A Graph-Transformation Modelling Framework for Supervisory Control

by

Jeremie Benhamron

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical & Computer Engineering

Waterloo, Ontario, Canada, 2016

© Jeremie Benhamron 2016

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Formal design methodologies have the potential to accelerate the development and increase the reliability of supervisory controllers designed within industry. One promising design framework which has been shown to do so is known as supervisory control synthesis (SCS).

In SCS, instead of manually designing the supervisory controller itself, one designs models of the uncontrolled system and its control requirements. These models are then provided as input to a special synthesis algorithm which uses them to automatically generate a model of the supervisory controller. This outputted model is guaranteed to be correct as long as the models of the uncontrolled system and its control requirements are valid. This accelerates development by removing the need to verify and rectify the model of the supervisory controller. Instead, only the models of the uncontrolled system and its requirements must be validated.

To address problems of scale, SCS can be applied in modular fashion, and implemented in hierarchical and decentralized architectures.

Despite the large body of research confirming the benefits of integrating SCS within the development process of supervisory controllers, it has still not yet found widespread application within industry. In the author's opinion, this is partly attributed to the non-user-friendly nature of the automaton-based modelling framework used create the models of the uncontrolled system (and control requirements in even-based SCS). It is believed that in order for SCS to become more accessible to a wider range of non experts, modelling within SCS must be made more intuitive and user-friendly.

To improve the usability of SCS, this work illustrates how a graph transformation-based modelling approach can be employed to generate the automaton models required for supervisory control synthesis. Furthermore, it is demonstrated how models of the specification can be intuitively represented within our proposed modelling framework for both event- and state-based supervisory control synthesis. Lastly, this thesis assesses the relative advantages brought about by the proposed graph transformation-based modelling framework over the conventional automaton based modelling approach.

Acknowledgements

First and foremost, I would like to thank my supervisor, Professor John G. Thistle for the guidance he has provided me with since the conception of this very interesting project. I have learnt a great deal working along his side. I would like to extend my thanks to Professor W. Murray Wonham for the invaluable knowledge I acquired in both his SCDES courses. Finally, I would like to express my gratitude to the readers of my thesis, Professors Stephen Smith, Krzysztof Czarnecki and John G. Thistle, for their essential feedback.

Table of Contents

Author's Declaration ii

Abstract iii

Acknowledgements iv

Table of Contents v

List of Tables viii

List of Figures ix

1 Introduction 1

1.1 Opening remarks 1

1.2 Supervisory control 3

1.3 Discrete event systems 5

1.4 Research objective 6

1.5 Contributions 6

1.6 Organization of document 7

2 Supervisory control theory (SCT) 8

2.1 The plant and its representations . . . 9

2.2 Modular design of plant . . . 12

2.3 Specification and admissible behaviour . . . 14

2.4 Supervisors & controlled systems . . . 17

2.5 Characterizing controlled systems . . . 18

2.6 Optimal nonblocking supervision . . . 19

2.7 Practical representation of supervisor . . . 20

2.8 Supervisor synthesis algorithm . . . 20

2.9 State-based supervisory control . . . 21

3 Graph rewriting 25

3.1 Preliminaries . . . 27

3.1.1 Graph theory . . . 27

3.1.2 Category theory . . . 29

3.1.3 Algebraic specification . . . 32

3.2 Algebraic approach . . . 36

3.2.1 Double pushout approach . . . 36

3.3 Graph transformation tools . . . 42

3.4 Extensions . . . 43

3.4.1 Typed graphs . . . 43

3.4.2 Typed attributed graphs . . . 44

3.4.3 Parametrized rules . . . 51

3.4.4 Graph constraints . . . 55

4 Graph rewriting modelling framework **57**

4.1 Implicit and induced automata 57

4.2 Generating deterministic finite-state automata 60

4.3 Modelling systems with implicit automata 61

4.3.1 Best modelling practices for attributed graphs 61

4.3.2 Modelling conventions 62

4.3.3 Creating implicit automata models 63

4.4 Application to supervisory control 65

4.4.1 Event-based synthesis 65

4.4.2 State-based synthesis 70

5 Discussion **74**

5.1 User-friendliness 74

5.2 Model intelligibility 75

5.3 Model modifiability 77

5.4 Susceptibility to error 79

5.5 Computational complexity 81

5.6 Industrial relevance 83

6 Conclusion **85**

6.1 Future work 86

Bibliography **87**

List of Tables

2.1 Examples of informal specifications 14

3.1 Examples of categories 30

List of Figures

1.1 System breakdown [27] 4

2.1 Online relationship between discrete event plant and supervisor 8

2.2 An example of a DFA 12

2.3 Constituent automata 14

2.4 Synchronous composition of constituent automata 15

2.5 Automaton representation of repair priority requirement 16

2.6 Mark bad states of H_a according to specification 22

2.7 Flag states leading to bad states via uncontrollable event 22

2.8 Flag states from which marked states cannot be reached via unflagged states 22

2.9 Remove bad states along with resultant loose edges 22

2.10 Remove inaccessible states along with resultant loose edges

. . . . 23

2.11 Resulting controlled behaviour 23

3.1 Algebraic approach to graph transformation 26

3.2 Directed graph with node and edge labels 27

3.3 A graph morphism from G to H 29

3.4 Commutative diagram for coproduct of a and b 30

3.5 Commutative diagram of pushout construction 31

3.6 Corresponding commutative diagram 32

3.7 Satisfiability criterion for atomic constraints 36

3.8 A graph transformation rule and its matching in G 37

3.9 Compact representation of `wp_from_M1_to_Buffer` 37

3.10 Double push out requirement for direct derivation 39

3.11 Construction of pushout of $D \leftarrow K \rightarrow R$ 40

3.12 GTS `Machine` equipped with inital graph `start` 41

3.13 Graph transition system induced by (`Machine`, `start`) 41

3.14 Compact representation of the graph transition system induced by (`Machine`, `start`) 42

3.15 Two TG-typed graphs $(t_G : G \rightarrow TG)$, $(t_H : H \rightarrow TG)$ and a TG-typed morphism ψ 44

3.16 An attributed type graph ATG and an ATG-typed graph 46

3.17 Two typed attributed graph transformation rules 49

3.18 Representation of composite terms within GTR 50

3.19 Source graph 50

3.20 Derivation involving rule with a product node 50

3.21 Parametrized graph transformation `repair` 51

3.22 A Parametrized direct derivation & the corresponding interpretation 52

3.23 A parametrized graph transformation system `Parametrized_Machines` and an initial graph `start` 53

3.24 A graph transition system induced by a parametrized graph transformation 54

3.25 A typed and attributed graph constraint 56

4.1 An implicit automaton 58

4.2 Generating DFA via implicit automata 61

4.3 Architecture of Big Factory 66

4.4 Implicit automaton of buffer specification 68

4.5 Induced automaton of buffer specification 68

4.6 Implicit automaton of buffer specification 69

4.7 Induced automaton of Repair priority specification 69

4.8 Topology of track 70

4.9 Implicit automaton of Guideway 71

4.10 Forbidden state constraint 72

4.11 Induced plant automaton of Guideway 73

4.12 State `s6` of the Guideway's induced automaton 73

5.1 Constituent automata of the Big Factory 76

5.2 Modular automaton model of Guideway 77

5.3 New topology of Guideway 78

5.4 New initial graph of Guideway 78

5.5 New marking constraint of Guideway 79

Chapter 1

Introduction

1.1 Opening remarks

When a company develops a system as a consumer product or for a specialized industrial application, it is generally necessary to develop new generations of the system over time. The company must do so to maintain its competitive edge in innovative business markets and keep up with rapidly evolving consumer demand. As companies must be able to meet short and frequent deadlines, they cannot afford to redesign systems from scratch. Instead, to decrease the time-to-market of next generation designs, new features must be integrated into previously existing system.

As such, a system's complexity incrementally increases with each new release. As a system matures, its increased complexity generally makes the modification and extension of its design more troublesome. Furthermore, as a system acquires more functionality, more regression testing must be performed in order to ensure that newly integrated functions do not interfere with or break previously existing functionalities. In order to mitigate such difficulties, engineers must develop systems in such a way that anticipated additions may be easily incorporated, implemented and tested.

One way to increase the speed and efficiency with which new generations of a system are developed is by adopting development methodologies which help streamline the different phases involved in updating the system. New formal methods providing benefits such as these are developed in the academic realm. One such formal method known as Supervisory Control Synthesis(SCS), facilitates the design of specific types of controllers referred to as supervisory controllers. In SCS, the engineer uses special synthesis algorithms to automatically generate supervisory controllers for a specific class of systems known as discrete event systems (DES). These synthesis algorithms are given models of the uncontrolled system behaviour (also known as

the plant) and a model of the control requirements (also known as the specification) and return as output a model of the supervisory controller.

As such, SCS eliminates the need to manually design the supervisory controller itself and allows the engineer to focus his efforts on the design and debugging of the plant and specification models. Moreover, the generated supervisory controller models are guaranteed to be correct given that the plant and specification models are valid. This makes for a faster and more reliable supervisory controller design procedure as compared with the manual design methodologies currently employed within industry. This is because in SCS, it removes the need to verify and correct the model of the supervisory controller; only the models of the uncontrolled system and its specification must be checked and validated.

SCS is based on an abstract mathematical framework known as Supervisory Control Theory (SCT). Although, SCS refers to the engineering design methodology which emerges from SCT, in practice these terms tend to be used interchangeably. For the sake of uniformity, in the sequel, we will use SCT to refer to both the engineering design methodology and underlying mathematical framework itself.

SCT, also known as the Ramadge-Wonham framework, was initiated by Ramadge and Wonham in the 1980s [23, 28]. Since its conception, the theory has gained significant popularity within the academic community. In addition, a number of software tools implementing the framework have been developed.

Despite commencing as an abstract mathematical theory, SCT has gradually matured into a viable engineering design methodology for discrete event supervisory controllers. However, although many aspects of the framework have become more practical, modelling has remained relatively unintuitive. This is due to the fact that the mathematical constructs used to model DES within SCT are finite state automata. Due to its simplicity, the finite state automaton is very practical from a theoretical and computational standpoint. However, for the exact same reason it is ill suited for modelling complex systems and requirements as will be shown later in this thesis.

In a synthesis-based controller development framework such as SCT, where synthesis algorithms are responsible for generating the controllers, modelling is one of the engineer's most important roles. An intuitive and user-friendly approach to modelling is therefore key in making SCT more accessible to non-experts.

Fortunately, alternative mathematical objects capable of modelling DES have been developed within academia. These include rewriting systems (RS) or reduction systems, which have long been studied within theoretical computer science. RS's refer to a general class of mathematical constructs which comprise a set of objects together with a binary relation that describes how the objects can be transformed. A well known type of rewriting system is the context free grammar

which has so called production rules that are used to specify how non-terminal symbols are transformed into a group of terminal and non-terminal symbols [17].

A graph rewriting system (GRS) or graph transformation system (GTS) is another type of RS. In a GRS, the manipulated objects are graphs and they are modified by so called graph transformation rules. Note that in this context the term graph refers to the mathematical object composed of nodes and edges, studied within the well known branch of mathematics called graph theory. Due to their graphical nature, graphs provide a very intuitive and visual means for representing the states of a system. When used for modelling purposes, GRS take advantage of this fact by representing the system's state with a graph and specify how it evolves via a set of graph transformation rules. Despite the seemingly informal flavour of this pictorial approach to modelling, GRS are based on rigorous and sound mathematical foundations [26]. GRS therefore provide an intuitive yet rigorous means for modelling the dynamics of a system.

In this thesis it is demonstrated how the use of GRS models for the discrete event plant and specification improve the usability of the SCT framework. Furthermore, it is shown how the adoption of this GRS modelling approach facilitates and expedites the process of updating prior plant and specification models, thereby increasing the framework's capacity to accelerate the integration of new system functionality within antecedent systems.

We would like to acknowledge the fact that using graph transformation systems to model systems is not a novel idea, and is not a contribution of this thesis. The main contribution of this thesis is demonstrating how a graph transformation modelling approach can be used to generate automaton models for use in the SCT framework (see section 1.5 for the specific contributions of this thesis).

The next section familiarizes the reader with the domain of supervisory control. Section 1.3 then introduces the the class of systems known as DES. In section 1.4 the general objective of this thesis will be stated. Section 1.5 explicitly states the specific contributions made by this work. Finally, section 1.6 provides an outline for the remainder of this work.

1.2 Supervisory control

The design of any automated system inevitably involves some form of control. A system operating under control generally comprises physical and control components.

As shown in figure 1.1, the physical components include sensors, actuators and the underlying structure which physically holds the system together. The sensors report physical quantities of interest associated to the system's environment or internal state. In a car, a sensor could measure engine temperature or tire pressure. The actuators are responsible for converting the control signals they receive into physical actions. They act directly on the system's underlying structure.

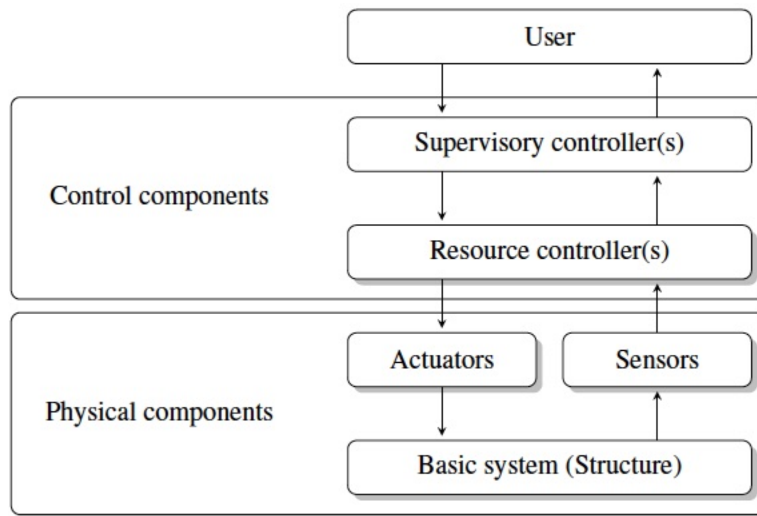


Figure 1.1: System breakdown [27]

The intake valve of a combustion engine is an example of an actuator. It closes and opens to regulate the engine intake of the air fuel mixture based on control signals. The control components, which constitute the brains of the system, must decide what instructions to send the actuators based on the information reported by the sensors. Depending on the application, the control components can be implemented as embedded systems, microprocessors or other information technologies.

The control solution employed to automate a complex CPS is generally organized into a hierarchy of control layers [3]. At the bottom of the control hierarchy one finds resource control also referred to as low level control or regulative control. This level of control is responsible for ensuring that simple processes are performed optimally. Low level control acts locally based on detailed information pertaining to a specific part of the system. It generally involves regulating and optimizing the behaviour of continuous-time processes. This is typically achieved via the implementation of analogue controllers.

Above the resource layer, one finds the supervisory control layer which may itself be subdivided hierarchically. Supervisory control, as the name suggests, is responsible for supervising and orchestrating overall system activity. In particular, it ensures that tasks are executed in the correct sequence. In doing so, it can prevent the system from coming to deadlock. Moreover, it can ensure safety by preventing the execution of dangerous sequences of actions. A supervisory controller (or supervisor) must interface with the resource layer in order to influence system behaviour. In contrast to resource control, supervisory control acts globally based on abstracted sensor information pertaining to various parts of the system.

Depending on the application, a supervisory controller may receive instruction from a user. The user may be a human or another automated system. In a hierarchical supervisory control architecture the user may even be another supervisor. The user typically operates the system by

issuing commands to the supervisor via some predefined user interface. Based on these general requests and the current state of the system, the supervisor issues an appropriate sequence of control commands to the resource controllers. If the user requests the execution of a certain task which should not be performed in the system's current state, the supervisor can postpone its execution until the system reaches a state in which the command can safely be executed. As such a supervisor can act as a buffer between the user and resource layer which prevents the user from mishandling the system.

