
<td>false</td>
</tr>
<tr>
<td>T ≠ _ ∧</td>
<td>PGM(r(T))=set</td>
</tr>
<tr>
<td></td>
<td>in3′(r(T))</td>
</tr>
<tr>
<td>¬(PGM(r(T))=set)</td>
<td>valid(p(T))</td>
</tr>
</tbody>
</table>

#### fst(T) ≡

<table>
<thead>
<tr>
<th>Condition</th>
<th>Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>T = _ ∨ PGM(r(T))=creatSflight</td>
<td></td>
</tr>
<tr>
<td>T ≠ _ ∧</td>
<td>PGM(r(T))=deletSflight</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=set</td>
</tr>
<tr>
<td></td>
<td>setFseats</td>
</tr>
<tr>
<td></td>
<td>fst(p(T)) ≥ in4′(r(T))</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=unsetFseats</td>
</tr>
<tr>
<td></td>
<td>fst(p(T)) &lt; in4′(r(T))</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=get</td>
</tr>
<tr>
<td></td>
<td>PGM(r(T))=set</td>
</tr>
<tr>
<td></td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>fst(p(T)) − in4′(r(T))</td>
</tr>
<tr>
<td></td>
<td>fst(p(T))</td>
</tr>
<tr>
<td></td>
<td>fst(p(T)) + in4′(r(T))</td>
</tr>
<tr>
<td></td>
<td>fst(p(T))</td>
</tr>
</tbody>
</table>

<id>.fno(T) ≡ fno(T<id>)

<id>.depdate(T) ≡ depdate(T<id>)
\begin{align*}
<id>\cdot \text{valid}(T) & \equiv \text{valid}(T_{<id>}) \\
<id>\cdot \text{fst}(T) & \equiv \text{fst}(T_{<id>}) \\
\end{align*}

3 Reservation

Output Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.cid</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.fno</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.depdate</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.bookdate</td>
<td>&lt;date&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{in}_1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>\text{in}_2</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>\text{in}_3</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>\text{in}_4</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>\text{in}_5</td>
<td>&lt;Reservation&gt;</td>
</tr>
<tr>
<td>t</td>
<td>&lt;real&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Oname</th>
<th>in</th>
<th>value</th>
<th>Abbreviated Event descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{createReservation}</td>
<td>&lt;id&gt;</td>
<td>\text{in}_5</td>
<td>(PGM: createReservation)</td>
<td></td>
</tr>
<tr>
<td>\text{deleteReservation}</td>
<td>&lt;id&gt;</td>
<td>\text{in}_1, \text{in}_2, \text{in}_3, \text{in}_4</td>
<td>(PGM: deleteReservation)</td>
<td></td>
</tr>
<tr>
<td>set</td>
<td>&lt;id&gt;</td>
<td>\text{in}_1, \text{in}_2, \text{in}_3, \text{in}_4</td>
<td>(PGM: set, cid', in1, in1', fno, in2, in2', depdate, in3, in3', bookdate', in4, in4',)</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>&lt;id&gt;</td>
<td>cid, fno, depdate, bookdate</td>
<td>(PGM: get, cid', fno', depdate', bookdate')</td>
<td></td>
</tr>
</tbody>
</table>

OUTPUT FUNCTIONS:

\begin{align*}
\text{cid}(T) & \equiv \\
T & = \_ \\
T & \neq \_ \\
\wedge & \quad (PGM(r(T)) = \text{set}) \\
\phi_n & = \text{in}_1'(r(T)) \\
\phi_n & = \text{cid}(p(T)) \\
\phi_n & = \text{in}_2'(r(T)) \\
\phi_n & = \text{fno}(p(T)) \\
\end{align*}

\begin{align*}
\text{fno}(T) & \equiv \\
T & = \_ \\
T & \neq \_ \\
\wedge & \quad (PGM(r(T)) = \text{set}) \\
\phi_n & = \text{in}_1'(r(T)) \\
\phi_n & = \text{in}_2'(r(T)) \\
\phi_n & = \text{fno}(p(T)) \\
\phi_d & = \text{in}_3'(r(T)) \\
\end{align*}
Appendix 3 The Flight Reservation System

\[ -(\text{PGM}(\tau(T)) = \text{set}) \quad \text{depdate}(p(T)) \]

\[
\begin{array}{c|c|c}
&T=\_&\phi_d \\
T \neq \_ & \text{PGM}(\tau(T)) = \text{set} & \text{in}_4'(\tau(T)) \\
&-(\text{PGM}(\tau(T)) = \text{set}) & \text{bookdate}(p(T)) \\
\end{array}
\]

\<id>.cid(T) \equiv \text{cid}(T_{id})
\<id>.fno(T) \equiv \text{fn}(T_{id})
\<id>.depkdate(T) \equiv \text{depdate}(T_{id})
\<id>.bookdate(T) \equiv \text{bookdate}(T_{id})

One reservation object represents one or more seats in a specific flight.

4 Customer
This module hides the representation of a Customer object.

Output Variables

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;id&gt;.cid</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.name</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.tel</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>&lt;id&gt;.snum</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

Input Variables

<table>
<thead>
<tr>
<th>Variable Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>in_1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>in_2</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>in_3</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>in_4</td>
<td>&lt;integer&gt;</td>
</tr>
<tr>
<td>in_5</td>
<td>&lt;Customer&gt;</td>
</tr>
</tbody>
</table>

Access Programs

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Oname</th>
<th>in</th>
<th>value</th>
<th>Abbreviated Event descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>creatCustomer</td>
<td>&lt;id&gt;</td>
<td></td>
<td></td>
<td>(PGM:creatCustomer)</td>
</tr>
<tr>
<td>deletCustomer</td>
<td></td>
<td>in_5</td>
<td></td>
<td>(PGM:deletCustomer)</td>
</tr>
<tr>
<td>set</td>
<td>&lt;id&gt;</td>
<td>in_1, in_2, in_3, in_4</td>
<td>(PGM:set, cid', 'in_1, in_1', name', 'in_2, in_2', tel', 'in_3, in_3', snum', 'in_4, in_4')</td>
<td></td>
</tr>
<tr>
<td>get</td>
<td>&lt;id&gt;</td>
<td></td>
<td>cid,name,tel,snum</td>
<td>(PGM:get, id', name', tel,snum')</td>
</tr>
</tbody>
</table>