To provide a concrete example of how a supervisor can fulfil this role, consider an automated airlock system of a spacecraft. When an astronaut attempts to enter the spacecraft, he presses a button which issues an "open outer door" request to the supervisory controller of the airlock system. In order to maintain a safe cabin pressure, the supervisor forbids the servicing of the "open outer door" request until the inner door is fully shut.

1.3 Discrete event systems

In control literature, the totality of a system's components is referred to as the *uncontrolled system* or *plant*. The plant, together with the control components form the *controlled system*. The plant *dynamics* or *uncontrolled system behaviour* is a mathematical model describing how the components of the system behave in the absence of control. Note that the term plant is sometimes used to refer to the dynamics of the plant itself. The *controlled system behaviour*, is a subset of the uncontrolled behaviour. It includes only those interactions possible in the presence of control. The system's *control requirements* or *specification* is a stipulation of what behaviours should and should not be present within the controlled behaviour.

When designing any controller, it is necessary to first understand and define the system's uncontrolled behaviour and control requirements. Only once the latter two milestones are achieved, is it possible to design a controller that yields a controlled behaviour compliant with the specifications. To characterize the plant, a model describing the system's uncontrolled behaviour is created. However, the controller only requires an abstracted system model to base its control decisions on. In other words, the controller need only be aware of the system details relevant to its purpose. As a result, the level and type of control under consideration, determines the appropriate degree of abstraction needed to create succinct plant models.

As supervisors are only responsible for ensuring tasks are executed in admissible sequences, they need only be aware of the order events occur in. At the supervisory control level, the plant is therefore most appropriately viewed as a discrete event system (DES) [5]. A DES is a discrete state system whose state transitions are caused by the occurrence of events which are both instantaneous and asynchronous in nature. In such systems, the initiation and termination of

tasks are modelled as discrete events. It is important to note that in a DES there is no explicit notion of time. Instead the passing of time is implied via the occurrence of events.

There exist several different approaches for modelling DES. In SCT, finite state automata are traditionally used to represent the discrete event models of the plant and specification.

1.4 Research objective

The overall objective of this research is to improve the user-friendliness of modelling within the SCT framework so that it may become more accessible to non-experts. This work attempts to accomplish this by introducing graph transformation as an intuitive modelling front end for SCT. The goal of this being, to dissociate the user from the current unintuitive automaton-based modelling framework. It is hoped that this work can ultimately lead to the incorporation of graph transformation tools (such as GROOVE) within the SCS software tool chain; or better yet to the integration of graph transformation modelling capabilities within SCS software tools themselves.

1.5 Contributions

The specific contributions of this work are as follows:

- We propose using graph transformation as an intuitive modelling front end for the SCT framework.
- We introduce a new graph-based front end model referred to as the implicit automaton.
- We illustrate how the implicit automaton can be used to generate the automaton models needed to perform supervisory control synthesis.
- We illustrate how parametrized, typed and attributed graph transformation rules can be used to concisely model the dynamics of discrete event systems. Furthermore, we show how the event set of the generated automaton can be extrapolated from these compact rule sets using so called interpretation of rules.
- In particular we illustrate how an implicit automaton can be used to model the plant in both event- and state-based SCT.
- We show how implicit automata can be used to express specifications within event-based SCT and how graph constraints can be used to specify mutual state exclusion specifications within state-based SCT.

- We illustrate the advantages and disadvantages of our proposed modelling approach over the conventional automaton-based modelling approach used within SCT.
- We describe how this modelling approach can be used in conjunction with the modular modelling techniques used within the SCT in order to alleviate the complexity demands which burden this new modelling approach.

1.6 Organization of document

The remainder of this thesis is organized in the following manner. In Chapter 2, we succinctly review the core concepts of supervisory control theory. In doing so we illustrate how modelling is conventionally performed in SCT. In chapter 3, we begin by reviewing some preliminary concepts used throughout the remainder of the thesis. We then present a particular approach to graph rewriting known as the algebraic approach. This chapter is concluded with the presentation of select graph rewriting extensions which are key to our proposed modelling approach. In chapter 4, the heart of this work, we present the proposed graph rewriting modelling framework. In particular, we illustrate how it is applied in event- and state-based supervisory control. Chapter 5 then discusses both the advantages and disadvantages of the proposed modelling approach over the conventional automaton-based modelling framework. Lastly, chapter 6 summarizes and concludes this thesis.

Chapter 2

Supervisory control theory (SCT)

This chapter summarizes the formal concepts and results underlying the supervisory control framework and illustrates its practical use. The original theory for supervisory control of discrete event systems was initiated by Peter J. Ramadge and W. Murray Wonham in the 1980s [28, 23] and has since been extended and optimized in a number of different ways. We refer the interested reader to [5, 29] for a comprehensive presentation of the theory and its various extensions.

SCT studies the problem of modifying the behaviour of a discrete event plant via feedback control in order to satisfy a set of specifications. In SCT, both the plant and supervisor are modelled as DESs which communicate with one another via discrete signals. The plant continuously reports to the supervisor the latest event that has occurred. Depending on the history of prior events, the controller may issue a disablement command which forbids the plant from executing some particular set of events. The closed-loop or controlled system formed by the dynamic interaction between the plant and supervisor is depicted in figure 2.1.

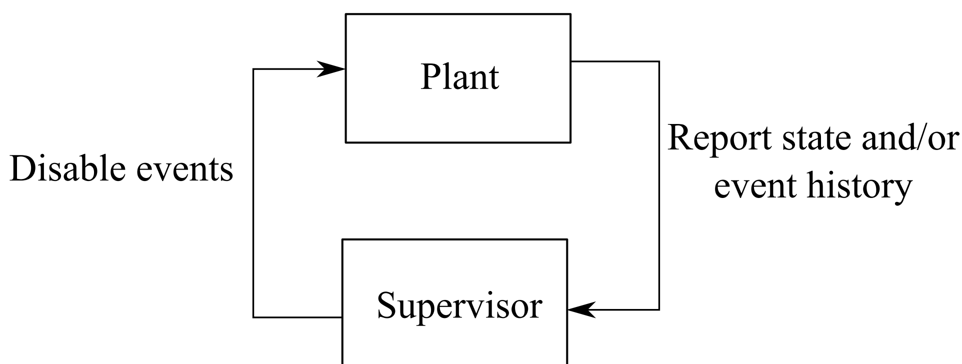


Figure 2.1: Online relationship between discrete event plant and supervisor

Note that, although supervisors may prohibit the occurrence of events, they are unable to coerce the execution of enabled events. A consequence of the supervisor's inability to force events is the requirement that the plant be independently capable of *initiating* or *generating* events. This assumption is often satisfied by the virtue of a human user operating the system.

As stated earlier, a supervisor is designed to yield a controlled system which meets a given set of specifications. SCT considers both safety and liveness. Safety specifications dictate what behaviours are prohibited by specifying what regions of the state space should be avoided or which sequences of events should be prevented. On the other hand, liveness refers to the minimum level of functionality the plant must retain under supervision.

Given a plant and a set of safety specifications, the goal is to create a supervisor which allows the plant to retain a maximum level of functionality without violating any of the safety specifications. Such a supervisor is referred to as maximally permissive or minimally restrictive with respect to the given specification.

The design of the supervisor must also take into account the fact that certain events cannot be disabled. The set of events the plant can generate must therefore be appropriately partitioned as controllable and uncontrollable events based on the modeler's intuitive understanding of the events. For instance, events representing the triggering of sensors are typically modelled as uncontrollable because a supervisor cannot prevent the system's environment from triggering a sensor. On the other hand, events involving actuation are viewed as controllable.

2.1 The plant and its representations

The plant (i.e. the uncontrolled system behaviour) can be given in terms of a formal language. In formal language theory an *alphabet* Σ is defined as a finite set of distinct symbols $\{\alpha, \beta, \ldots, \sigma\}$. A (non-empty) *string* or *word* over alphabet Σ is a finite sequence of symbols $\sigma_1\sigma_2\ldots\sigma_k$, where $k \geq 1$. The set of non-empty strings over alphabet Σ is denoted Σ^+. The symbol $\epsilon \notin \Sigma$ is used to denote the empty sequence of symbols and is called the *empty string*. The set of strings over Σ, is then denoted by Σ^* and defined as $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$. A language L over Σ is then defined as any subset of Σ^*. Therefore, the empty set $\emptyset$ and Σ^* itself are both languages over Σ.

By interpreting symbols as *events* and strings as sequences of events, a language can be used to represent a set of sequences of events. As such, the behaviour of a plant is represented by a pair of *regular languages* (L, L_m) where $L, L_m \subseteq \Sigma^*$(see [17] for a definition of regular languages). In this context, Σ^* is referred to as the *event set* or *alphabet* and specifies the set of events that the plant can generate. The language L, referred to as the plant's *closed behaviour*, represents the set of sequences of events the plant can generate. The plant's *marked behaviour*, L_m, is used to indicate

a set of sequences in L which represent the successful completion of important tasks. As such, we have $L_m \subseteq L$.

The *prefix closure of a language* L, denoted by $\overline{L}$, is the language consisting of all prefixes of strings contained in L. A language L is *closed* if $L = \overline{L}$, i.e. it contains all the prefixes of all of its strings. Any plant's closed behaviour is necessarily a closed language, as the possibility of the occurrence of a string (i.e. sequence of events) requires possibility of the occurrence of all of its prefixes.

The language representation of the plant is advantageous from a theoretical perspective as it facilitates the proof of theorems and allows for concise definitions. However, regular languages may have an infinite number of strings and therefore cannot be explicitly represented as such. *Finite-state automata* are finite mathematical models that can be used to represent regular languages. Formally, a finite state automaton is a 5-tuple

$$A = (Q, \Sigma, \Delta, Q_0, Q_m)$$

where

Q is a finite set of *discrete states*,
Σ is a finite set of *event labels*,
$\Delta \subseteq Q \times \Sigma \times Q$ is the *transition relation*,
$Q_0 \subseteq Q$ is the set of *initial states*, and
$Q_m \subseteq Q$ is the set of *marked states.*

To define the languages generated and marked by an automaton, the *extended transition relation* $\hat{\Delta} \subseteq Q \times \Sigma^* \times Q$ is inductively defined according to

(i) $(\forall q \in Q)\ (q, \epsilon, q) \in \hat{\Delta}$

(ii) $(\forall q, q'' \in Q)(\forall s \in \Sigma^*)(\forall \sigma \in \Sigma)\ (q, s\sigma, q'') \in \hat{\Delta}$ iff $(\exists q' \in Q)\ (q, s, q') \in \hat{\Delta}\ \&\ (q', \sigma, q'') \in \Delta$.

$L(A)$, the *language generated* by the automaton $A = (Q, \Sigma, \Delta, Q_0, Q_m)$ is then defined as

$$L(A) = \{s \in \Sigma^* | (\exists q_0 \in Q_0)(\exists q \in Q)\ (q_0, s, q) \in \hat{\Delta}\}$$

and $L_m(A)$ the *language marked* by A is given by

$$L_m(A) = \{s \in L(A) | (\exists q_0 \in Q_0)(\exists q_m \in Q_m)\ (q_0, s, q_m) \in \hat{\Delta}\}$$

Accordingly, a plant with closed and marked behaviour (L, L_m) can be represented by an automaton A whose generated and marked languages are equal to the closed and marked behaviours of the plant, that is $L = L(A)$ and $L_m = L_m(A)$. Intuitively, the automaton can be thought of as computational model which keeps track of the state of the plant as events occur. It begins in an initial state $q_0 \in Q_0$, prior to occurrence of any event. Upon the occurrence of some event $\sigma \in \Sigma$, the automaton's state is updated according to the transition relation Δ. I.e., if the automaton is in state q while σ occurs and $(q, \sigma, q') \in \Delta$, then the automaton transitions to state q'. Events are assumed to occur asynchronously and instantaneously. Furthermore, it is assumed that events occur non-deterministically, meaning they cannot be predicted. When the automaton arrives at a marked state $q_m \in Q_m$ it signifies the completion of an important task.

Intuitively, when the plant is in a specific state q and an event σ occurs, the plant should transition to a unique state. This condition on the transition relation can be formally stated as follows

$$(\forall q \in Q)(\forall \sigma \in \Sigma)[(\exists q' \in Q)(q, \sigma, q') \in \Delta \implies (\exists! q' \in Q)(q, \sigma, q') \in \Delta]$$

where $\exists!$ is the uniqueness quantifier defined as

$$\exists! x P(x) \equiv \exists x\, (P(x) \wedge \neg \exists y (P(y) \wedge y \neq x))$$

When the transition relation Δ satisfies this condition, it is a (partial) function. In this case, we speak of a *transition function* $\delta : Q \times \Sigma \to Q$ and write $\delta(q, \sigma) = q'$ in place of $(q, \sigma, q') \in \Delta$. Likewise, in place of an extended transition relation $\hat{\Delta}$, we speak of an *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to Q$ and write $\hat{\delta}(q, s) = q'$ in place of $(q, s, q') \in \hat{\Delta}$. In the latter case of the extended transition function, we omit the accent on top of the extended transition function and write δ in place of $\hat{\delta}$. In addition, we write $\delta(q, \sigma)!$ to indicate the transition function is defined over an input (q, σ). This is equivalent to the statement $(\exists q' \in Q)\ (q, \sigma, q') \in \Delta$.

A *deterministic finite state automaton* (DFA) is a finite state automaton that has a single initial state and a (partial) transition function instead of a relation. Formally a DFA is a 5-tuple $A = (Q, \Sigma, \delta, q_0, Q_m)$ where q_0 is now just a single element of Q. In standard supervisory control theory, the plant is usually represented by a DFA $G = (Q, \Sigma, \delta, q_0, Q_m)$. The plant is sometimes referred to as a generator, hence the use of the letter G.

Automata can be represented by graphs by using vertices to represent its states and edges to represent its transitions. Event labels are associated to each edge to indicate what event occurs

when a state transition takes place. Initial states are indicated by dangling incoming edges and marked states are drawn as filled-in vertices. To differentiate controllable from uncontrollable transitions, they are respectively drawn with solid and dashed edges. Figure 2.2 illustrates the automaton model of a basic machine.

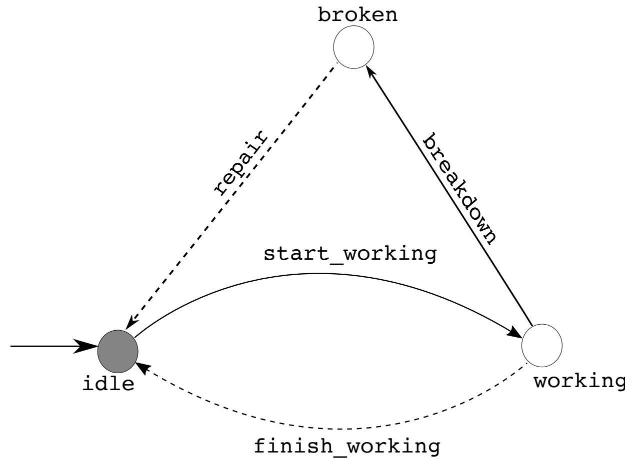


Figure 2.2: An example of a DFA

In this DFA we have

$$
\begin{aligned}
Q &= \{\texttt{working}, \texttt{idle}, \texttt{broken}\}, \\
\Sigma &= \{\texttt{start_working}, \texttt{finish_working}, \texttt{breakdown}, \texttt{repair}\}, \\
q_0 &= \texttt{idle}, \\
Q_m &= \{\texttt{idle}\}.
\end{aligned}
$$

2.2 Modular design of plant

Hypothetically, the plant may be produced by manually specifying the explicit state, transition and marking structure of its automaton model. However, this approach is only feasible for sufficiently simple plants. The number of states (and transitions) needed to model a system tends to grow exponentially as its complexity increases. This is a well known issue referred to as the *state explosion problem*. To circumvent this problem, one can create several automata which represent the system's components and obtain the complete plant model by computing the *synchronous composition* of these *constituent automata*.

The *synchronous composition* (or *parallel composition*) of two DFA $G_1 = (Q_1, \Sigma_1, \delta_1, q_{0_1}, Q_{m1})$ and $G_2 = (Q_2, \Sigma_2, \delta_2, q_{0_2}, Q_{m2})$, denoted by $G_1||G_2$, yields the automaton $(Q, \Sigma, \delta, q_0, Q_m)$ which is

defined by

$$
\begin{aligned}
Q &= Q_1 \times Q_2, \\
\Sigma &= \Sigma_1 \cup \Sigma_2, \\
q_0 &= (q_{01}, q_{02}), \\
Q_m &= Q_{m1} \times Q_{m2},
\end{aligned}
$$

and $(\forall (q_1, q_2) \in Q)(\forall \sigma \in \Sigma)$

$$
\delta((q_1, q_2), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \delta_2(q_2, \sigma)) & \text{if } \sigma \in \Sigma_1 \cap \Sigma_2, \delta_1(q_1, \sigma)! \text{ and } \delta_2(q_2, \sigma)! \\ (\delta_1(q_1, \sigma), q_2) & \text{if } \sigma \in \Sigma_1 - \Sigma_2 \text{ and } \delta_1(q_1, \sigma)! \\ (q_1, \delta_2(q_2, \sigma)) & \text{if } \sigma \in \Sigma_2 - \Sigma_1 \text{ and } \delta_2(q_2, \sigma)! \\ \text{undefined} & \text{otherwise} \end{cases}
$$

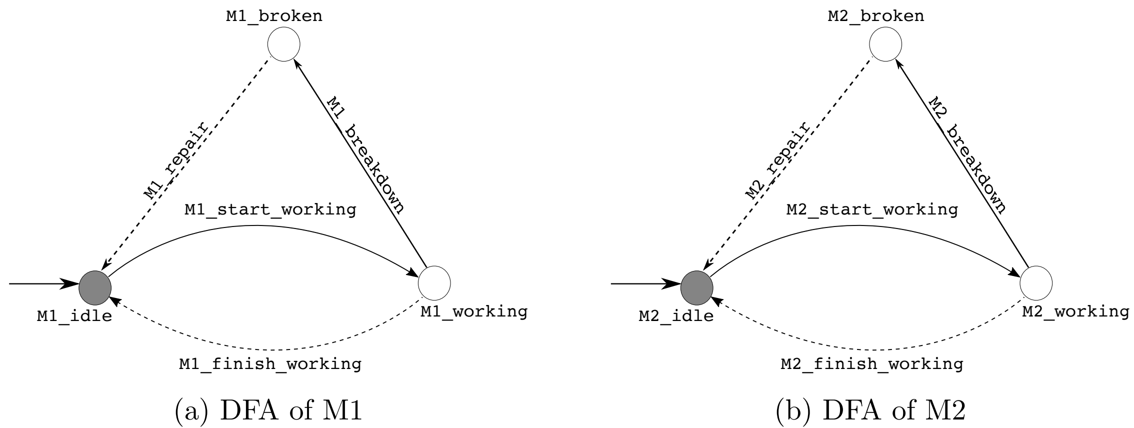
The synchronous composition of two subsystems G_1 and G_2 represents the joint operation of both subsystems in which the execution of shared events $\sigma \in \Sigma_1 \cap \Sigma_2$ must occur simultaneously and the execution of private events $\sigma \in \Sigma_1 \cup \Sigma_2 - \Sigma_1 \cap \Sigma_2$ occur independently. When $\Sigma_1 \cap \Sigma_2 = \emptyset$, the subsystems G_1 and G_2 are completely uncoupled and $G_1 || G_2$ models the concurrent yet independent operation of G_1 and G_2. This is sometimes referred to as *shuffle* or *interleaving composition* of G_1 and G_2.

In this definition of the synchronous composition, the state set is defined to be $Q = Q_1 \times Q_2$. In general however, only a subset of states $q \in Q$ are accessible from the initial state. These unreachable states are superfluous as they may be discarded without affecting the marked and generated languages of $G_1 || G_2$. Therefore in practice, only the *accessible component* of $G_1 || G_2$, is computed. The accessible component of an automaton $G = (Q, \Sigma, \delta, q_0, Q_m)$, is denoted by $Ac(G) = (Q_{ac}, \Sigma, \delta_{ac}, q_0, Q_{acm})$ and defined by

$$
\begin{aligned}
Q_{ac} &= \{q \in Q | (\exists s \in \Sigma^*) \ \delta(q_0, s) = q\}, \\
Q_{acm} &= Q_{ac} \cap Q_m, \\
\delta_{ac} &= \delta|_{Q_{ac} \times \Sigma},
\end{aligned}
$$

where $\delta|_{Q_{ac} \times \Sigma}$ is the restriction of δ to $Q_{ac} \times \Sigma$.

To illustrate these concepts, suppose one would like to model a system consisting of two machines which operate independently (when unsupervised) and whose individual behaviours are given by the automata depicted in figure 2.3.



(a) DFA of M1

(b) DFA of M2

Figure 2.3: Constituent automata

As the machines are assumed to function independently, their automaton models do not have any events in common. The synchronous composition of these two automata is then computed to obtain the complete plant model depicted in figure 2.4.

Recall, that the state set of an automaton resulting from a synchronous composition is given by $Q = Q_1 \times Q_2$, where Q_1 and Q_2 are the state sets of the constituent automata. For instance, in the example above $q_0 = (\texttt{M1_idle}, \texttt{M2_idle})$. Note that in general $Q_{ac} \subseteq Q$. However, in this case, all nine states of the synchronous composition are accessible - that is: $Q_{ac} = Q$. This is always the case when the event sets of the constituent automata are disjoint.

2.3 Specification and admissible behaviour

A system specification is a list of behavioural requirements a controlled system must satisfy. System specifications are initially specified in terms of natural language. Two examples of safety specifications for the two-machine system modelled by the plant automaton illustrated in figure 2.4 are provided in table 2.1. Note that as an automaton model only specifies the dynamics of a system, the underlying structure of the system is ambiguous. Therefore, the same automaton model can be associated to structurally different systems possessing the same overall dynamics. In this example, we assume that automaton depicted in figure 2.4 models a system comprising of two machines which are part of the same work cell.

i) If both machines break down, machine 2 must be repaired before machine 1.

ii) Only one machine can work at a time.

Table 2.1: Examples of informal specifications

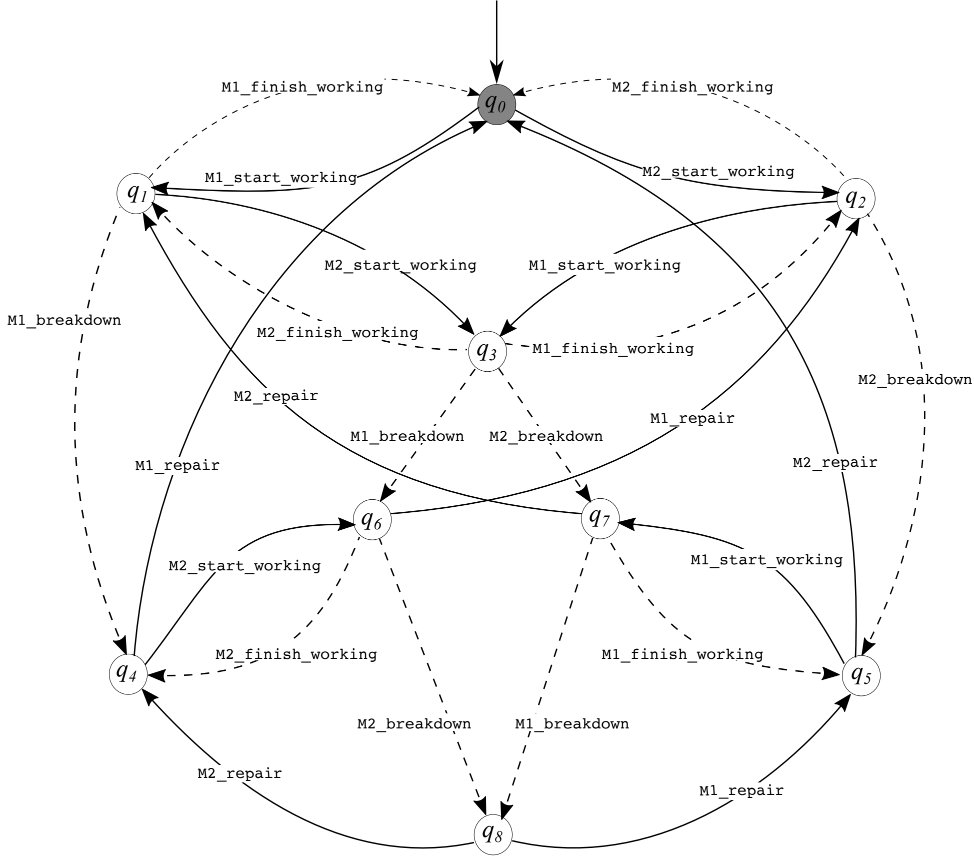


Figure 2.4: Synchronous composition of constituent automata

A hypothetical motivation for the second specification for instance is that the machines will interfere with one another if they operate at the same time.

The latter two statements are safety requirements because they indirectly stipulate how the controlled system should not act. Statement i) indicates that machine 1 should not be repaired before machine 2 and statement ii) indicates that both machines should not be working at the same time.

In event-based SCT a system's safety specifications are represented by a regular language $L_{spec} \subseteq \Sigma^*$ where Σ is the event set of the related plant. The language L_{spec} contains all the sequences of events in Σ^* which do not violate the specifications it is meant to represent. Any language representing a specification must be closed. Intuitively, this is because the prefix of a legal sequence of events is necessarily itself a legal sequence of events. As with the plant, a specification, is in practice represented using a DFA H_{spec} such that $L(H_{spec}) = L_{spec}$. For instance, requirement i) given in table 2.1, is embodied by the DFA illustrated in figure 2.5. This automaton permits all sequences except for those in which the event `M1_repair` occurs when machine 2 is in a broken state.

The DFA which representing requirement ii) of table 2.1 can be obtained by removing state (`M1_working`, `M2_working`) along with all of its incoming and outgoing transitions from the

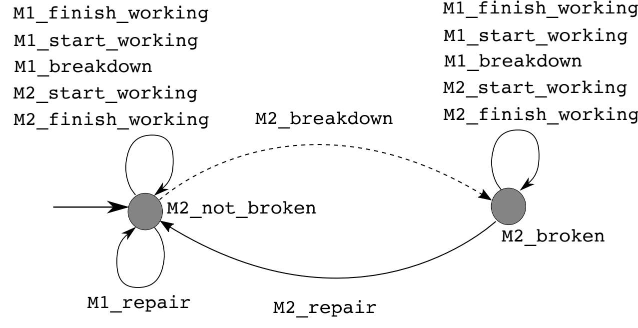


Figure 2.5: Automaton representation of repair priority requirement

synchronous composition of the two machines displayed in figure 2.4.

As system specifications typically consists of several behavioural requirements, the resulting specification automaton, H_{spec}, tends to be quite complex. For this reason creating H_{spec} manually is generally prohibitively difficult. As a result, modular approaches must be employed to construct H_{spec}. One possibility is to construct separate sub-specification DFA's, $H_{spec,1}, H_{spec,2}, \ldots, H_{spec,n}$, for each requirement in the specification and obtain H_{spec} by computing the *product* of these sub-specification automata.

The *product* (or *fully synchronous composition*) of a DFA $G_1, G_2, \ldots, G_n$ with $G_i = (Q_i, \Sigma_i, \delta_i, q_{0i}, Q_{mi})$, is denoted by $G_1 \times \ldots \times G_n$ and defined as the automaton

$$G_1 \times \ldots \times G_n = Ac\left(Q_1 \times \ldots \times Q_n,\ \bigcup_{i=1}^{n} \Sigma_i,\ \delta,\ (q_{01}, \ldots, q_{0n}),\ Q_{m1} \times \ldots \times Q_{mn}\right)$$

where $\delta : (Q_1 \times \ldots \times Q_n) \times \bigcup_{i=1}^{n} \Sigma_i \to (Q_1 \times \ldots \times Q_n)$ is defined by

$$\delta((q_1, \ldots, q_n), \sigma) = \begin{cases} (\delta_1(q_1, \sigma), \ldots, \delta_n(q_n, \sigma)) & \text{if } \sigma \in \bigcap_{i=1}^{n} \Sigma_i \;\; \& \;\; \delta_i(q_i, \sigma)! \;\; \text{for} \;\; i = 1, \ldots, n \\ \text{undefined} & \text{otherwise} \end{cases}$$

The product $G_1 \times \ldots \times G_n$ models the fully synchronized operation of $G_1, \ldots, G_n$ in the sense that it allows only the simultaneous execution of shared events. Logically, $G_1 \times \ldots \times G_n$ then executes exactly those sequences of events which all $G_1, \ldots, G_n$ may execute. In fact, it can be be shown that

$$L(G_1 \times \ldots \times G_n) = \bigcap_{i=1}^{n} L(G_i), \text{ and}$$
$$L_m(G_1 \times \ldots \times G_n) = \bigcap_{i=1}^{n} L_m(G_i).$$

As each language $L(H_{spec,i})$ represents the set of legal sequences with respect to a single system requirement, $L(H_{spec}) = L(H_{spec,1} \times \ldots \times H_{spec,n}) = \bigcap_{i=1}^{n} L(H_{spec,i})$ represents the set of legal sequences with respect to the entire system specification.

The so-called *admissible behaviour*, represented by the pair of languages (L_a, L_{am}), represents the sequences of events which are both possible with respect to the plant and legal with respect to the specification. Formally it is defined as

$$L_a = L(H_{spec}) \cap L(G)$$
$$L_{am} = L_m(H_{spec}) \cap L_m(G)$$

L_a is a closed language because, intuitively, the admissibility of a sequence of events logically implies the admissibility of its prefixes.

In practice, the admissible behaviour is represented by an automaton H_a such that $L_a = L(H_a)$ and $L_{am} = L_m(H_a)$. This automaton is obtained by computing the product of the plant and specification automaton $H_{spec} \times G = H_a$.

Note that when constructing H_{spec} one can ensure all of its states are marked so that the plant G exclusively determines the marking in H_a. In this case, L_{am} is given by

$$L_{am} = L_m(H_a) = L(H_{spec}) \cap L_m(G)$$

As will be seen in section 2.6, the goal will be to construct a supervisor which constrains the plant to generate sequences of events within the admissible behaviour (L_a, L_{am}).

2.4 Supervisors & controlled systems

The supervisor issues disablement commands to the plant in order to restrict its behaviour within a subset of the admissible behaviour. In the literature, a disablement command is formally referred to as a *control pattern* and is mathematically represented by a set $\gamma \subseteq \Sigma$ of enabled events. As uncontrollable events cannot be disabled, a control pattern γ must be a superset of the set of uncontrollable events Σ_u. As such, the *set of all possible control patterns* Γ is defined as

$$\Gamma = \{\gamma \in Pwr(\Sigma) | \gamma \supseteq \Sigma_u\}$$

A supervisor is a controller which generates control patterns based on sequences of events a plant has undergone. Formally, a supervisor V for a plant $G = (Q, \Sigma, \delta, q_0, Q_m)$ is defined as any map

$$V : L(G) \to \Gamma$$

Given a sequence of events $w \in \Sigma^*$, a supervisor generates a control pattern $V(w) \in \Gamma$ indicating which events are enabled. It is assumed that supervisors can only disable controllable events since control patterns are defined to be supersets of Σ_u.

The *controlled system* or *closed-loop system* V/G is the DES obtained from conjoining the supervisor V and plant G according to the feedback configuration displayed in 2.1. When G is

under supervision by V, following the generation of a string w, it is prevented from executing any disabled event $\tau \notin V(w)$. The closed behaviour of V/G, denoted $L(V/G)$, represents the set of sequences of events which G can still generate under the supervision of V. Formally, $L(V/G)$ is the language inductively defined as follows:

(i) $\epsilon \in L(V/G)$,

(ii) $s\sigma \in L(V/G)$ if $s \in L(V/G)$, $\sigma \in V(s)$ and $s\sigma \in L(G)$,

(iii) Any other string $s \in \Sigma^*$ is not in $L(V/G)$.