OUTPUT FUNCTIONS:

\[ \text{cid}(T) = \]

\[
\begin{array}{c|c|c}
&T=\_&\phi_n \\
T \neq \_ & \text{PGM}(\tau(T)) = \text{set} & \text{in}_1'(\tau(T)) \\
\end{array}
\]
name(T) \equiv
\begin{align*}
&
T = \_ \\
& T \neq \_ \land \\
& -(PGM(r(T))=\text{set}) \\
& -(PGM(r(T))=\text{set})
\end{align*}
\begin{align*}
& \phi_0 \\
& \text{in}^2_r(r(T)) \\
& \text{name}(p(T))
\end{align*}

tel(T) \equiv
\begin{align*}
&
T = \_ \\
& T \neq \_ \land \\
& -(PGM(r(T))=\text{set}) \\
& -(PGM(r(T))=\text{set})
\end{align*}
\begin{align*}
& \phi_0 \\
& \text{in}^3_r(r(T)) \\
& \text{tel}(p(T))
\end{align*}

snump(T) \equiv
\begin{align*}
&
T = \_ \\
& T \neq \_ \land \\
& -(PGM(r(T))=\text{set}) \\
& -(PGM(r(T))=\text{set})
\end{align*}
\begin{align*}
& 0 \\
& \text{in}^4_r(r(T)) \\
& \text{tel}(p(T))
\end{align*}

<id>.-cid(T) \equiv \text{cid}(T_{<id>})
<id>.-name(T) \equiv \text{name}(T_{<id>})
<id>.-tel(T) \equiv \text{tel}(T_{<id>})
<id>.-snump(T) \equiv \text{snump}(T_{<id>})

One object of this type is a description of a customer that can be used for all transactions. One customer may have several reservations from different specific flights, and each reservation represents one or more seats in a specific flight.

A3.2 The Specification of the System Requirements

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>custid</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>flitno</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>ddate</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>bdate</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>bst</td>
<td>&lt;status&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gflight</td>
<td>&lt;GenericFLight&gt;</td>
</tr>
<tr>
<td>sflight</td>
<td>&lt;SpecificFLight&gt;</td>
</tr>
<tr>
<td>customer</td>
<td>&lt;Customer&gt;</td>
</tr>
<tr>
<td>cid</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>fno</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>d</td>
<td>&lt;date&gt;</td>
</tr>
</tbody>
</table>

150
Appendix 3 The Flight Reservation System

PGM <string>

ACCESS PROGRAMS

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>addGflight</td>
<td>gflight</td>
<td>(PGM:addGflight, gflight)</td>
<td></td>
</tr>
<tr>
<td>remGflight</td>
<td>gflight</td>
<td>(PGM:remGflight, gflight)</td>
<td></td>
</tr>
<tr>
<td>addSflight</td>
<td>sflight</td>
<td>(PGM:addSflight, sflight)</td>
<td></td>
</tr>
<tr>
<td>remSflight</td>
<td>sflight</td>
<td>(PGM:remSflight, sflight)</td>
<td></td>
</tr>
<tr>
<td>addCustomer</td>
<td>customer</td>
<td>(PGM:addCustomer, customer)</td>
<td></td>
</tr>
<tr>
<td>remCustomer</td>
<td>customer</td>
<td>(PGM:remCustomer, customer)</td>
<td></td>
</tr>
<tr>
<td>makResv</td>
<td></td>
<td>(PGM:makResv, cid, fno, dt', resv', bst')</td>
<td></td>
</tr>
<tr>
<td>canResv</td>
<td></td>
<td>(PGM:canResv, cid, fno, dt', resv', bst')</td>
<td></td>
</tr>
</tbody>
</table>

Auxiliary function definitions:

s(fno,d,T): (trace) → (event descriptor): returns the event descriptor describing the most recent changes on a SpecificFlight object with flight number “fno” and departure date “d”.

\[
s(fno,d,T) = T = \_ \\
T \neq \_ \land PGM(r(T)) = addSflight \land fno = sflight.fno'(r(T)) \land d = sflight.depdate'(r(T)) \land \\
\neg(PGM(r(T)) = addSflight \land fno = sflight.fno'(r(T)) \land d = sflight.depdate'(r(T)))
\]

n(cid, T): (trace) → (integer): returns the number of seats that a customer wants to reserve, this customer’s id is “cid”.

\[
n(cid, T) = T = \_ \\
T \neq \_ \land PGM(r(T)) = addCustomer \land cid = customer.cid'(r(T)) \land \\
\neg(PGM(r(T)) = addCustomer \land cid = customer.cid'(r(T)))
\]

OUTPUT FUNCTIONS:

custid(T) ==

\[
custid(T) = T = \_ \\
T \neq \_ \land PGM(r(T)) = makResv \lor PGM(r(T)) = canResv
\]

flitno(T) ==

\[
flitno(T) = T = \_ \\
T \neq \_ \land PGM(r(T)) = makResv \lor PGM(r(T)) = canResv
\]

ddate(T) ==

\[
ddate(T) = T = \_ \\
T \neq \_ 
\]
\[ T \neq \_ \land PGM(r(T)) = \text{makResv} \lor PGM(r(T)) = \text{canResv} \]
\[ \neg(PGM(r(T)) = \text{makResv} \lor PGM(r(T)) = \text{canResv}) \]
\[ d'(r(T)) \]
\[ ddate(p(T)) \]

\[ bdate(T) = \]
\[ T = \_ \]
\[ T \neq \_ \land PGM(r(T)) = \text{makResv} \]
\[ \neg(e = \phi_0) \]
\[ \neg(sflight.fst(e) \geq n(cid'(r(T)), T)) \]
\[ \neg(sflight.fst(e) < n(cid'(r(T)), T)) \]
\[ bdate(p(T)) \]

\[ * \text{toDate(t) is a function that converts the time in the event to the format of type <date>.} \]

\[ bst(T) = \]
\[ T = \_ \]
\[ T \neq \_ \land PGM(r(T)) = \text{makResv} \]
\[ e = \phi_0 \]
\[ \neg(sflight.fst(e)) \geq n(cid'(r(T)), T) \]
\[ \neg(sflight.fst(e)) < n(cid'(r(T)), T) \]
\[ bst(p(T)) \]

where \( e = s(fno'(r(T)), d'(r(T)), T) \) and

\[ n(cid'(r(T)), T) = \]
\[ T = \_ \]
\[ T \neq \_ \land PGM(r(T)) = \text{addCustomer} \land cid'(r(T)) = \text{customer}.cid'(r(T)) \]
\[ 
\neg(PGM(r(T)) = \text{addCustomer} \land cid'(r(T)) = \text{customer}.cid'(r(T)) ) 
\]
\[ n(cid'(r(T)), p(T)) \]

\[ 1. \text{AgentHiding} \]

This component is for agents to send requests for reserving/cancelling reservations, receiving responses and displaying the requested information.