The marked behaviour of V/G, denoted by $L_m(V/G)$, represents the subset of tasks which can still be completed under supervision. It is defined as

$$L_m(V/G) = L_m(G) \cap L(V/G)$$

An important property of a supervisor is that it can guarantee the controlled system a minimum level of functionality by ensuring that it will always allow for the completion of some subset of important tasks. This leads to definition of a *nonblocking supervisor.*

Definition(Nonblocking): A supervisor V for a plant G is *nonblocking* if $\overline{L_m(V/G)} = L(V/G)$. In this case, the controlled system V/G is also said to be *nonblocking.* Conversely V and V/G are *blocking* if $L_m(V/G) \neq L(V/G)$.

A nonblocking supervisor ensures that the plant can never arrive at a state from which it is blocked from reaching a marked state. Intuitively, this means that a nonblocking supervisor always prevents the plant from reaching a state from which it is blocked from completing important tasks. As such, a certain degree of liveness can be ensured by requiring that the supervisor is nonblocking.

2.5 Characterizing controlled systems

The behaviour of the controlled system is inherently shaped by the supervisor. Due to the assumption that supervisors can only disable controllable events, it is not possible, in general, to construct a supervisor V for G such that $L_m(V/G) = L_m$ and $L(V/G) = L$ for arbitrary choices of $L_m \subseteq L_m(G)$ and $L \subseteq L(G)$. To characterize those languages which qualify as the marked and closed behaviour of a controlled system, the controllability property is now introduced.

Definition(Controllability): A language $K \subseteq \Sigma^*$ is said to be *controllable with respect to G* if

$$(\forall s \in \Sigma^*)(\forall \sigma \in \Sigma_u)\ s \in \overline{K}\ \&\ s\sigma \in L(G) \Longrightarrow s\sigma \in \overline{K}$$

In words, K is controllable if any string $s\sigma \in L(G)$, which is created by appending an uncontrollable event σ to one of K's prefixes V, is also a prefix of K. When K is not controllable with respect to G, it is said to be *uncontrollable* with respect to G.

Given an arbitrary plant G and a compatible supervisor V, the closed and marked behaviours of the controlled system V/G are always controllable with respect G.

In practice, the controlled system is often required to be nonblocking in order to ensure a minimum level of functionality. As such, it is important to identify what languages constitute the marked behaviour of a nonblocking controlled system. The answer to this question is one of the main results of SCT and is referred to as the "Nonblocking Controllability Theorem".

Nonblocking Controllability Theorem: Let G be an arbitrary plant. If $K \subseteq L_m(G)$ and $K \neq \emptyset$ then there exists a nonblocking supervisor V for G such that $L_m(V/G) = K$ if and only if

(i) K is controllable with respect to G,

(ii) K is $L_m(G)$-closed, meaning $K = \overline{K} \cap L_m(G)$.

When K satisfies these conditions, the nonblocking supervisor V yielding the marked behaviour $L_m(V/G) = K$ is given by

$$V(w) = \Sigma_u \cup \{\sigma \in \Sigma_c | s\sigma \in \overline{K}\}$$

2.6 Optimal nonblocking supervision

Given a plant G and admissible behaviour $L_a \subseteq L(G)$ and $L_{am} \subseteq L_m(G)$ (obtained from the specifications), the objective is to obtain a nonblocking supervisor V for G such that $L(V/G) \subseteq L_a$ and $L_m(V/G) \subseteq L_{am}$. In general, there may not exist a nonblocking supervisor V such that $L(V/G) = L_a$ and $L_m(V/G) = L_{am}$ because L_a, L_{am} may not be controllable or because $L_a \neq \overline{L_{am}}$.

This objective is met in an optimal fashion when $L(V/G)$ is the largest (in the sense of subset containment) controllable sublanguage of L_a such that $L(V/G) = \overline{L_m(V/G)}$. In other words, V is optimal when for all other nonblocking supervisors S' for G for which $L(S'/G) \subseteq L_a$ we have $L(S'/G) \subseteq L(V/G)$. In this case we say that V is maximally permissive.

To prove the existence of such a maximally permissive supervisor, one must prove the existence of such a language. To this end, define the set of all sublanguages of language E that are controllable with respect to G as

$$\mathcal{C}(E) = \{K \subseteq E | K \text{ is controllable w.r.t. } G\}$$

Since the language $\emptyset$ is always controllable, $\mathcal{C}(E)$ is nonempty for arbitrary E. Moreover, $\mathcal{C}(E)$ is closed under arbitrary unions, meaning $K \cup K' \in \mathcal{C}(E)$ $(\forall K, K' \in \mathcal{C}(E))$. From these two statements, it follows that the supremal element of $\mathcal{C}(E)$, denoted by *sup* $\mathcal{C}(E)$ always exists. Intuitively, *sup* $\mathcal{C}(E)$ is the largest controllable sublanguage of E. It is obtained as follows:

$$\sup \mathcal{C}(E) = \bigcup_{K \,\in\, \mathcal{C}(E)} K$$

It can be shown that if E is $L_m(G)$-closed then so is $\sup \mathcal{C}(E)$. Therefore, from the nonblocking controllability theorem it follows that if L_{am} is $L_m(G)$-closed and $\sup \mathcal{C}(L_{am}) \neq \emptyset$, then there exists a nonblocking supervisor V such that $L_m(V/G) = \sup \mathcal{C}(L_{am})$. Furthermore, this nonblocking supervisor V yielding the controlled behaviour $L_m(V/G) = \sup \mathcal{C}(L_{am}) \subseteq L_{am}$ and closed behaviour $L(V/G) = \overline{\sup \mathcal{C}(L_{am})} \subseteq L_a$ is maximally permissive.

2.7 Practical representation of supervisor

In practice, it is generally not possible to explicitly represent a supervisor for a plant $G = (Q, \Sigma, \delta, q_0, Q_m)$ as a map $V : L(G) \to \Gamma$ because its domain $L(G)$ may have an infinite number of strings. Instead, the map characterizing a supervisor for a plant G can be implicitly represented by a DFA $S = (Y, \Sigma, \zeta, y_0, Y_m)$ which when composed with G via fully synchronous composition yields an automaton $S \times G$ such that

$$L(S \times G) = L(V/G), \text{ and}$$
$$L_m(S \times G) = L_m(V/G).$$

The automaton S can be used to retrieve the relevant portion of V that is $V|_{L(V/G)}$, the restriction of V to $L(V/G)$. The remainder of the domain, $L(G) - L(V/G)$, is unimportant as the modification of the control patterns associated to these input strings has no effect on the closed and marked behaviours of V/G. The control pattern $V(w)$ associated to any string $w \in L(V/G)$, is obtained from S as follows

$$V(w) = \{\sigma \in \Sigma | \zeta(y_0, w\sigma)!\}$$

Intuitively, S acts as a state machine which keeps track of which events should be enabled/disabled based on its current state. After the occurrence of the sequence w the plant and supervisor will be in states $\delta(q_0, w) = q$ and $\zeta(y_0, w) = y$ respectively. At this point, the plant is only permitted to execute an event σ if $\zeta(y, \sigma)!$.

2.8 Supervisor synthesis algorithm

The development of SCT culminates in the design of *synthesis algorithms* which receive as input automaton models of the plant and specification and return as output an automaton

representation of a maximally permissive supervisor. To illustrate how these synthesis algorithms are generally implemented an we present an example of a basic implementation.

The algorithm begins by computing the product of the plant and specification automata to yield an automaton model of the admissible behaviour. In mathematical terms

$$H_a = G \times H_{spec} = (Q_p \times Q_s,\ \Sigma,\ \delta,\ (q_{0p}, q_{0s}),\ Q_{mp} \times Q_{ms})$$

where $G = (Q_p,\ \Sigma,\ \delta_p,\ q_{0p},\ Q_{mp})$ and $H_{spec} = (Q_s,\ \Sigma,\ \delta_s,\ q_{0s},\ Q_{ms})$.

If $L(H_a)$ is controllable and $L(H_a) \neq \overline{L_m(H_a)}$ then H_a constitutes the representation of a nonblocking supervisor and the algorithm just returns H_a. However, H_a does not generally satisfy these properties. In this case the main idea is to keep on removing states and transitions from H_a until it becomes a valid representation of a nonblocking supervisor. To accomplish this, a subset of states in H_a are flagged as bad. In particular, a state $q = (q_p, q_s) \in Q_p \times Q_s$ is flagged if either

i) $(\exists \sigma \in \Sigma_u) \quad \delta_{plant}(q_p, \sigma)! \quad \& \quad \delta(q, \sigma)$ is undefined or,

ii) $(\forall q_m \in Q_{mp} \times Q_{ms})(\nexists w \in \Sigma^*)\delta(q, w) = q_m$.

Clause i) flags the states in H_a which cause $L(H_a)$ to be uncontrollable with respect to G. Clause ii) flags states which cause H_a to be blocking.

The algorithm then repeatedly executes a two-step iteration which flags additional states. The first step flags states from which a bad state can be reached via an uncontrollable event. The second step flags states from which a marked state cannot be reached via unflagged states. This two-step iteration terminates when its execution can no longer flag additional states. The algorithm subsequently deletes all flagged states from H_a. This is followed by the deletion of all resultantly inaccessible states. Note that loose edges are gradually removed as their source or target states are deleted.

The resulting automaton, which is the output of the algorithm, is a representation of a nonblocking supervisor for the plant G given as input. Moreover, this supervisor is maximally permissive with respect to the specification H_{spec} also given as input. The following series of figures provides a generic example of an execution of the synthesis algorithm we have just described. The example starts off with figure 2.6 which illustrates the automaton H_a whose bad states have already been flagged.

2.9 State-based supervisory control

In section 2.3 we discussed how a system specification could be represented as a language by interpreting its strings as the set of legal sequences the controlled system is permitted to execute.

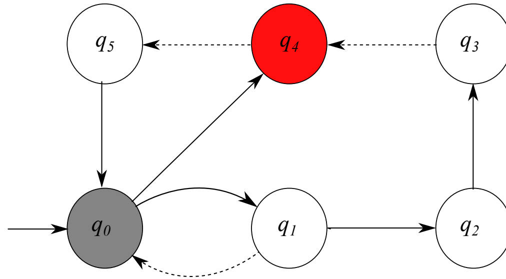


Figure 2.6: Mark bad states of H_a according to specification

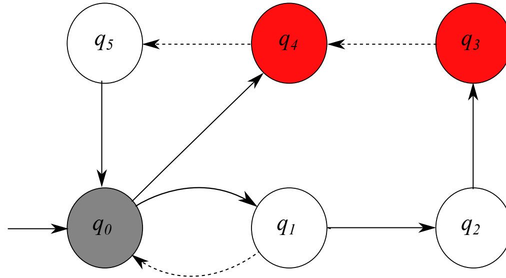


Figure 2.7: Flag states leading to bad states via uncontrollable event

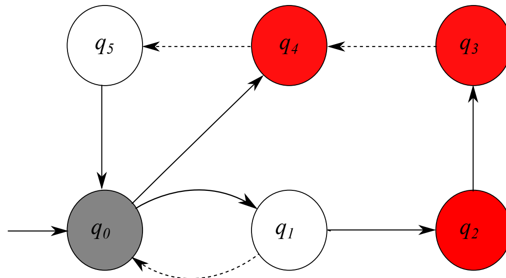


Figure 2.8: Flag states from which marked states cannot be reached via unflagged states

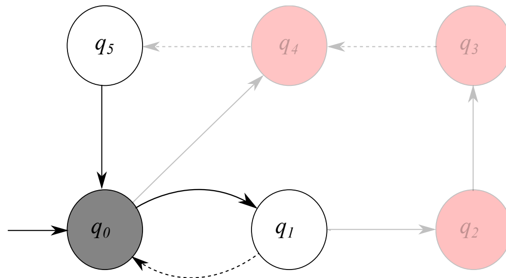


Figure 2.9: Remove bad states along with resultant loose edges

This approach is referred to as event-based SCT as control requirements are stated in terms of legal sequences of events. In event-based SCT, the concepts of system state and state transition did not play a fundamental role in the formulation of the problem statement. The (state-based) automaton models were simply introduced to serve as finite representations of regular languages.

State-based SCT (see chapter 7 of [29]) is an alternative approach in which the plant is no longer viewed in terms of languages but rather, in terms of the state-transition structure of an

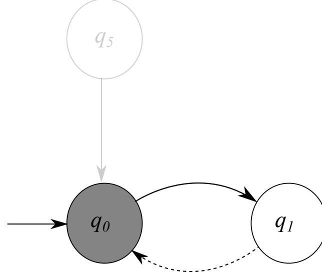


Figure 2.10: Remove inaccessible states along with resultant loose edges

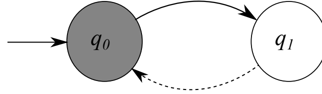


Figure 2.11: Resulting controlled behaviour

automaton. The advantage of regarding the plant as an automaton, is the ability to directly reference subsets of its states and transitions. This allows for a system specification to be expressed directly in terms of what states and transitions can be visited and executed respectively.

As the number of legal states and transitions tend to be much larger than the number of illegal states and transitions, it is typically more convenient to express the specifications in terms of what states and transitions should be avoided. As such, the following two types of specifications are used: mutual state exclusion and state-event exclusion. Informally, a mutual state exclusion specification indicates that a subset of the plant's state set should be avoided. On the other hand, a state-event exclusion specification indicates a subset of the plant's transitions should not be executed by specifying that a particular event should not be executed from a particular set of states.

These two types of specifications are used in nonblocking supervisory control of state tree structures (see chapter 4 of [19]). In the latter, a mutual state exclusion specification for a plant with state set Q is defined as a (forbidden) state predicate $P_A : Q \to \{true, false\}$ such that

$$(\forall q \in Q)\ P_A(q) = true\ \Leftrightarrow\ q \in A$$

where $A \in Pwr(Q)$. States $q \in Q$ which yield $P_A(q) = 1$, are forbidden states which must not be visited. We will use $Pred(Q)$ to denote the set of all predicates over the state set Q.

A state-event exclusion for a plant with state and event set Q and Σ, can be defined as a pair (P_A, σ) where $P_A \in Pred(Q)$ and $\sigma \in \Sigma$. Given a state-event exclusion (P_A, σ) a transition (q, τ, q') should not be executed if $P_A(q) = true$ and $\sigma = \tau$.

To illustrate these concepts, recall the two example specifications provided in table 2.1. The first

requirement is modelled as the state-event exclusion:

$$(P_{\{(\texttt{M1_broken, M2_broken})\}}, \texttt{M1_repair})$$

and the second requirement is modelled as the mutual state exclusion:

$$P_{\{(\texttt{M1_working, M2_working})\}}$$

These mutual state exclusion and state-event exclusion specifications are then used to respectively flag bad states and transitions in the plant automaton they are intended for. For instance, the latter two (state-based) specifications would be used to flag states and transitions in the automaton depicted in figure 2.4.

Chapter 3

Graph rewriting

In this chapter, the fundamental concepts of graph rewriting are presented. It is a vast and rich topic which has been thoroughly studied by computer scientists. This presentation of the theory is by no means complete. Rather, its objective is to familiarize the reader with the aspects of the theory which are relevant to our application. Interested readers may refer to [26] for a more exhaustive treatment of the topic.

The theory of graph rewriting or graph transformation was developed out of the desire to generalize textual rewriting approaches, such as Chomsky grammars and term rewriting, to deal with more complex structures. Some of the earliest developments in the field date back to the late sixties and early seventies [21, 22].

Graph rewriting refers to the rule-based modification of graphs where so called graph transformation rules are applied to graphs in order to obtain new graphs. Roughly speaking, one can think of a graph transformation rule as consisting of a precondition and a set of instructions. The precondition determines the rule's applicability to a given graph and the set of instructions specifies how the rule should transform the graph if it is applied. The following sentence therefore roughly describes the semantics of a graph transformation rule: "If the graph has property X then modify it according to the instructions in Y".

Our interest in graph transformation systems is motivated by the flexible and intuitive manner in which they can be used to model a variety of discrete event systems. Modelling via graph transformation is accomplished by using a graph to represent the system state and graph transformations to represent state transitions. The generality of this graph based modelling paradigm stems from the fact that any time a system's state can be embodied by a finite set of related entities, it can be captured by a graph in which nodes act as the entities and edges as relationships between them. The system's state evolves, as graph rewriting rules introduce or remove entities and change their relationships.

Various approaches to graph rewriting have been developed, each operating under different assumptions. The *Node replacement* approach represents one of the simplest approaches to graph rewriting as it only permits the replacement of individual nodes by subgraphs. In this approach the newly integrated subgraph is connected to the remainder of the graph according to how the node it replaced was originally connected. A more general approach is the so called *algebraic approach* where an entire subgraph can be replaced by a new subgraph. In this approach, the replacement of a subgraph with another is accomplished in two main steps. First, unwanted nodes and edges are deleted from the original subgraph and then the new subgraph is *glued* along the remaining portion of the old subgraph.

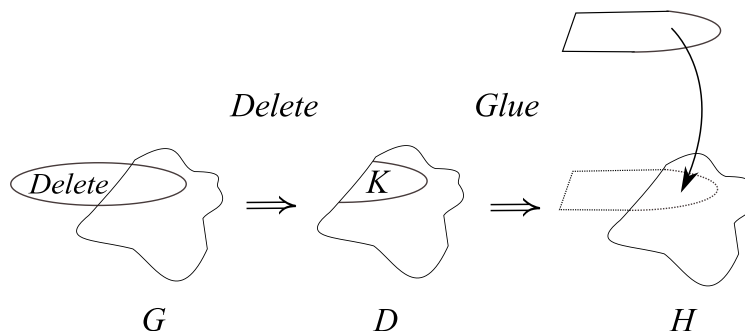


Figure 3.1: Algebraic approach to graph transformation

The general format of a graph transformation within the algebraic approach is given in figure 3.1. The operation performed in the second step, which is known as the *gluing of graphs*, is elegantly formulated in terms of a notion from category theory known as the *pushout*. The entire transformation, which involves both the deletion and gluing steps, was originally formulated as a *double pushout* in the category of graphs and total graph morphisms. Later on however, a *single pushout* in the category of graphs and partial graph morphism was shown to be an alternatively valid description. The double and single pushout approaches are not completely equivalent from an operational stand point. In particular, the conditions under which a transformation can be applied are slightly more relaxed in the single pushout approach making it slightly more general. Readers interested in a detailed presentation and comparison of the two approaches are referred to chapters 3 and 4 of [26].

For the purposes of this work we have chosen the algebraic approach to graph rewriting as it provides enough flexibility to express non-trivial system models and specifications. However, the fundamental principle underlying this work which is to use rewriting systems to intensively specify a system model and in turn generate its extensive counterpart, is independent of the rewriting approach one chooses to employ.

The remainder of this chapter is devoted to explaining the fundamental concepts underlying the algebraic approach to graph rewriting and is split up into three sections. Section 3.1 reviews background mathematics which are heavily relied upon in the following two sections. Section 3.2

then presents the fundamentals of the algebraic approach. Lastly, section 3.4 presents some key extensions to graph rewriting which we make use of in our modelling framework.

3.1 Preliminaries

This preliminaries section has been included in order to make this monograph as self-contained as possible. The first two subsections (3.1.1 and 3.1.2) respectively review concepts from graph theory and category theory needed to present the algebraic approach. Subsection 3.1.3 reviews notions of algebraic specification which are needed in defining typed attributed graphs discussed in subsection 3.4.2.

3.1.1 Graph theory

Graph theory is a branch of discrete mathematics which studies mathematical structures known as graphs [8]. Graphs, which are composed of vertices connected by edges, can be used to model binary relationships relating a collection of entities. There are many different sub-branches of graph theory which focus on different aspects of theory. In the following section, we review concepts from graph theory pertaining to graph rewriting theory.

Definition (Directed and labelled graph): Let $\Lambda = (\Lambda_V, \Lambda_E)$ be a label alphabet consisting of a set of *node labels* and *edge labels*. A *directed graph* over Λ is a 4-tuple $G = (V, E, s, t, lv, le)$ where V is a finite set of *nodes*, E is a finite set of *directed edges*, $s, t : E \to V$ are *source* and *target functions* assigning source and target nodes $s(e)$ and $t(e)$ to every edge $e \in E$ and $lv : V \to \Lambda_V$ and $le : E \to \Lambda_E$ are *labelling functions* assigning labels $lv(v)$ and $le(e)$ to every node and edge $v \in V$ and $e \in E$ respectively. The components of G can also be denoted by V_G, E_G, s_G, t_G, lv_G and le_G. The set of all graphs over Λ is denoted by $\mathcal{G}_\Lambda$.

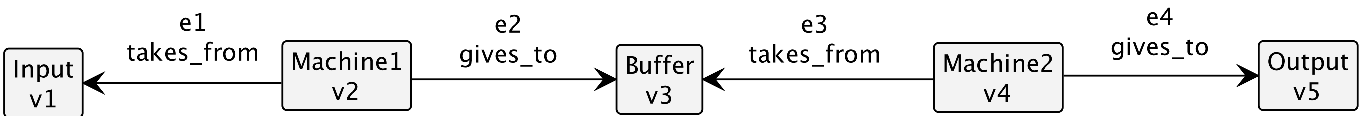


Figure 3.2: Directed graph with node and edge labels

In figure 3.2 an example of a graph directed graph with node and edge labels is given. In this example we have:

$$\Lambda = (\Lambda_V, \Lambda_E) = (\{\texttt{Input, Machine1, Buffer, Machine2, Output}\}, \{\texttt{takes_from, gives_to}\})$$

$$V = \{v_1, v_2, v_3, v_4, v_5\}$$

$$E = \{e_1, e_2, e_3, e_4\}$$

$$s(e_1) = v2,\ t(e_1) = v1$$

$$lv(v_1) = Input,\ le(e_1) = takes_from$$

Definition (Order of graph): Let $G = (V, E, s, t, lv, le)$ be a directed labelled graph. The *order* of G, denoted by $|V_G|$, is the number of nodes in G. The graph of order zero, i.e the graph on zero nodes and zero edges will be denoted by K_0.

Definition (Directed and unlabelled Graph): In this work, a *directed and unlabelled graph* G is defined as a directed and labelled graph $G = (V, E, s, t, lv, le)$ over the special label alphabet $\Lambda = (\Lambda_V, \Lambda_E) = (\{*\}, \{*\})$. The special symbol $*$ is referred to as the *empty label.* This label is omitted in the drawings of graphs.

Definition (Graph morphism): Let $G, H \in \mathcal{G}_\Lambda$. A *(total) graph morphism* $\psi : G \to H$ is a pair of functions $\psi_V : V_G \to V_H$ and $\psi_E : E_G \to E_H$ that are structure preserving in the sense that adjacent vertices in G are mapped to adjacent vertices in H and the nodes and edges in G are mapped to nodes and edges in H with the same labels. More formally:

$$\begin{aligned} \psi_V \circ s_G &= s_H \circ \psi_E, \\ \psi_V \circ t_G &= t_H \circ \psi_E, \\ lv_G &= lv_H \circ \psi_V, \\ le_G &= le_H \circ \psi_E. \end{aligned}$$

The graphs G and H are *domain* and *codomain* of ψ, respectively. A morphism ψ is said to be injective or surjective if both ψ_V and ψ_E are injective or surjective respectively. A graph morphism from G to H is illustrated in figure 3.3. The dashed vertical arrows represent the morphism's node and edge mapping components and the dashed horizontal line simply delineates the separation between domain G and codomain H. This morphism is injective but not surjective.

Definition (Graph isomorphism): A *graph isomorphism* from G to H is a graph morphism in which ψ_V, ψ_E are bijective. We say G and H are isomorphic when there exists a graph isomorphism between them and write $G \cong H$.

Definition (Subgraph): Let $G, H \in \mathcal{G}_\Lambda$. G is a subgraph of H, denoted $G \subseteq H$, if there exists an injective graph morphism ψ from G to H.

Definition (Match): Let $G, H \in \mathcal{G}_\Lambda$ and $m : G \to H$ be some a graph morphism. The *match* of G in H with respect to the graph morphism m, is the image of G in H, $m(G) \subseteq H$. In this context,

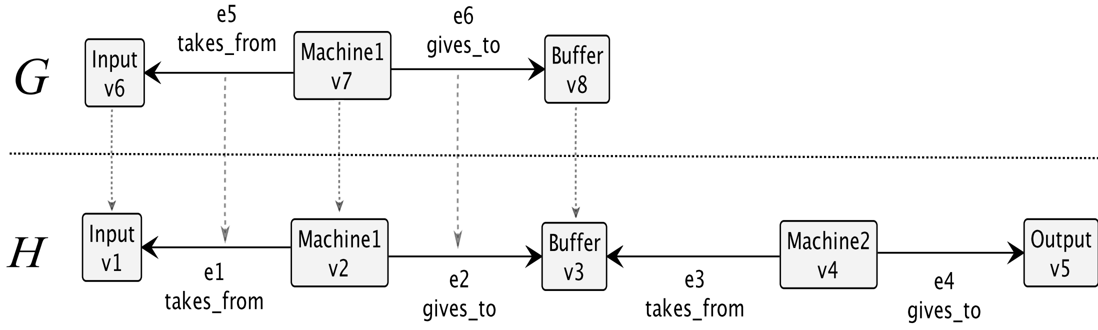


Figure 3.3: A graph morphism from G to H

the graph morphism m is referred to as a *matching*. The match $m(G)$ is said to be injective if the matching m is injective, in which case $m(G) \cong G$.

Definition (Deletion of match): Let $G, L \in \mathcal{G}_\Lambda$ be graphs and $m : L \to G$ be a graph morphism. The object $G - m(L)$, obtained by *deleting the match* $m(L)$ from G is defined as

$$G - m(L) = (V_{G-m(L)}, E_{G-m(L)})$$

where $V_{G-m(L)} = V_G - m(V_L)$ and $E_{G-m(L)} = E_G - m(E_L)$ and the minus sign in this case is taken to be set difference. Note that $G - m(L)$ is not necessarily a graph as it may contain half edges or even loose edges[1].

3.1.2 Category theory

The aim of this section is to revise the category theory fundamentals which underlie the algebraic approach and explain how these notions relate to graph rewriting. In particular, the notion of a pushout is defined and its meaning within the context of graph rewriting is discussed. Readers are referred to [4] for a comprehensive presentation of category theory.

Definition (Category): A *category* $\mathcal{C}$ is a collection $Ob_\mathcal{C}$ of *objects* denoted by $a, b, c, ..., A, B, C, ...$ and a collection $Mor_\mathcal{C}$ of *morphisms (arrows)* denoted by by $f, g, h, ...$ equipped with:

- two operations dom, cod which assign to each morphism f two objects called the *domain (source)* and *codomain (target)* of f respectively,
- an operation id assigning to each object b an *identity morphism* id_b such that $dom(id_b) = cod(id_b) = b$ (id_b is called the *identity of* b),
- a so called *composition* operation denoted $\circ$ assigning to each pair f, g of morphisms with $dom(f) = cod(g)$, a morphism $f \circ g$ such that $dom(f \circ g) = dom(g)$ and $cod(f \circ g) = cod(f)$,

[1] A half (resp., loose) edge is an edge incident on one (resp., zero) vertices.

which follows the following axioms:

> ***identity law***: for any morphisms f, g such that $cod(f) = b = dom(g)$, we have $id_b \circ f = f$ and $g \circ id_b = g$,
>
> ***associative law***: for any morphisms f, g, f such that $dom(f) = b = cod(g)$ and $dom(g) = b = cod(h)$, we have $(f \circ g) \circ h = f \circ (g \circ h)$.

The notation $f : a \to b$ is used to express that the morphism f has $dom(f) = a$ and $cod(b)$. The set of all morphisms with source a and target b is denoted by $\mathcal{C}[a, b]$. As such $f \in \mathcal{C}[a, b]$ provides an alternative means for expressing the domain and codomain of a morphism.

A few examples of categories are listed in table 3.1 below.

Category	Collection of Objects	Collection of Morphisms
Set	sets	functions
Rel	sets	relations
Grp	groups	group homomorphisms
$\mathbf{Graph}_\Lambda$	Λ-labelled graphs	(total) graph morphisms
$\mathbf{GraphP}_\Lambda$	Λ-labelled graphs	paritial graph morphisms

Table 3.1: Examples of categories

In the category **Set** the composition operation is simply the standard composition of functions and identity morphism of a given set is the bijective function mapping each element to itself. In the category **Graph** the composition of two morphisms $\psi_1 : G' \to G''$, $\psi_2 : G'' \to G'''$ yields a third morphism $\psi_2 \circ \psi_1 = \psi : G' \to G'''$ given by $\psi = (\psi_{2V} \circ \psi_{1V}, \psi_{2E} \circ \psi_{1E})$. If the graph morphisms are injective composition express the fact that $G' \subseteq G'' \subseteq G'''$.

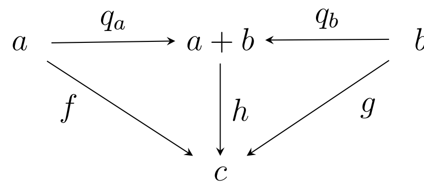


Figure 3.4: Commutative diagram for coproduct of a and b

Definition (Coproduct): Let $\mathcal{C}$ be a category and let a, b be objects in $Ob_{\mathcal{C}}$. The *coproduct* of a and b is an object denoted by $a + b$ equipped with two morphisms $q_a : a \to a + b$, $q_b : b \to a + b$ such that for any $f \in \mathcal{C}[a, b]$ and $g \in \mathcal{C}[b, c]$, there is a unique morphism $h \in \mathcal{C}[a + b, c]$ such that $h \circ q_a = f$ and $h \circ q_b = g$ or equivalently that the diagram in 3.4 commutes.

In the category **Set** the coproduct of two sets A, B is isomorphic to their disjoint union, denoted by $A \sqcup B$ and defined as

$$A \sqcup B = (A \times \{0\}) \cup (B \times \{1\}) = \{(a, 0) | a \in A\} \cup \{(b, 1) | b \in B\}$$

Likewise, in the category **Graph** the coproduct of two graphs G, H is the disjoint union of the two graphs denoted $G + H$ where the morphisms q_a, q_b play the role of identifying which nodes and edges in $G + H$ were contributed by G and H respectively.

Definition (Pushout): Let $f : a \to b$ and $g : a \to c$ be two morphisms with a common domain. A *pushout* of (f, g) is a triple $(b +_a c, g', f')$ consisting of an object $b +_a c$ and two morphisms $g' : b \to b +_a c$ and $f' : c \to b +_a c$ such that

i. $g' \circ f = f' \circ g : a \to b +_a c$, and

ii. for all other triples $(d,\ g'' : b \to d,\ f'' : c \to d)$ such that $g'' \circ f = f'' \circ g$, there exists a unique morphism $u : b +_a c \to d$ such that $u \circ g' = g''$ and $u \circ f' = f''$.

The commutative diagram shown in figure 3.5 illustrates the relationships involved in the pushout construction.

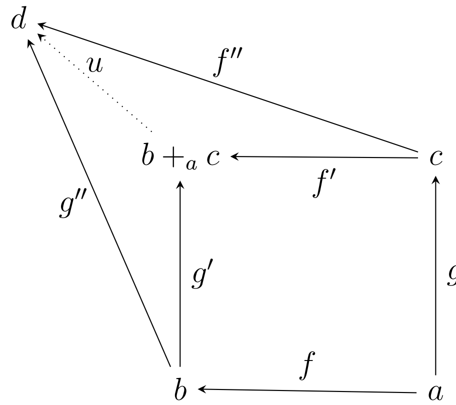


Figure 3.5: Commutative diagram of pushout construction

If a pushout of (f, g) exists, it is unique up to isomorphism. More precisely, if in the above diagram, the triple (d, g'', f'') is another pushout of (f, g), then $u : b +_a c \to d$ is an isomorphism.