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cid_1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>fno_1</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>ddate_1</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>bdate_1</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>bst_1</td>
<td>&lt;status&gt;</td>
</tr>
<tr>
<td>cp_1</td>
<td>&lt;string&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>i_11</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>i_12</td>
<td>&lt;num&gt;</td>
</tr>
</tbody>
</table>
Appendix 3 The Flight Reservation System

i₁₃ <date>
i₁₄ <status>
PGM₁ <string>

### ACCESS PROGRAMS

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>makResv</td>
<td>(PGM₁:makResv, i₁₁, i₁₂, i₁₃, i₁₄, rsv₁’, bst₁’)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>canResv</td>
<td>(PGM₁:canResv, i₁₁, i₁₂, i₁₃, i₁₄, rsv₁’, bst₁’)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### OUTPUT FUNCTIONS:

**cid₁(T) ≡**

\[
T = _ \quad \phi_n
\]

\[
T \neq_\wedge PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv}
\]

\[
\neg (PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv})
\]

\[
cid₁(p(T))
\]

**fno₁(T) ≡**

\[
T = _ \quad \phi_n
\]

\[
T \neq_\wedge PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv}
\]

\[
\neg (PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv})
\]

\[
fno₁(p(T))
\]

**ddate₁(T) ≡**

\[
T = _ \quad \phi_d
\]

\[
T \neq_\wedge PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv}
\]

\[
\neg (PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv})
\]

\[
\text{ddate₁}(p(T))
\]

**bdate₁(T) ≡**

\[
T = _ \quad \phi_d
\]

\[
T \neq_\wedge PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv}
\]

\[
\neg (PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv})
\]

\[
\text{addBooking}
\]

\[
\text{remBooking}
\]

**bst₁(T) ≡**

\[
T = _ \quad \text{empty}
\]

\[
T \neq_\wedge PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv}
\]

\[
\neg (PGM₁(r(T)) = \text{makResv} \lor PGM₁(r(T)) = \text{canResv})
\]

\[
bst₁(p(T))
\]

2. GFAM (Generic Flight information Access Management)
This component maintains an up-to-date time table for each generic flight, provides access programs for adding new object of GenericFlight and getting information about a certain generic flight by flight number.

This is a component for storing a set of objects of type GenericFlight and providing schedule information according to requests. Each GenericFlight can be uniquely identified by a flight number. Therefore a flight number is given to this component as input to get information about a certain generic flight.

A generic flight is one kind of airplane with the same flight number that flies regularly. We assume one generic flight is always one type of airplane. Therefore it has the same capacity, and always flies at the same time of day.

### OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>gft2</td>
<td>&lt;GenericFlight&gt;</td>
</tr>
</tbody>
</table>

### INPUT VARIABLES

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM21</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>PGM22</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>i21</td>
<td>&lt;GenericFlight&gt;</td>
</tr>
<tr>
<td>i22</td>
<td>&lt;num&gt;</td>
</tr>
</tbody>
</table>

### ACCESS PROGRAMS

<table>
<thead>
<tr>
<th>ProgramName</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>addGFlight</td>
<td>i21</td>
<td></td>
<td>(PGM21:addGFlight, 'i21, i21')</td>
</tr>
<tr>
<td>remGFlight</td>
<td>i21</td>
<td></td>
<td>(PGM21:remGFlight, 'i21, i21')</td>
</tr>
<tr>
<td>get</td>
<td>gft2</td>
<td>i22</td>
<td>(PGM22:get, i22, i21, gft2')</td>
</tr>
</tbody>
</table>

### Auxiliary function definition:

\[ g(fno, T) : (\langle \text{trace} \rangle \rightarrow \langle \text{event descriptor} \rangle) \text{ returns the event descriptor describing the most recent change on a GenericFlight object with flight number "fno".} \]

\[
\begin{align*}
g(fno, T) & = \\
T & = \_ \\
T & \neq _\land \ (PM_{21}(r(T))=\text{addGflight} \land fno=\text{gflight.fno'}(r(T)) \\
\neg (PM_{21}(r(T))=\text{addGflight} \land fno=\text{gflight.fno'}(r(T)) & \\
\phi_0 & \\
r(T) & \\
g(fno, p(T)) &
\end{align*}
\]

### OUTPUT FUNCTIONS:

\[ gft2(T) = \]

\[
\begin{align*}
T & = \_ \\
T & \neq _\land \ (PM_{22}(r(T)) \mid e=\phi_0 \\
\neg e=\phi_0 \land \neg (PM_{22}(r(T))=\text{get}) & \\
\phi_g & \\
\phi_g & \\
\phi_g & \\
l_{i21}(e) & \\
gft2(p(T)) &
\end{align*}
\]

Where \( e=\text{g}(i_{22}'(r(T)), T) \)
3. SFAM (Specific Flight information Access Management)

This component maintains an up to date picture of booking status of each specific flight, provides access programs for adding new object of SpecificFlight, updating the booking list of a specific flight status and getting current information of a specific flight by flight number and departure date.

It is a component for storing a set of objects of type SpecificFlight. Each SpecificFlight can be uniquely identified by a flight number and a departure date. A flight number and a departure date will be given to this component to get the information about a specific flight.

A specific flight is one airplane with a certain flight number that flies at a certain day. Each specific flight associates with a generic flight by flight number. Each generic flight can be associated with a class of specific flights with the same flight number.

### OUTPUT VARIABLES

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>res&lt;sub&gt;3&lt;/sub&gt;</td>
<td>&lt;status&gt;</td>
</tr>
<tr>
<td>cp&lt;sub&gt;31&lt;/sub&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>cp&lt;sub&gt;32&lt;/sub&gt;</td>
<td>&lt;string&gt;</td>
</tr>
</tbody>
</table>

### INPUT VARIABLES

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM&lt;sub&gt;31&lt;/sub&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>PGM&lt;sub&gt;32&lt;/sub&gt;</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>i&lt;sub&gt;31&lt;/sub&gt;</td>
<td>&lt;SpecificFlight&gt;</td>
</tr>
<tr>
<td>i&lt;sub&gt;32&lt;/sub&gt;</td>
<td>&lt;GenericFlight&gt;</td>
</tr>
<tr>
<td>i&lt;sub&gt;33&lt;/sub&gt;</td>
<td>&lt;num&gt;</td>
</tr>
<tr>
<td>i&lt;sub&gt;34&lt;/sub&gt;</td>
<td>&lt;date&gt;</td>
</tr>
<tr>
<td>i&lt;sub&gt;35&lt;/sub&gt;</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