In the category **Set** a pushout of two functions $f : A \to B$, $g : A \to C$ is isomorphic to the triple $(P,\ g' : B \to P,\ f' : C \to P)$ where $P = B \sqcup C / \pi$ and π is the finest equivalence relation such that

$$(\forall a \in A)\ (f(a), 0) \equiv (g(a), 1)\ (mod \pi)$$

and g' and f' are *canonical projections* associated to π respectively mapping elements in B and C to their corresponding equivalences classes.

Therefore in **Set**, the pushout can be thought of as a generalization of the disjoint union where two elements in B, C are merged if they have a common preimage in A and distinguished otherwise. Analogously in **Graph**, the pushout generalizes the disjoint union of graphs. The pushout of two graph morphisms $r : K \to R$, $d : K \to D$ is the triple (H, r^*, m^*) where H is the graph obtained by fusing the nodes and edges of R, D which have a common preimage in K and disjointly including those remaining. We say H is the graph obtained by *gluing* R and D *along* K. For any other triple (H', r'^*, m'^*) such that $r'^* \circ r = m'^* \circ d$, the graph H' will be "over-fused" with respect to K.

Lemma(Injective morphisms in graph pushouts): Let $l : K \to L$ and $d : K \to D$ be two graph morphisms sharing a common domain K. Let $(G, m : L \to G, l^* : D \to G)$ be pushout for (l, d) in the category **Graph**.

$$\begin{array}{ccc} K & \xrightarrow{l} & L \\ d \downarrow & & \downarrow m \\ D & \xrightarrow[l^*]{} & G \end{array}$$

Figure 3.6: Corresponding commutative diagram

If l is injective then l^* is injective as well[9].

In chapter 3, this lemma will provide useful information about the so-called *context graph*.

Definition (Initial and final objects): Let u be an object of a category $\mathcal{C}$. u is called *initial* if there exits exactly one morphism from u to a for every object a in $\mathcal{C}$. Conversely, u is called *final* if there exits exactly one morphism from a to u for every object a in $\mathcal{C}$.

Initial and final objects are also called universal objects. Universal objects do not always exist within a category. However, when they do exist, it can be shown that they are unique up to isomorphism. In the category **Set**, the empty set is an initial object and a singleton is a final object.

3.1.3 Algebraic specification

In this section we define notions from algebraic specification such as *data type signatures*, *many-sorted algebras* and *term algebras*. These notions will be used when defining attributed graphs. The following presentation of these notions is largely based on [11].

Definition (Signature): A *signature* $SIG = (S, OP)$ consists of a finite set of *sorts* S and a set of *constant and operation symbols* OP. The set OP is the union of pairwise disjoint subsets K_s and $OP_{w,s}$ for all $s \in S$ and $w \in S^+$.

$$OP = \bigcup_{s \in S} K_s \quad \cup \bigcup_{w \in S^+, s \in S} OP_{w,s}$$

The set K_s is the set of *constant symbols of sort* $s \in S$. The set $OP_{w,s}$ is the set of *operation symbols with argument sort* $w \in S^+$ and *range sort* $s \in S$.

An example of a signature named `natbool` possessing the sorts `nat` and `bool` is given below. In this signature `True,False,0` $\in K_s$ are constant symbols and $\neg, \wedge, ..., < \in OP_{w,s}$ are operation symbols. Note that a signature is a purely syntactic structure as the sorts are not associated to any sets and operation symbols are not associated to any functions.

```
natbool =
             sorts:     bool, nat
              opns:  True, False:  → bool
                               0 :  → nat
                               ¬ :  bool → bool
                               ∧ :  bool bool → bool
                               ∨ :  bool bool → bool
                            succ :  nat → nat
                               + :  nat nat → nat
                              == :  nat nat → bool
                               > :  nat nat → bool
                               < :  nat nat → bool
```

Definition (SIG-Algebra): An *algebra* $A = (S_A, OP_A)$ *of signature* $SIG = (S, OP)$, or *SIG-Algebra* for short, consists of two families $S_A = \{A_s | s \in S\}$ and $OP_A = \{N_A | N \in OP\}$ where

i. A_s are sets for each $s \in S$, called *base sets* or *carrier sets* of A. In this work we assume that all base sets are disjoint, that is, $(\forall s_1, s_2 \in S) A_{s_1} \cap A_{s_2}$,

ii. for all constant symbols $N \in K_s$ and $s \in S$, $N_A \in A_s$ are elements called constants of A, and

iii. for all operation symbols $N \in OP_{s_1 ... s_n, s}$, $s_1 ... s_n \in S^+$ and $s \in S$, $N_A : A_{s_1} \times A_{s_2} \times ... \times A_{s_n} \rightarrow A_s$ are functions called *operations* of A.

An algebra over the signature `natbool` is obtained by associating to each sort a carrier set and to each constant and operation symbol an element in a carrier set and a function respectively. An example of an algebra over `natbool` is given by $(\mathbb{B}, \mathbb{N}, True, False, 0, \neg, \wedge, ..., <)$, where

$True, False \in \mathbb{B}$, $0 \in \mathbb{N}$ and $\neg, \wedge, ..., <$ are actual functions. Algebras such as `natbool` are referred to as *many sorted algebras* as they possess several carrier sets.

Definition (SIG-Homomorphism): Let $SIG = (S, OP)$ be a signature and let both A, B be SIG-algebras. A *SIG-homomorphism* is a family of functions $\{f_s : A_s \rightarrow B_s | s \in S\}$ such that:

i. $f_s(N_A) = N_B$ for all $N \in OP$ and $s \in S$, and

ii. $f_s(N_A(a_1, ..., a_n)) = N_B(f_{s_1}(a_1), ..., f_{s_n}(a_n))$ for all $N \in OP$ with $N : s_1...s_n \rightarrow s$ and all $a_1 \in A_{s_1}, ..., a_n \in A_{s_n}$.

The class of Sig-Algebras together with the class of Sig-Homomorphisms form a category denoted by **CAT(SIG)**. An algebra $Z = (Z_A, OP_Z)$ whose base sets are singletons is final in **CAT(SIG)**. This is because SIG-homomorphisms are families of functions and there can only exist one function from a given set to a singleton. Note that all final objects in **CAT(SIG)** are isomorphic to one another.

Definition (Variables & Terms): Let $SIG = (S, OP)$ be a signature. For each $s \in S$ let X_s be a set called the *variables of sort* s where the sets X_s are assumed pairwise disjoint and also disjoint with OP. The family $X = \{X_s | s \in S\}$ is referred to as *a sort-indexed family of sets of variables for SIG*.

For each $s \in S$ inductively define the *set of terms over* X *of sort* s, $T_{OP,s}(X)$, by:

i. $X_s \cup K_s \subseteq T_{OP,s}(X)$ (basic terms)

ii. $N(t_1, ..., t_n) \in T_{OP,s}(X)$ (composite terms)
for all $N \in OP$ with $N : s_1...s_n \rightarrow s$ and all terms $t_1 \in T_{OP,s_1}(X), ..., T_{OP,s_n}(X)$

iii. No other terms belong to $T_{OP,s}(X)$.

The set of terms over $\emptyset$ (i.e. without variables) of sort s, $T_{OP,s}(\emptyset)$, is denoted by $T_{OP,s}$.

The *set of terms over* X and the *set of terms without variables* are respectively defined as

$$T_{SIG}(X) = \bigcup_{s \in S} T_{OP,s}(X) \qquad T_{SIG} = \bigcup_{s \in S} T_{OP,s}$$

Note that $T_{SIG} \subseteq T_{SIG}(X)$. If $A \in X_{bool}$ and $x, y \in X_{nat}$, then $\neg A, x + y, True \in T_{natbool}(X)$ are examples of terms.

For each $x \in \bigcup_{s \in S} X_s$, we inductively define the *set of terms over* X *containing the variable* x, $T^x_{SIG}(X)$ by:

i. $x \in T^x_{SIG}(X)$ (base case)

ii. $N(t_1, ..., t_n) \in T^x_{SIG}(X)$ (inductive case)
if $N(t_1, ..., t_n) \in T_{SIG}(X)$ and $\exists i \in \{1, ..., n\} \;\; t_i \in T^x_{SIG}(X)$

iii. No other terms belong to $T^x_{SIG}(X)$.

Clearly $T^x_{SIG}(X) \subset T_{SIG}(X)$ for all $x \in \bigcup_{s \in S} X_s$. Using the terms mentioned above we have $\neg A, True \notin T^x_{SIG}(X)$ and $x + y \in T^x_{SIG}(X)$.

Definition (Term Algebra): Let X be a sort-indexed family of set of variables for $SIG = (S, OP)$. The special SIG-algebra (S_T, OP_T) called the *algebra of terms* or *term algebra* is defined purely based on X and SIG as follows:

i. $S_T = \{T_{OP,s}(X) | s \in S\}$,

ii. $OP_T = \{N_T | N \in OP\}$,

iii. for all $s \in S$ and $N \in K_s$, $N_T \in T_{OP,s}(X)$ is N, and

iv. for all $N \in OP_{s_1...s_n,s}$, $s_1...s_n \in S^+$ and $s \in S$,
$N_T : T_{OP,s_1}(X) \times T_{OP,s_2}(X) \times ... \times T_{OP,s}(X) \to T_{OP,s}(X)$ is a function and is defined by $N_T(t_1, ..., t_n) \coloneqq N(t_1, ..., t_n)$ for all $t_1 \in T_{OP,s_1}(X), ..., t_n \in T_{OP,s_n}(X)$.

In the literature this algebra is denoted by $T_{SIG}(X)$, the same notation used for set of terms over X, SIG.

Definition (Variable Assignment): Let X be a sort-indexed family of sets of variables for $SIG = (S, OP)$ and A be a SIG-Algebra. An *assignment* α for X is a family of functions $\{\alpha_s : X_s \to A_s | s \in S\}$ assigning to each variable in X a value in A. The *extended assignment* or simply *extension of* α, $\overline{\alpha}$ is a family of functions $\{\overline{\alpha}_s : T_{OP,s}(X) \to A_s | s \in S\}$ and is recursively defined by:

i. $\overline{\alpha}_s(x) = ass_s(x)$ for all $x \in X_s$

ii. $\overline{\alpha}_s(N) = N_A$ for all $N \in K_s$

iii. $\overline{\alpha}_s(N(t_1, ..., t_n)) = N_A(\overline{\alpha}_s(t_1), ..., \overline{\alpha}_s(t_n))$ for all $N(t_1, ..., t_n) \in T_{SIG}(X)$.

Note that the extended assignment $\overline{\alpha} : T_{SIG}(X) \to A$ is actually a SIG-homomorphism from $T_{SIG}(X)$ to A. With $x, y \in X_{nat}$ and $\alpha_{nat}(x) = 4$, $\alpha_{nat}(y) = 6$, we have $\overline{\alpha}_{nat}(x + y + 0) = (\overline{\alpha}_{nat}(x) + \overline{\alpha}_{nat}(y) + \overline{\alpha}_{nat}(0)) = (\alpha_{nat}(x) + \alpha_{nat}(y) + 0) = (4 + 6) = 10$.

It can be shown that given a fixed X and assignment α, there is a unique SIG-homomorphism $\overline{\alpha}$ which makes the diagram in figure 3.7 commute where $u : X \hookrightarrow T_{SIG}(X)$ is the inclusion map $u(x) = x$ [11]. Furthermore, this unique SIG-homomorphism $\overline{\alpha}$ is the extended assignment defined above. The converse statement, which is that given a SIG-homomorphism $\overline{\alpha} : T_{SIG}(X) \to A$ there is a unique assignment $\alpha : X \to A$ which makes the diagram in figure 3.7 commute, follows from the fact that $X_s \subset T_{OP,s}(X)$ for all $s \in S$.

$$\begin{array}{ccc} X & \xrightarrow{\alpha} & A \\ & {\scriptstyle u}\searrow \quad \nearrow{\scriptstyle \overline{\alpha}} & \\ & T_{SIG}(X) & \end{array}$$

Figure 3.7: Satisfiability criterion for atomic constraints

3.2 Algebraic approach

Having presented the necessary graph theory and category theory fundamentals, we now present the algebraic approach to graph rewriting. As mentioned above, there are two slightly different variations of the algebraic approach, known as the *double and single pushout approach*. As the double pushout variant conveys all the main ideas underlying the algebraic approach, we see no added benefit in presenting its slightly more complex and general single pushout counterpart. For the sake of simplicity, we chose to present the double pushout approach.

3.2.1 Double pushout approach

Definition (Graph transformation rule): A graph transformation (or production) rule (GTR) $t = \langle L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\rangle$[2] over $\mathcal{G}_\Lambda$ is a so-called *rule span* $\langle L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\rangle$ consisting of three graphs $L, K, R \in \mathcal{G}_\Lambda$ along with two graph morphisms $l : K \to L$ and $r : K \to R$ where l is assumed to be injective. A rule t is said to be injective if r is injective. The graphs L and R are called the *left hand side* and *right hand side* respectively. K is referred to as the *interface* or *gluing graph*. The interface K represents what the left and right hand sides have in common. If $G \in \mathcal{G}_\Lambda$ and m is a graph morphism of the form $m : L \to G$ we say m is a *matching for t in G*.

Figure 3.8 gives an example of a graph transformation rule `wp_from_M1_to_Buffer` and a matching for it in G.

[2]This notation is used to emphasize the fact that both graph morphisms ψ_1, ψ_2 have the same domains.

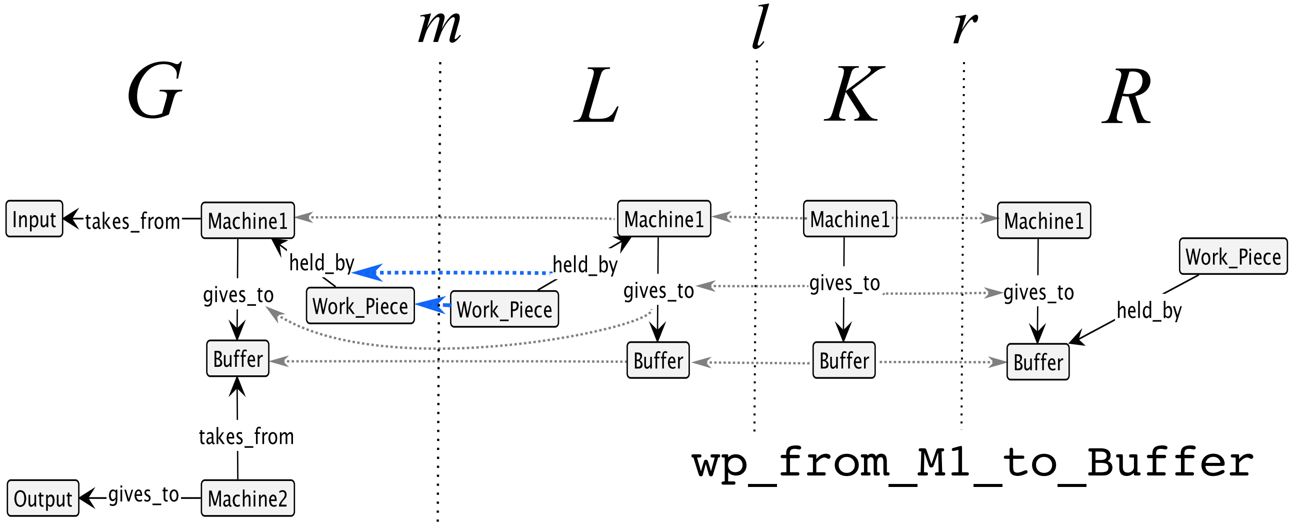


Figure 3.8: A graph transformation rule and its matching in G

Three graphs are technically required to explicitly specify a graph transformation rule. However, in eliminating redundant information common to L, K and R, it is possible to express a rule via a single graph. This compact graphical representation is employed within the graph transformation tool known as GROOVE (see section 3.3). Figure 3.9 illustrates how the rule `wp_from_M1_to_Buffer` would be represented in GROOVE.

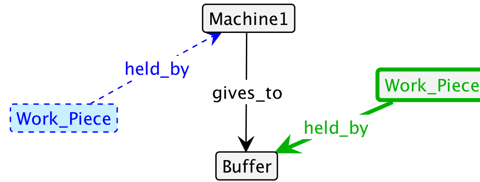


Figure 3.9: Compact representation of `wp_from_M1_to_Buffer`

The elements of the graph drawn in black correspond to those of K. They are called *readers* because they are matched and preserved by the rule. Those drawn in blue are called *erasers* as they are matched and deleted by the rule. They belong to $L - l(K)$. Those drawn in green are called *creators* as they created by the rule. They belong to $R - r(K)$. Rules will be represented in this way throughout the remainder of this work.

Definition (Gluing condition): Let $G \in \mathcal{G}_\Lambda$, $t = \langle L \leftarrow K \rightarrow R \rangle$ be some transformation rule over $\mathcal{G}_\Lambda$ and $m : L \rightarrow G$ be some matching for t in G. The matching m *satisfies or maintains the gluing condition for* t if the following two sub-conditions are satisfied:

1. *Dangling condition*:

$$\left(\forall e \in E_{G-m(L)}\right) s(e) \notin V_{m(L-l(K))} \;\wedge\; t(e) \notin V_{m(L-l(K))}$$

In other words, the dangling condition stipulates that no edge in $G - m(L)$ [3] is incident to a node deleted by the rule. This has the effect that if a node is deleted, then so is any edge adjacent to it.

The dashed blue arrows in figure 3.8 indicate, $m(L - l(K))$, the portion of L in G which will be deleted when the rule is applied. One can verify that the matching m satisfies the dangling condition for `wp_from_M1_to_Buffer` since the deletion of $m(L - l(K))$ from G does not leave behind any dangling edges. If G also possessed an edge between the nodes labelled "Work_Piece" and "Input" for example, m would no longer satisfy the dangling condition for `wp_from_M1_to_Buffer`.

2. *Identification condition*:

$$(\forall v, v' \in V_L)\, v \neq v' \Longrightarrow \big[m(v) = m(v') \Longrightarrow v, v' \in V_{l(K)}\big]$$
$$(\forall e, e' \in E_L)\, e \neq e' \Longrightarrow \big[m(e) = m(e') \Longrightarrow e, e' \in E_{l(K)}\big]$$

This says that for all distinct nodes (edges) x, x' in L, $m(x) = m(x')$ only if x, x' are both in K. More intuitively, a pair of nodes (edges) $x, x' \in L$ may only be mapped non-injectively, if neither of the two nodes (edges) are erased by the rule. The reasoning behind the identification condition is more subtle. It is essentially imposed to prevent matches which lead to ill-defined applications of graph transformation rules. Note that the identification condition is always satisfied for injective matchings.

Definition (Applicability of a graph transformation rule): Let $G \in \mathcal{G}_\Lambda$ and $t = \langle L \leftarrow K \rightarrow R \rangle$ be some transformation rule over $\mathcal{G}_\Lambda$. The graph transformation rule p is said to be *applicable* to a source graph G if there exists a matching $m : L \rightarrow G$ for t in G which satisfies the *gluing condition*.

If the existence of an injective matching is prerequired to apply t, then t is only applicable to G when $L \subseteq G$, where L is the left hand side of t.

Definition (direct derivation): Let $G, H \in \mathcal{G}_\Lambda$, $\mathcal{R}$ be a set of graph transformation rules over $\mathcal{G}_\Lambda$ and $t \in \mathcal{R}$ be applicable to G. Say G *directly derives* H *through* t, denoted by $G \Rightarrow_t H$ if (1) and (2) are graph pushouts in figure 3.10.

Simply put, a direct derivation $G \Rightarrow_t H$ is the application of a rule t to a *source graph* G, which yields a *target graph* H. When necessary, the notation $G \Rightarrow_{t,m,m^*} H$ can be used in order to specify the matching $m : L \rightarrow G$, with respect to which the rule t is applied and the so called *comatch*, $m^* : R \rightarrow H$, of t in H. Both subscripts may be dropped when t, m are clear from context or irrelevant. Write $G \Rightarrow_{\mathcal{R}} H$ if there exists a $t \in \mathcal{R}$ such that $G \Rightarrow_t H$.

[3] $G - m(L)$ is the structure obtained by removing the contents of $m(L)$ from G. Note that the resulting structure is not necessarily a graph as there is no guarantee that all of its remaining edges will have both source and target nodes.

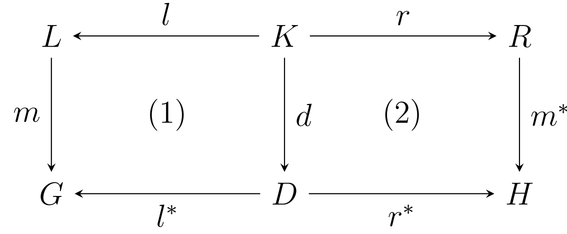


Figure 3.10: Double push out requirement for direct derivation

Given a source graph $G \in \mathcal{G}_\Lambda$ and a rule t over $\mathcal{G}_\Lambda$, applying t to G to obtain a graph H comprises the following steps:

i. Find a matching m for t in G (or equivalently find a match of L in G, $m(L)$).

ii. Verify the match m does not violate the gluing condition.

iii. Remove $m(L - l(K))$ from G to obtain the so called *context graph*[4] D and morphism d. That is, $D = G - m(L - l(K))$ is the graph obtained by removing from G the nodes and edges of L which are not in the image l.

iv. Construct the pushout of $D \overset{d}{\leftarrow} K \overset{r}{\rightarrow} R$ to obtain the target graph H. That is construct H by gluing R and D along K according to morphisms r and d.

As morphism l is by definition injective for all transformation rules, it follows from the lemma of section 3.1.2 that l^* is also always injective. This implies that $D \subseteq G$. When a transformation rule is injective(morphism r is injective), $G \supseteq D \subseteq H$.

Step i is by far the most computationally intensive step involved in this procedure. When only injective matchings are permitted this step is simplified and involves solving an instance of the subgraph isomorphism problem which is NP-complete [7]. When non-injective matchings are accepted, find a matching involves solving an instance of the graph matching problem which is also NP-complete [1]. In section 5.5, more information is provided on the complexity of each step of a direct derivation.

To visualize step iii consider graph D in figure 3.11 and graphs G, L in figure 3.8. The portion of L which has no preimage in K is deleted from G to obtain D. As for step iv, we illustrate in figure 3.11 how R, D are glued along K to obtain H. Note that a rule may be applied to a given graph via several different matchings.

[4] D is a graph, as the gluing condition ensures that the source and target nodes of each edge in D are also nodes in D.

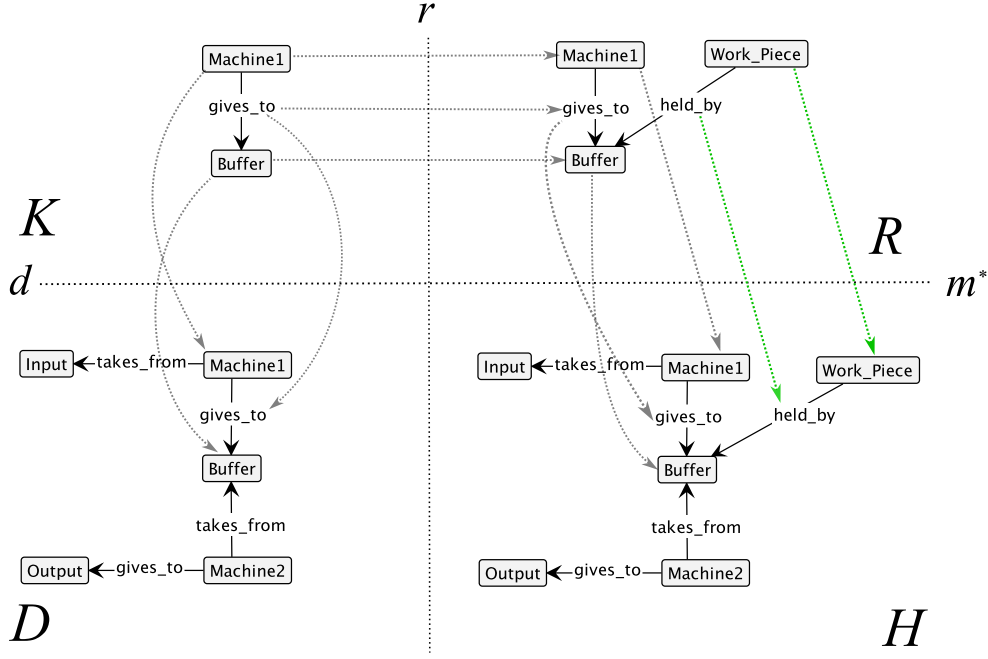


Figure 3.11: Construction of pushout of $D \leftarrow K \rightarrow R$

Definition (Derivation): Let $G, G_0, G_1, ..., G_n, H \in \mathcal{G}_\Lambda$ and $\mathcal{R}$ be a set of graph transformation rules over $\mathcal{G}_\Lambda$. A *derivation* from G to H through $\mathcal{R}$ consists of a sequential composition of direct derivations of the form $G = G_0 \Rightarrow_\mathcal{R} G_1 \Rightarrow_\mathcal{R} ... \Rightarrow_\mathcal{R} G_n \cong H$. A derivation from G to H of unspecified finite length over $\mathcal{R}$ is denoted by $G \overset{*}{\Rightarrow}_\mathcal{R} H$. The relation $\overset{*}{\Rightarrow}_\mathcal{R}$ is the transitive-reflexive closure of $\Rightarrow_\mathcal{R}$. The notation $G \overset{n}{\Rightarrow}_\mathcal{R} H$ may be used when the number of steps in the transformation is made explicit.

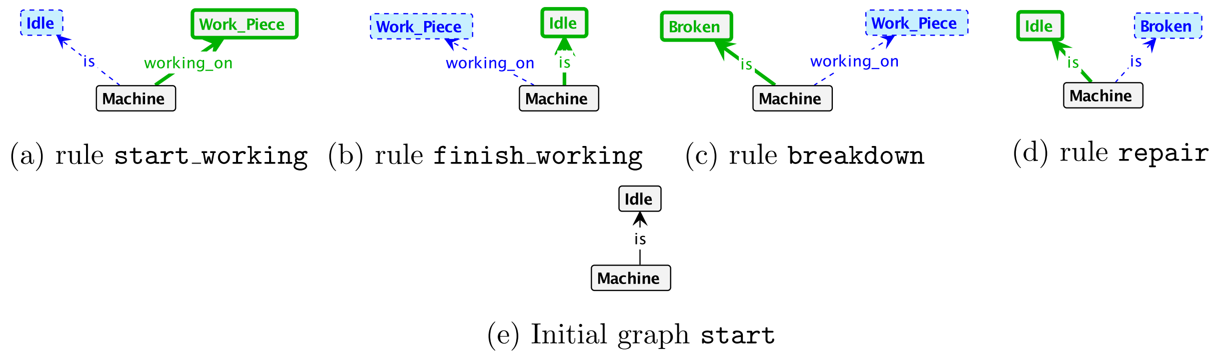
Definition (Graph transformation systems): A *graph transformation system* (GTS) $gts = (\mathcal{R}, \Lambda)$ over $\mathcal{G}_\Lambda$, consists of label alphabet Λ and a set of graph transformation rules $\mathcal{R}$ over $\mathcal{G}_\Lambda$. Given a $gts = \langle \mathcal{R}, \Lambda \rangle$ and a set of *initial graphs* $\mathcal{G}_0 \subseteq \mathcal{G}_\Lambda$, the *set of reachable graphs*, denoted $R(gts, \mathcal{G}_0)$, is defined as $R(gts, \mathcal{G}_0) = \{G \in \mathcal{G}_\Lambda | G_0 \overset{*}{\Rightarrow}_\mathcal{R} G, G_0 \in \mathcal{G}_0\}$.

Figure 3.12 displays a GTS called `Machine`. Subfigures 3.12a-3.12d illustrate `Machine`'s transformation rules and subfigure 3.12e shows its initial graph.

Definition (Induced graph transition system): The *graph transition system* $T = (S, \rightarrow, S_0)$ *induced* by the graph transformation system $gts = (\mathcal{R}, \Lambda)$ equipped with initial graphs $\mathcal{G}_0 \subseteq \mathcal{G}_\Lambda$ consists of a *state set* $S = \{G | G \in R(gts, \mathcal{G}_0)\}$ of reachable graphs, a *transition relation* $\rightarrow \subseteq S \times \mathcal{R} \times \mathcal{MOR}_{\mathbf{Graph}_\Lambda} \times S$,[5] where $(G, t, m, H) \in \rightarrow$ iff $G \Rightarrow_{t,m} H'$ with $H' \cong H$ and *an initial state set* $S_0 = \mathcal{G}_0 \subseteq S$. We call T finite if $|S| \in \mathbb{N}$ and infinite otherwise.

A graph transition system is finite or infinite, depending on the GTS and the initial state set it is induced by. If for example, a GTS possesses a rule which unconditionally always adds a node to

[5]Recall from section 3.1.2 that $\mathcal{MOR}_{\mathbf{Graph}_\Lambda}$ is the class of morphisms of the category $\mathbf{Graph}_\Lambda$.



(a) rule `start_working` (b) rule `finish_working` (c) rule `breakdown` (d) rule `repair`

(e) Initial graph `start`

Figure 3.12: GTS `Machine` equipped with inital graph `start`

the source graph, it will induce an infinite graph transition system regardless of the initial states it is paired with.

In figure 3.13 we illustrate the finite graph transition system induced by the pair (`Machine`, `start`) which are displayed above in figure 3.12.