### ACCESS PROGRAMS

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>addSFlight</td>
<td>i&lt;sub&gt;31&lt;/sub&gt;</td>
<td>PGM&lt;sub&gt;31&lt;/sub&gt;:addSFlight, ‘i&lt;sub&gt;31&lt;/sub&gt;, i&lt;sub&gt;31&lt;/sub&gt;’</td>
<td></td>
</tr>
<tr>
<td>remSFlight</td>
<td>i&lt;sub&gt;31&lt;/sub&gt;</td>
<td>PGM&lt;sub&gt;31&lt;/sub&gt;:remSFlight, ‘i&lt;sub&gt;31&lt;/sub&gt;, i&lt;sub&gt;31&lt;/sub&gt;’</td>
<td></td>
</tr>
<tr>
<td>addBooking</td>
<td>i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;, i&lt;sub&gt;35&lt;/sub&gt;</td>
<td>PGM&lt;sub&gt;32&lt;/sub&gt;:addBooking, ‘i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;, i&lt;sub&gt;35&lt;/sub&gt;, res&lt;sub&gt;3&lt;/sub&gt;’</td>
<td></td>
</tr>
<tr>
<td>remBooking</td>
<td>i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;, i&lt;sub&gt;35&lt;/sub&gt;</td>
<td>PGM&lt;sub&gt;32&lt;/sub&gt;:remBooking, ‘i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;, i&lt;sub&gt;35&lt;/sub&gt;, res&lt;sub&gt;3&lt;/sub&gt;’</td>
<td></td>
</tr>
<tr>
<td>availSeats</td>
<td>numAseats&lt;sub&gt;3&lt;/sub&gt;</td>
<td>i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;</td>
<td>PGM&lt;sub&gt;32&lt;/sub&gt;:availSeats, ‘i&lt;sub&gt;32&lt;/sub&gt;, i&lt;sub&gt;33&lt;/sub&gt;, i&lt;sub&gt;34&lt;/sub&gt;, numAseats&lt;sub&gt;3&lt;/sub&gt;’</td>
</tr>
</tbody>
</table>

### OUTPUT FUNCTIONS:

res<sub>3</sub>(T) =

<table>
<thead>
<tr>
<th>T= empty</th>
<th>res&lt;sub&gt;3&lt;/sub&gt;(T) = empty</th>
</tr>
</thead>
<tbody>
<tr>
<td>T≠ empty ∧ addBooking ( \land ) PGM&lt;sub&gt;32&lt;/sub&gt;(( \tau ))(T) ( \land ) e=( \phi_0 ) ( \rightarrow (e=\phi_0) ) sflight.fst(e) ( \geq ) i&lt;sub&gt;35&lt;/sub&gt;'(( \tau ))(T) ( \land ) sflight.fst(e) &lt; i&lt;sub&gt;35&lt;/sub&gt;'(( \tau ))(T)</td>
<td>resed</td>
</tr>
</tbody>
</table>
This component maintains an up to date set of customer information, provides access programs for adding new customer and getting information about a certain customer by customer id.

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>snum4</td>
<td>&lt;integer&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM41</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>PGM42</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>i41</td>
<td>&lt;Customer&gt;</td>
</tr>
<tr>
<td>i42</td>
<td>&lt;num&gt;</td>
</tr>
</tbody>
</table>

**ACCESS PROGRAMS**

<table>
<thead>
<tr>
<th>Program Name</th>
<th>Value</th>
<th>In</th>
<th>Abbreviated Event Descriptor</th>
</tr>
</thead>
<tbody>
<tr>
<td>addCustomer</td>
<td>i41</td>
<td></td>
<td>(PGM41:addCustomer, ‘i41, i41’)</td>
</tr>
<tr>
<td>remCustomer</td>
<td>i41</td>
<td></td>
<td>(PGM41:remCustomer, ‘i41, i41’)</td>
</tr>
<tr>
<td>getNumseats</td>
<td>snum4</td>
<td>i41,i42</td>
<td>(PGM42:getNumseats, ‘i41, ‘i42, snum4’)</td>
</tr>
</tbody>
</table>

**OUTPUT FUNCTIONS:**

\[snum4(T) = n(i42'(r(T)), T)\]
5. PGM splitter

This component diverts different values of the PGM variable to different components. The system PGM variable can be split into 4 output variables, each connecting to one component.

**OUTPUT VARIABLES**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>cp51</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>cp52</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>cp53</td>
<td>&lt;string&gt;</td>
</tr>
<tr>
<td>cp54</td>
<td>&lt;string&gt;</td>
</tr>
</tbody>
</table>

**INPUT VARIABLES**

<table>
<thead>
<tr>
<th>Field Name</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>PGM5</td>
<td>&lt;string&gt;</td>
</tr>
</tbody>
</table>

\[ cp_{51}(T) \equiv \]

\[ T = _\]

\[ T \neq _ \land \begin{align*}
    PGM_5(r(T)) &= \text{makResv} \\
    PGM_5(r(T)) &= \text{remResv} \\
    \neg (PGM_5(r(T)) &= \text{makResv} ) \\
    \lor (PGM_5(r(T)) &= \text{makResv} )
\end{align*} \]

\[ \phi_0 \]

\[ cp_{52}(T) \equiv \]

\[ T = _\]

\[ T \neq _ \land \begin{align*}
    PGM_5(r(T)) &= \text{addGFlight} \\
    PGM_5(r(T)) &= \text{remGFlight} \\
    \neg (PGM_5(r(T)) &= \text{addGFlight} ) \\
    \lor (PGM_5(r(T)) &= \text{addGFlight} )
\end{align*} \]

\[ \phi_0 \]

\[ cp_{53}(T) \equiv \]

\[ T = _\]

\[ T \neq _ \land \begin{align*}
    PGM_5(r(T)) &= \text{addSFlight} \\
    PGM_5(r(T)) &= \text{remSFlight} \\
    \neg (PGM_5(r(T)) &= \text{addSFlight} ) \\
    \lor (PGM_5(r(T)) &= \text{addSFlight} )
\end{align*} \]

\[ \phi_0 \]

\[ cp_{54}(T) \equiv \]

\[ T = _\]

\[ T \neq _ \land \begin{align*}
    PGM_5(r(T)) &= \text{addCustomer} \\
    PGM_5(r(T)) &= \text{remCustomer} \\
    \neg (PGM_5(r(T)) &= \text{addCustomer} ) \\
    \lor (PGM_5(r(T)) &= \text{addCustomer} )
\end{align*} \]

\[ \phi_0 \]
A3.4 The Derivation of the Network Behavioral Description

In this section we use the formula $f_{kd} = \pi_k \circ f_k = f_k(\pi(T))$ to extend all the component trace functions to network trace functions, simplify them and replace the internal variables in the final network output functions by their network trace functions.

\[ \pi_1 \circ cid_1 \equiv cid_{1d}(T) = cid_1(\pi(T)) = \phi_n \]

\[ \pi_1(T) \neq _\land \quad \text{PGM}_1(r(\pi(T))) = \text{makResv} \lor \text{PGM}_1(r(\pi(T))) = \text{canResv} \]

\[ \neg (\text{PGM}_1(r(\pi(T))) = \text{makResv} \lor \text{PGM}_1(r(\pi(T))) = \text{canResv}) \lor T \neq _\land \quad \text{cid}_{1d}(p(\pi(T))) \]

\[ \equiv \text{(Lemma 4.4)} \]

\[ \pi_1(T) \neq _\land \quad \text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv} \]

\[ \neg (\text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv}) \lor T \neq _\land \quad \text{cid}_{1d}(p(T)) \]

\[ \equiv (\text{PGM}_1 \leftarrow \gamma_8 \leftarrow \text{cp}_{51}, i_{11} \leftarrow \text{cid} \]

\[ \text{row 2) is from row2} \quad \text{3) is from row 1} \quad \text{and row 4) in definition of cp}_{51} \]

\[ \pi_1(T) = _\land \quad \text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv} \]

\[ \neg (\text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv}) \lor \phi_n \]

\[ \text{cid}'(r(T)) \]

\[ \text{fno}'(T) \]

\[ \equiv (\text{PGM}_1 \leftarrow \gamma_8 \leftarrow \text{cp}_{51}, i_{12} \leftarrow \text{fno} \]

\[ \text{ddate}_{1d}(T) \equiv \text{ddate}_{1d}(\pi(T)) \]

\[ \pi_1(T) = _\land \quad \text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv} \]

\[ \neg (\text{PGM}(r(T)) = \text{makResv} \lor \text{PGM}(r(T)) = \text{canResv}) \lor \phi_d \]

\[ i_{13} \equiv \text{fno}(\pi(T)) \]

\[ i_{14} \equiv \text{ddate}_{1d}(\pi(T)) \]
Appendix 3 The Flight Reservation System

\[(\neg(PGM_1(r(T))) = makResv \lor PGM_1(r(T)) = canResv) \land ddate_1(p(T))\]

\[\equiv (\neg(PGM_1(r(T))) = makResv \lor PGM_1(r(T)) = canResv) \land ddate_1(p(T))\]

\[\equiv (PGM_1 \leftarrow \gamma_8 \leftarrow cp_{31}, i_{13} \leftarrow d)\]

\[\equiv (PGM_1 \leftarrow \gamma_8 \leftarrow cp_{31})\]

\[\equiv (PGM_1 \leftarrow \gamma_8 \leftarrow cp_{31})\]

\[\equiv (PGM_1 \leftarrow \gamma_8 \leftarrow cp_{31})\]

\[\equiv (i_{14} \leftarrow \gamma_3 \leftarrow res_3)\]

\[\equiv (i_{14} \leftarrow \gamma_3 \leftarrow res_3)\]
\[
\begin{align*}
\text{cp}_{1d}(T) &\equiv \text{cp}_1(\pi_1(T)) \\
\pi_1(T) &\neq \_ \\
\pi_1(T) &\neq \_ \\
\text{PGM}_1(r(\pi_1(T))) &\equiv \text{makResv} \\
\text{PGM}_1(r(\pi_1(T))) &\equiv \text{canResv} \\
\text{addBooking} \\
\text{remBooking}
\end{align*}
\]

Where \( e' = g('i_{22}(r(\pi_2(T))), p(\pi_2(T))) \)
Appendix 3 The Flight Reservation System

\[ \pi_T(T) = \_ \]

\[ \pi_T(T) \neq \_ \land \neg \text{PGM}_{21}(\pi_T(T)) = \text{addGflight} \land 'i_{22}(\pi_T(T)) = \text{gflight.fno'}(\pi_T(T)) \]

\[ \phi_0 \]

\[ r(\pi_T(T)) \]

\[ \text{gflight}(\pi_T(T)) \]

\[ \text{p(} \pi_T(T) \text{)} \]

res_{3d}(T) \equiv res_3(\pi_T(T)) =

\[ \pi_T(T) = \_ \]

\[ \pi_T(T) \neq \_ \land \neg \text{PGM}_{32}(\pi_T(T)) = \text{addBooking} \land 'i_{22}(\pi_T(T)) = \text{sflight.fno'}(\pi_T(T)) \]

\[ \phi_0 \]

\[ r(\pi_T(T)) \]

\[ \text{gflight}(\pi_T(T)) \]

\[ \text{p(} \pi_T(T) \text{)} \]

\[ \text{res}_{3d}(T) \equiv \text{res}_3(\pi_T(T)) \equiv \pi_T(T) = \_ \equiv \pi_T(T) \neq \_ \land \neg \text{PGM}_{31}(\pi_T(T)) = \text{addSflight} \land 'i_{33}'(\pi_T(T)) = \text{sflight.fno'}(\pi_T(T)) \land 'i_{34}'(\pi_T(T)) = \text{sflight.depdate'}(\pi_T(T)) \]

\[ \phi_0 \]

\[ r(\pi_T(T)) \]

\[ \text{gflight}(\pi_T(T)) = \text{sflight.fno'}(\pi_T(T)) \]

\[ \text{sflight.depdate'}(\pi_T(T)) = \text{sflight.depdate'}(\pi_T(T)) \]

Where \( e' = s(i_{33}',(\pi_T(T))), i_{34}',(\pi_T(T)), \pi_T(T) = \_ \)

\[ \pi_T(T) = \_ \]

\[ \pi_T(T) \neq \_ \land \neg \text{PGM}_{31}(\pi_T(T)) = \text{addSflight} \land 'i_{33}'(\pi_T(T)) = sflight.fno'(\pi_T(T)) \land 'i_{34}'(\pi_T(T)) = sflight.depdate'(\pi_T(T)) \]

\[ \phi_0 \]

\[ r(\pi_T(T)) \]

\[ s(i_{33}',(\pi_T(T))), i_{34}',(\pi_T(T)), \text{p(} \pi_T(T) \text{)} \]

\[ \equiv i_{33}, i_{34}, \text{sflight and PGM}_3 \text{ and sflight change values only in C}_3 \Rightarrow \text{PGM}_{31}(\pi_T(T)) = \text{PGM}_3(\pi_T(T)), i_{33}'(\pi_T(T)) = i_{33}'(\pi_T(T)), \text{sflight.fno'}(\pi_T(T)) = \text{sflight.fno'}(\pi_T(T)), \text{sflight.depdate'}(\pi_T(T)) = \text{sflight.depdate'}(\pi_T(T)) \]
Appendix 3 The Flight Reservation System

\[
\pi_4(T) = _-
\]

\[
\pi_4(T) \neq _\Leftrightarrow \begin{array}{l}
PGM_{41}(r(T)) = \text{addCustomer} \\
i_{42}(r(T)) = \text{customer}.cid'(r(T)) \\
(PGM_{41}(r(T)) = \text{addCustomer} \\
i_{42}(r(T)) = \text{customer}.cid'(r(T))
\end{array}
\]

\[
\equiv (PGM_{41}, i_{41}, i_{42} \text{ change only in } C_4) \equiv 
\]

\[
\pi_4(T) = _-
\]

\[
\pi_4(T) \neq _\Leftrightarrow \begin{array}{l}
PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T)) \\
(PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T))
\end{array}
\]

\[
\equiv (PGM \leftarrow i_{41} \leftarrow \text{cp}_{54}, i_{41} \leftarrow \text{customer}, i_{42} \leftarrow i_{41} \leftarrow \text{cid}_{1}) 
\]

\[
\pi_4(T) = _-
\]

\[
\pi_4(T) \neq _\Leftrightarrow \begin{array}{l}
PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T)) \\
(PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T))
\end{array}
\]

\[
\equiv (\neg \text{ex}(PGM()) = \text{addCustomer} \text{ can be merged to row 3})
\]

\[
\pi_4(T) = _-
\]

\[
\pi_4(T) \neq _\Leftrightarrow \begin{array}{l}
PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T)) \\
(PGM(r(T)) = \text{addCustomer} \\
cid'(r(T)) = \text{customer}.cid'(r(T))
\end{array}
\]

\[
\equiv (\neg \text{ex}(PGM()) = \text{addCustomer} \text{ can be merged to row 3})
\]

\[
\pi_5(T) = _-
\]

\[
\pi_5(T) \neq _\Leftrightarrow \begin{array}{l}
PGM_5(r(T)) = \text{makResv} \\
\text{canResv}
\end{array}
\]

\[
\equiv \pi_5(T) = \Leftrightarrow T = _- \text{ (if none of these programs is called, there is no event in the network yet)}
\]

\[
r(\pi_5(T)) = r(T) \text{ (because all the other program calls are started from this component)}
\]
\[
\begin{array}{c}
\text{PGM}_5(\tau(T)) = \text{canResv} \\
\equiv (\text{PGM}_5 \leftarrow \text{PGM})
\end{array}
\]

\[
\begin{array}{c|cc}
T=\_ & \phi_0 \\
T\_\_ & \text{makResv} \\
& \text{canResv} \\
\land & \text{makResv} \\
\text{PGM}(\tau(T)) = \text{canResv} & \text{canResv} \\
\equiv & \phi_0
\end{array}
\]

Similarly:

\[
\begin{array}{c}
\text{cp52}_d(T) = \text{cp52}(\pi_5(T))
\end{array}
\]

\[
\begin{array}{c|cc}
\pi_5(T)=\_ & \phi_0 \\
\pi_5(T)\_ \land & \text{addGFlight} \\
& \text{canGFlight} \\
& \phi_0 \\
\equiv & \phi_0
\end{array}
\]

\[
\begin{array}{c}
\text{cp53}_d(T) = \text{cp53}(\pi_5(T))
\end{array}
\]

\[
\begin{array}{c|cc}
\pi_5(T)=\_ & \phi_0 \\
\pi_5(T)\_ \land & \text{addSFlight} \\
& \text{remSFlight} \\
& \phi_0 \\
\equiv & \phi_0
\end{array}
\]

\[
\begin{array}{c}
\text{cp54}_d(T) = \text{cp54}(\pi_5(T))
\end{array}
\]

\[
\begin{array}{c|cc}
\pi_5(T)=\_ & \phi_0 \\
\pi_5(T)\_ \land & \text{addCustomer} \\
& \text{remCustomer} \\
& \phi_0 \\
\equiv & \phi_0
\end{array}
\]
### A3.5 Convert the System Trace Functions to Network Trace Functions

To make the system functions comparable with the network functions, we need to convert all the system requirement tables to network functions tables by composing them with the mapping function $\pi_0$ and simplify them by comparing the event frequencies in the system and the network.

**$\pi_0$ custid = custid($\pi_0(T)$) =**

<table>
<thead>
<tr>
<th>$\pi_0(T)$</th>
<th>$\phi_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neq _n$</td>
<td>$\phi_n$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\neq _n$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\pi_0(T)$ = makResv $\lor$ PGM($\pi_0(T)$) = canResv</td>
</tr>
<tr>
<td></td>
<td>$\neg$((PGM($\pi_0(T)$) = makResv $\lor$ PGM($\pi_0(T)$) = canResv))</td>
</tr>
<tr>
<td></td>
<td>$\phi_n$</td>
</tr>
<tr>
<td></td>
<td>$\phi_n$</td>
</tr>
</tbody>
</table>

|= (The set of final access programs in the network is the same with those in the system. Therefore the outside observable event in the network is also observable in the system, and vice versa) |

**$\pi_0$ flitno = flitno($\pi_0(T)$) =**

<table>
<thead>
<tr>
<th>$\pi_0(T)$</th>
<th>$\phi_n$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neq _n$</td>
<td>$\phi_n$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\neq _n$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\pi_0(T)$ = makResv $\lor$ PGM($\pi_0(T)$) = canResv</td>
</tr>
<tr>
<td></td>
<td>$\neg$((PGM($\pi_0(T)$) = makResv $\lor$ PGM($\pi_0(T)$) = canResv))</td>
</tr>
<tr>
<td></td>
<td>$\phi_n$</td>
</tr>
<tr>
<td></td>
<td>$\phi_n$</td>
</tr>
</tbody>
</table>

**$\pi_0$ ddate = ddate($\pi_0(T)$) =**

<table>
<thead>
<tr>
<th>$\pi_0(T)$</th>
<th>$\phi_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neq _d$</td>
<td>$\phi_d$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\neq _d$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\pi_0(T)$ = makResv $\lor$ PGM($\pi_0(T)$) = canResv</td>
</tr>
<tr>
<td></td>
<td>$\neg$((PGM($\pi_0(T)$) = makResv $\lor$ PGM($\pi_0(T)$) = canResv))</td>
</tr>
<tr>
<td></td>
<td>$\phi_d$</td>
</tr>
<tr>
<td></td>
<td>$\phi_d$</td>
</tr>
</tbody>
</table>

**$\pi_0$ bdate = bdate($\pi_0(T)$) =**

<table>
<thead>
<tr>
<th>$\pi_0(T)$</th>
<th>$\phi_d$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\neq _d$</td>
<td>$\phi_d$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\neq _d$</td>
</tr>
<tr>
<td>$\pi_0(T)$</td>
<td>$\pi_0(T)$ = makResv $\lor$ PGM($\pi_0(T)$) = canResv</td>
</tr>
<tr>
<td></td>
<td>$\neg$((PGM($\pi_0(T)$) = makResv $\lor$ PGM($\pi_0(T)$) = canResv))</td>
</tr>
<tr>
<td></td>
<td>$\phi_d$</td>
</tr>
<tr>
<td></td>
<td>$\phi_d$</td>
</tr>
</tbody>
</table>
\[\pi_0(T) \neq _\land : \text{PGM}(r(T)) = \text{makResv} \quad \text{toDate}(t(r(T))) \quad \text{bddate}(p(T))\]

\[\equiv\]

\[T=\_] \quad \phi_0\]

\[T \neq _\land : \text{PGM}(r(T)) = \text{makResv} \quad \text{toDate}(t(r(T))) \quad \text{bddate}(p(T))\]

\[\pi_0 \circ \text{bst} \equiv \text{bst}(\pi_0(T)) \equiv\]

\[\begin{array}{|c|c|c|}
\hline
\pi_0(T) = \_ & \text{PGM}(r(T)) = \text{makResv} & e'' = \phi_0 \\
\wedge & \text{toDate}(t(r(T))) & \text{bddate}(p(T)) \\
\hline
\pi_0(T) = \_ & \text{PGM}(r(T)) = \text{canResv} & \text{reserved} \\
\wedge & \text{toDate}(t(r(T))) & \text{bddate}(p(T)) \\
\hline
\end{array}\]

Where \(e'' \equiv s(fno'(r(T))), d'(r(T)), \pi_0(T)\) and

\[e = s(fno'(r(T))), d'(r(T)), T) =\]

\[\begin{array}{|c|c|c|}
\hline
T=\_ & \text{PGM}(r(T)) = \text{addSflight} \wedge fno'(r(T)) = sflight.fno'(r(T)) \wedge \\
\wedge & d'(r(T)) = sflight.depdate'(r(T)) & s(fno'(r(T)), d'(r(T)), p(T)) \\
\hline
\end{array}\]

**A3.6 Conformance Checking**

1) \(\pi_1 \circ \text{cid}_1 \subseteq \pi_0 \circ \text{custid}\)

\[\begin{array}{|c|c|c|}
\hline
\pi_1(T) = \_ & \phi_n & \text{cid}'(r(T)) \\
\wedge & \text{cid}_1d(p(T)) & \text{cid}_1d(p(T)) \\
\hline
\end{array}\]

\[\begin{array}{|c|c|c|}
\hline
\pi_0 \circ \text{custid} \equiv \text{cid}_1d(\pi_0(T)) \equiv \\
\begin{array}{|c|c|}
\hline
\pi_1(T) = \_ & \phi_n \\
\wedge & \text{cid}'(r(T)) \\
\hline
\end{array}\]

\[\begin{array}{|c|c|c|}
\hline
\pi_0 \circ \text{custid} \equiv \text{cid}_1d(\pi_0(T)) \equiv \\
\begin{array}{|c|c|}
\hline
T_\_ & \phi_n \\
\wedge & \text{cid}'(r(T)) \\
\hline
\end{array}\]

166
In table $\pi_1 \circ cid_1$:

**Row 1**

$\pi_1(T) = \_ \leftrightarrow T = _\lor -ex(PGM()=makResv \lor PGM()=canResv)(T)$, therefore, $\pi_1(T) = _\land cid_1 = \phi$ is subsumed by row 1) and row 3) in the requirement table $\pi_0 \circ custid$

**Row 2** and 3 are subsumed by row 2) and 3) in the requirement table $\pi_0 \circ custid$, respectively.

From the above we may conclude that $\pi_1 \circ cid_1 \subseteq \pi_0 \circ custid$. The followed $\pi_1 \circ fno_1 \subseteq \pi_0 \circ flitno$, $\pi_1 \circ ddate_1 \subseteq \pi_0 \circ ddate$ and $\pi_1 \circ bdate_1 \subseteq \pi_0 \circ bdate$ can be proved similarly.

2) $\pi_1 \circ fno_1 \subseteq \pi_0 \circ flitno$

| $\pi_1(T) = \_ \_
\pi_1(T) \neq \_ | PGM(r(T))=makResv \lor PGM(r(T))=canResv |
\hline
$ \phi_n$
\hline
fno'(r(T))
\hline

3) $\pi_1 \circ ddate_1 \subseteq \pi_0 \circ ddate$

| $\pi_1(T) = \_ \_
\pi_1(T) \neq \_ | PGM(r(T))=makResv \lor PGM(r(T))=canResv |
\hline
$ \phi_d$
\hline
d'(r(T))
\hline
date(p(T))
\hline

4) $\pi_1 \circ bdate_1 \subseteq \pi_0 \circ bdate$

| $\pi_1(T) = \_ \_
\pi_1(T) \neq \_ | PGM(r(T))=makResv |
\hline
$ \phi_d$
\hline
toDate(t(r(\pi_1(T))))
\hline
\hline
5) $\pi_1 \circ bst_1 \subseteq \pi_0 \circ bst$
\[ \pi_1 \circ \text{bst}_1 \equiv \text{bst}_{id}(T) \equiv \text{bst}_{t}(\pi_1(T)) \equiv \]

| \( \pi_1(T) \) | \( \pi_2(T) \) | \( \text{PGM}(r(T)) \) | \( e' \equiv \phi_0 \) | \( \text{sflight.fst}(e') \geq n(\text{cid}'(r(T)),T) \) | \( \text{sflight.fst}(e') < n(\text{cid}'(r(T)),T) \) | \( \text{PGM}(r(T)) = \text{canResv} \) | \( \text{makResv} \) | \( \text{reserved} \) | \( \text{no enough seats} \) | \( \text{cancelled} \) | \( \text{bst}(p(T)) \) |
|--------------|--------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|----------------|
| \( \neq \_ \wedge \neq \_ \wedge \) | \( \pi_2(T) \) | \= \text{makResv} \) | \( \neg (e' = \phi_0) \) | \( \text{sflight.fst}(e') \geq n(\text{cid}'(r(T)),T) \) | \( \text{sflight.fst}(e') < n(\text{cid}'(r(T)),T) \) | \( \neg (\text{PGM}(r(T)) = \text{canResv}) \) | \( \text{PGM}(r(T)) = \text{canResv} \) | \( \text{cancelled} \) | \( \text{bst}(p(T)) \) |

\[ \pi_0 \circ \text{bst} \equiv \text{bst}(\pi_0(T)) \equiv \]

| \( T = \_ \) | \( \text{PGM}(r(T)) = \_ \) | \( \text{makResv} \) | \( e = \_ \) | \( \neg (e = \phi_0) \) | \( \text{sflight.fst}(e') \geq n(\text{cid}'(r(T)),T) \) | \( \text{sflight.fst}(e') < n(\text{cid}'(r(T)),T) \) | \( \neg (\text{PGM}(r(T)) = \text{canResv}) \) | \( \text{PGM}(r(T)) = \text{canResv} \) | \( \text{cancelled} \) | \( \text{bst}(p(T)) \) |

Where \( e' \equiv s(i_{13}'(r(\pi_1(T))), i_{14}'(r(\pi_1(T))), \pi_3(T)) \equiv \)

| \( \pi_3(T) \) | \( \pi_3(T) \neq \_ \wedge \pi_3(T) \neq \_ \wedge \) | \( \text{PGM}(r(T)) = \text{addSflight} \wedge fno'(r(T)) = \text{sflight.fno'}(r(T)) \wedge \) | \( d'(r(T)) = \text{sflight.depdate'}(r(T)) \) | \( \neg (\text{PGM}(r(T)) = \text{addSflight} \wedge fno'(r(T)) = \text{sflight.fno'}(r(T)) \wedge d'(r(T)) = \text{sflight.depdate'}(r(T)) ) \) |

| \( T = \_ \) | \( \text{PGM}(r(T)) = \_ \) | \( \text{addSflight} \wedge fno'(r(T)) = \text{sflight.fno'}(r(T)) \wedge \) | \( d'(r(T)) = \text{sflight.depdate'}(r(T)) \) | | | | | | |

| \( T 
eq \_ \wedge \) | \( \text{addSflight} \wedge fno'(r(T)) = \text{sflight.fno'}(r(T)) \wedge \) | \( d'(r(T)) = \text{sflight.depdate'}(r(T)) \) | | | | | | |

In the network function table \( \pi_1 \circ \text{bst}_1 \):

**Row 1:** \( \pi_1(T) = \_ \Leftrightarrow T = \_ \vee \neg \text{ex(PGM()) = makResv} \wedge \text{PGM()} = \text{canResv}(T) \)

“\( \pi_1(T) = \_ \wedge \text{bst}_1 = \text{empty} \)” is subsumed by row 1) and row 6) in the requirement table \( \pi_0 \circ \text{bst} \)

**Row 2:** From the definition of \( e' \) and \( e \) it can be seen \( e' = \phi_0 \Rightarrow e = \phi_0 \), because:

1) \( \pi_3(T) = \_ \Rightarrow T = \_ \vee \neg \text{ex(PGM()) = addSflight}(T) \)
2) \( \pi_3(T) \neq \_ \Rightarrow T = \_ \)
3) \( \pi_3(T) \neq \_ \wedge \pi_3(T) \neq \_ \Rightarrow T = \_ \)

From 1), 2) and 3),

**Row 2** “\( \pi_3(T) \neq \_ \wedge \pi_3(T) \neq \_ \wedge \text{PGM}(r(T)) = \text{makResv} \wedge e' = \phi_0 \wedge \text{bst}_1 = \text{empty} \)” in the network function table is subsumed by row 2) and 6) in the requirement table. (if \( a \Rightarrow c \) then \( a \wedge b \Rightarrow c \wedge b \))

**Row 3:** \( \neg (e' = \phi_0) \wedge \text{sflight.fst}(e') \geq \text{snum}_4'(r(T)) \Rightarrow \neg (e = \phi_0) \wedge \text{sflight.fst}(e') \geq n(\text{cid}'(r(T)),T) \)

because:

\( \text{sflight.fst} (\pi_1(T)) = \text{sflight.fst}(r(T)) \) (sflight changed only in SFAM)

\( \pi_3(T) \neq \_ \wedge (\text{PGM}(r(T)) = \text{addSflight} \wedge fno'(r(T)) = \text{sflight.fno'}(r(T)) \wedge d'(r(T)) = \text{sflight.depdate'}(r(T)) \)

\( \wedge \text{sflight.fst}(r(\pi_1(T))) \geq n(\text{cid}'(r(T)),T) \Rightarrow \)
T_≠_ ∧ (PGM(r(T))=addSflight ∧ fno’(r(T))=sflight.fno’(r(T)) ∧ d’(r(T))=sflight.depdate’(r(T))) ∧ sflight.fst(r(T)) ≥ n(cid’(r(T)), T)

Therefore row 3) in the network table is subsumed by row 3) in the requirement table.

**Row 4:** similar to the above, \(¬(e'=\phi_0) ∧ sflight.fst(e')<snum_4'(r(T)) \Rightarrow ¬(e=\phi_0) ∧ sflight.fst(e)<n(cid'(r(T)), T)\), so that row 4) in the network table is subsumed by row 4) in the requirement table.

**Row 5** \(\pi_1(T)_≠_ ∧ \pi_3(T)_≠_ ∧ PGM(r(T))=canResv ∧ bst_1 = cancelled\) is subsumed by row 5) in the requirement table \(\pi_0° bst\) because \(\pi_1(T)_≠_ ∧ \pi_3(T)_≠_ \Rightarrow T_≠_\)

(if a \Rightarrow c then a∧b\Rightarrow c∧b)

**Row 6** \(\pi_1(T)_≠_ ∧ ¬(PGM(r(T))=makResv ∨ PGM(r(T))=canResv) ∧ bst_1 = bst_{1d}(p(T))\) is subsumed by row 6) in the requirement table \(\pi_0° bst\) because \(\pi_1(T)_≠_ \Rightarrow T_≠_\)

From the above we can conclude that \(\pi_1° bst_1 \subseteq \pi_0° bst\).
Bibliography


• Reprinted with improvements in Yeh, R., *Current Trends in Programming Methodology* (I), Prentice Hall, 1977


(special issue on the 7th International Conference on Software Engineering).


edited by Gerald E. Peterson, IEEE Computer Society Press, IEEE Catalog Number

- Reprinted as Chapter 16 in item .