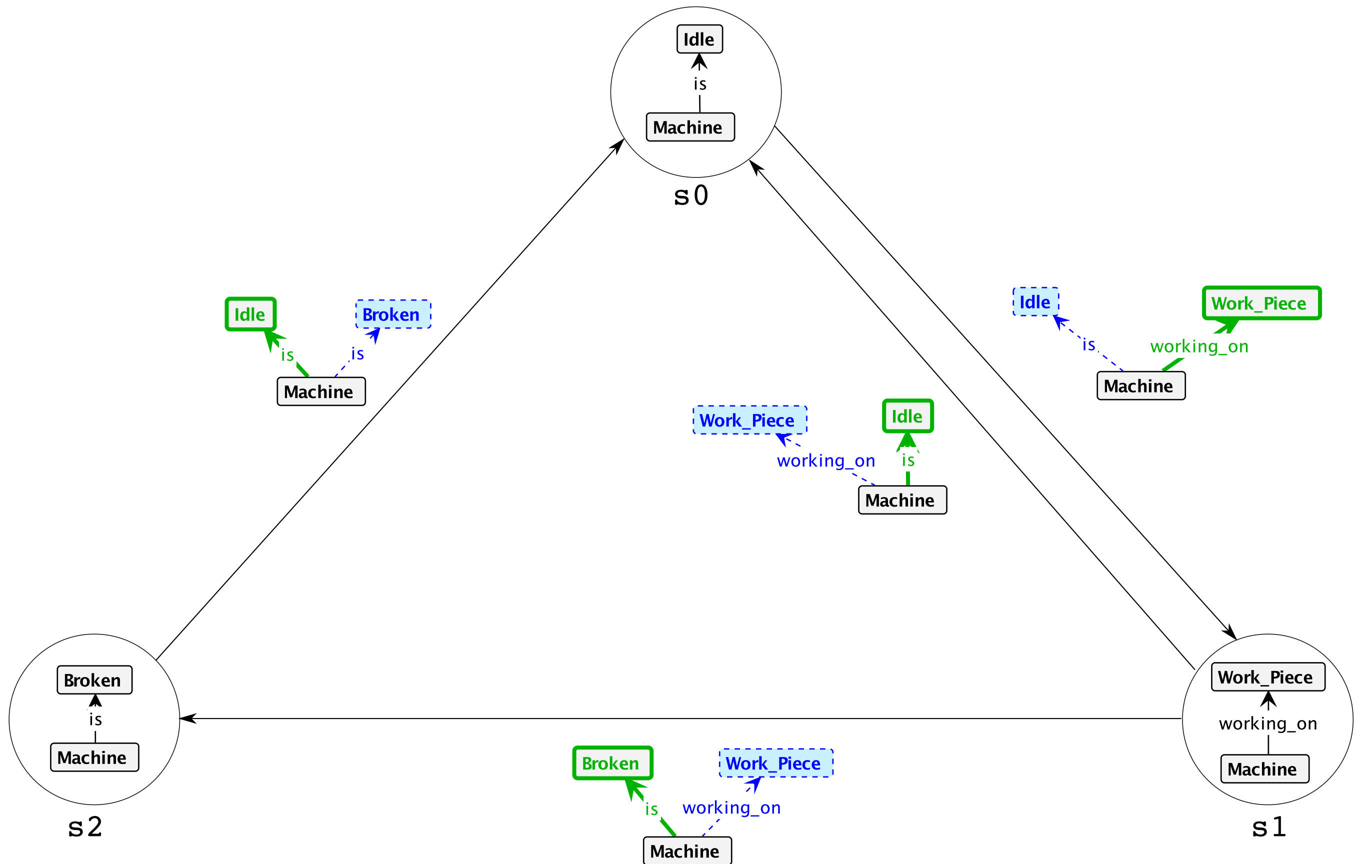


Figure 3.13: Graph transition system induced by (`Machine`, `start`)

From a practical standpoint, a graph transition system is built by recursively applying all applicable transformation rules (of the GTS it is induced by) starting from the set of initial graphs. When identical states (isomorphic graphs) are encountered they are merged. This

recursive process can potentially never end if the graph transition system is infinite. In the case where it is finite, the process terminates when no new transitions can be explored. Note that the order in which transitions are added to the graph transition system depends on the search strategy employed i.e. depth first search, breadth first search, etc.

In order to represent a graph transition system more compactly, the graphs underlying the states may be omitted and the rule spans associated to the transitions may be replaced by corresponding rule names. Figure 3.14, illustrates how the graph transition depicted in figure 3.13 is compactly represented.

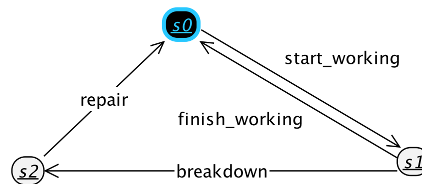


Figure 3.14: Compact representation of the graph transition system induced by (`Machine`, `start`)

3.3 Graph transformation tools

Many graph transformation tools (GTT) implementing the algebraic approach have been developed. These software tools are geared towards a variety of different applications including but not limited to model transformation, model checking, rapid system prototyping and state space exploration. In all cases, the primary function of these tools is to receive as input graph transformation systems together with a set of initial graphs and automate the process of constructing the induced graph transition system. Some of these tools comprise a graphical user interface which can be used to graphically create and edit transformation rules and graphs. Others are terminal-based and require the user to specify graphs and graph transformation rules textually.

GROOVE is one such graph transformation tool developed at the university of Twente. It implements the *single pushout approach*[18] which is very similar to the double pushout approach. It is intended as a multi-purpose tool which can be used for model checking, model analysis, model transformation, rapid system prototyping, state space exploration, etc.

Due to its versatility GROOVE was chosen as the supporting GTT for this research. Although it is not geared toward generating controlled DES models, it possesses all necessary features required to illustrate how a GTT can be used to do so. Furthermore, its intuitive graphical user interface helps illustrates the user-friendliness of this modelling approach.

3.4 Extensions

New concepts extending the core of graph rewriting theory are continually developed in academia. These theoretical extensions are progressively integrated into graph transformation tools such as GROOVE in order to improve their modelling capabilities. In this section we present five features integrated within GROOVE which play a key role within our proposed modelling framework. Although these concepts are presented within the context of the double pushout approach, the general ideas behind them are not specific to any particular approach.

3.4.1 Typed graphs

In graph rewriting a universe of discourse is established to specify what class of graphs is of interest and what graphs are in some sense disallowed. Up until this point, we have assumed $\mathcal{G}_\Lambda$ to be our domain of discourse. That is, we have constrained all the involved graphs to be defined over label alphabet Λ. However, specifying the domain of discourse via a label alphabet only restricts which labels may appear on nodes and edges without imposing any restrictions on how they may be connected to one another. For some applications, including ours, this mechanism may not provide enough precision for choosing the domain of discourse. Type graphs constitute an alternative means for specifying the allowable class of graphs. Type graphs generalize label alphabets as they specify the set of allowable *node and edge types* as well as how these node and edge types may be connected. The following presentation of typed graphs is based on [16].

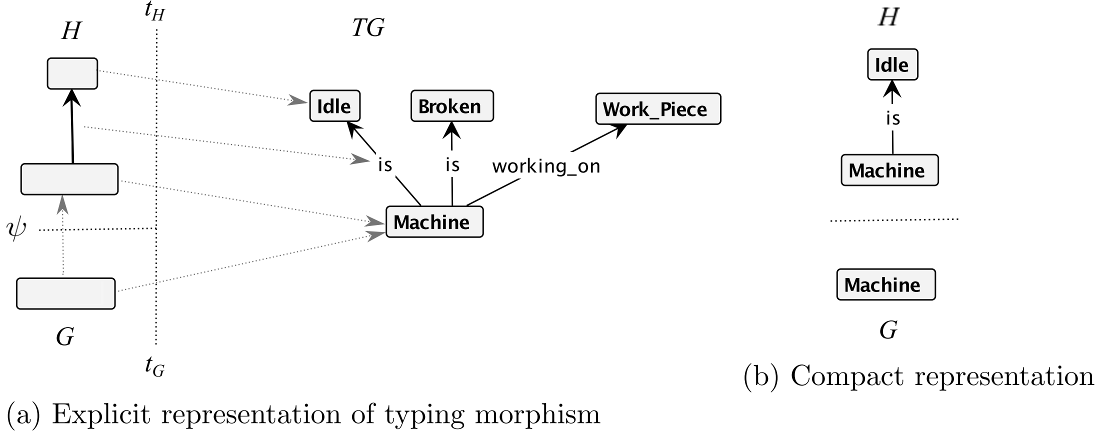
Definition (Typed graph): Let TG be a fixed unlabelled graph. A *TG-typed graph* or a *graph typed over* TG typically denoted $(t_G : G \to TG)$ consists of an unlabelled graph G and a graph morphism $t_G : G \to TG$ called the *typing morphism*, which intuitively assigns a “type” to each node and edge of G. We call TG a *type graph* and G an *instance graph*. We use $\mathcal{G}_{TG}$ to denote the set of all TG-typed graphs.

Note that when typing is employed graphs are not labelled. Instead, each node and edge of the type graph is implicitly understood to represent a different type and the nodes and edges of an instance graph are assigned one such type via the typing morphism. For convenience however, the edges and nodes of type graphs are informally labelled to indicate the type of each edge and node.

Definition (Morphism of typed graphs): Let TG be a fixed type graph. A *morphism of TG-typed graphs* $\psi : (t_G : G \to TG) \to (t_H : H \to TG)$ is a graph morphism $\psi : G \to H$ which *preserves typing* with respect to TG, i.e, $t_H \circ \psi = t_G$.

The set of TG-typed graphs together with the set of morphisms of TG-typed graphs defines the category $\mathbf{Graph_{TG}}$. In figure 3.15a we give an example of two TG-typed graphs $(t_G : G \to TG)$, $(t_H : H \to TG)$ and a TG-typed morphism ψ. In this figure $(t_G : G \to TG)$, $(t_H : H \to TG)$’s

typing morphisms are explicitly represented. An informal alternative representation of the TG-typed instance graphs is depicted in figure 3.15b, where the typing morphisms are implicitly specified via labels indicating the types of the nodes and edges and respecting the constraints embodied by the structure of TG. To avoid confusion we refer to these informal labels as *type labels*.



(a) Explicit representation of typing morphism

(b) Compact representation

Figure 3.15: Two TG-typed graphs ($t_G : G \to TG$), ($t_H : H \to TG$) and a TG-typed morphism ψ

Definition (Typed graph transformation rule): Let TG be a fixed type graph. A *TG-typed graph transformation rule* $t = \langle L \xleftarrow{l} K \xrightarrow{r} R\rangle$ is a graph transformation where $L, K, R \in \mathcal{G}_{TG}$ and l, r are morphisms of TG-typed graphs.

Definition (Typed graph transformation system): Let TG be a fixed type graph. A *typed graph transformation system* (TGTS) $tgts = (\mathcal{R}, TG)$ *over type graph* TG, consists of a type graph TG and a set of TG-typed graph transformation rules $\mathcal{R}$. Given a TGTS $tgts = \langle \mathcal{R}, TG\rangle$ and a set of *initial graphs* $\mathcal{G}_0 \subseteq \mathcal{G}_{TG}$, the *set of reachable graphs*, denoted $R(tgts, \mathcal{G}_0) \subseteq \mathcal{G}_{TG}$, is defined as $R(tgts, \mathcal{G}_0) = \{G \in \mathcal{G}_{TG} | G_0 \overset{*}{\Rightarrow}_{\mathcal{R}} G, G_0 \in \mathcal{G}_0\}$.

3.4.2 Typed attributed graphs

Heretofore, nodes have been used to represent any entity in a system. The theory of attributed graphs introduces additional structure which distinguishes entities representing objects from those representing object attributes (i.e information about objects). This is accomplished by using designated *data vertices* to express information about objects and using *object vertices* to represent the objects themselves. In this context, edges relating data and objects are referred to as *attributes* and those relating two objects are referred to as *links*. Each attribute edge has a name and a data type and the data vertex it is incident to holds a value of that type. The

machinery of algebraic specification presented in section 3.1.3 is used to define the mathematical structure of data types.

This treatment of attributed graphs closely follows that of [15]. In this formulation, attribute edges may only originate from object nodes and lead to data vertices. Hence, these kinds of graphs are referred to as node-attributed graphs. However, generalizations permitting edge-attribution have also been developed [12] by employing so-called *E-graphs* which allow edges to originate from other edges. To facilitate the presentation of these notions we first start by defining attributed graphs without typing and then reintroduce typing and define typed attributed graphs.

Throughout the remainder of this thesis we fix $SIG = (S, OP)$ to be a specific data type signature over sort set S and constant and operation symbol set OP. As such, all attributed graphs which follow, are attributed over this data type signature.

Definition (Attributed graph): An *attributed graph over the algebraic signature SIG or a SIG-attributed graph* $AG = (G_{AG}, A_{AG})$ consists of a graph G_{AG} and *SIG-Algebra* $A_{AG} = (S_{A_{AG}}, OP_{A_{AG}})$ such that

i. $\bigcup_{s\in S} A_{AG,s} \subseteq V_{G_{AG}}$, and

ii. $(\forall e \in E_{G_{AG}})s(e) \notin \bigcup_{s\in S} A_{AG,s}$

where $S_{A_{AG}} = \{A_{AG,s}|s \in S\}$.

$DV_{AG} = \bigcup_{s\in S} A_{AGs}$ is referred to as the *set of data vertices of* AG and $OV_{AG} = V_{G_{AG}} - DV_{AG}$ is referred to as the *set of object vertices of* AG. Note that there are no edges leaving data vertices. The *set of attributes* of AG is defined as $Attr_{AG} = \{e \in E_{G_{AG}}|t(e) \in DV_G\}$ and the *set of links* of AG is defined as $Link_{AG} = E_{G_{AG}} - Attr_{AG}$. For convenience we define the set of *involved data vertices* as $IDV_{AG} = \{d \in DV_{AG}|(\exists e \in Attr_{AG})\ t(e) = d\}$. The *structural component of the graph* of AG is defined as $Graph(AG) = G_{AG} - (DV_{AG} + Attr_{AG})$; it consists solely of objects and links.

Definition (Attributed graph morphism): Given two SIG-attributed graphs $AG_1 = (G_{AG_1}, A_{AG_1})$ and $AG_2 = (G_{AG_2}, A_{AG_2})$, a *SIG-attributed graph morphism* $\psi = (\psi_G, \psi_A) : AG_1 \to AG_2$ consists of a graph morphism $\psi_G = (\psi_V, \psi_E) : G_{AG_1} \to G_{AG_2}$ and a *SIG-homomorphism* $\psi_A = \{\psi_s : A_{AG_1,s} \to A_{AG_2,s}|s \in S\} : A_{AG_1} \to A_{AG_2}$ such that $\bigsqcup_{s\in S} \psi_s \subseteq \psi_V$ where $\bigsqcup_{s\in S} \psi_s : DV_{AG_1} \to DV_{AG_2}$ is the function defined by

$$(\forall s \in S)(\forall a \in DV_{AG_1})\ \bigsqcup_{s\in S} \psi_s(a) = \psi_s(a)$$

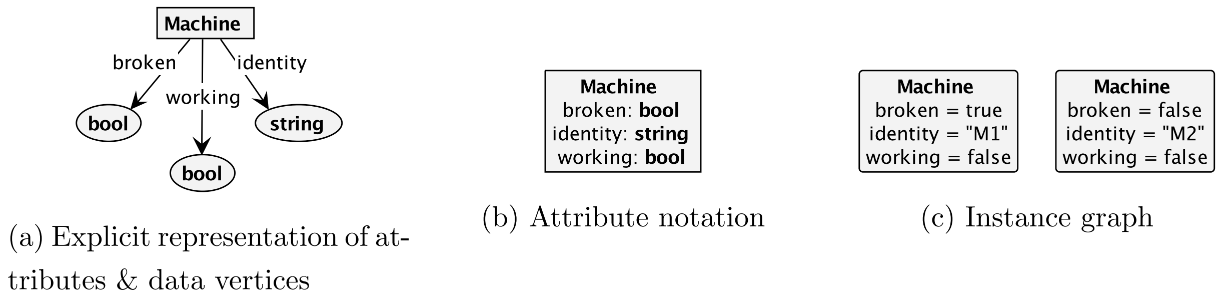
This notation is used because $\bigsqcup_{s\in S} \psi_s$ is obtained by taking the disjoint union of the functions ψ_s[6], viewed as relations and $\bigsqcup_{s\in S} \psi_s \subseteq \psi_V$ means that the former function is a restriction of the latter.

[6]The disjoint union is used in place of regular union in order to ensure that the resulting relation is in fact a function.

The collection of SIG-attributed graphs together with the collection of SIG-attributed graph morphisms form the category of SIG-attributed graphs **AGraph(SIG)**.

Definition (Attributed type graph): A *SIG-attributed type graph* is a SIG-attributed graph $ATG = \langle TG, Z \rangle$ over the final[7] SIG-algebra Z possessing singleton carrier sets $Z_s = \{s\}$ for each $s \in S$.

In figure 3.16a we give an example of an attributed type graph where the data vertices `bool`, `string` and attributes `broken`, `identity` are explicitly drawn as nodes and edges respectively. The `Machine`, `Work_Piece` nodes are object vertices and the `held_by` edge is a link. Note that data vertices are drawn as ellipses to differentiate them from object nodes drawn as rectangles. For convenience, attributes and their values may be inscribed within the object vertex they belong to as demonstrated in figure 3.16b and 3.16c. This is referred to as attribute notation. In the context of type graphs, an attribute's associated data vertex specifies its sort not its value. In this context, attributes are intended as attribute declarations which specify which attributes objects can potentially have.



(a) Explicit representation of attributes & data vertices

(b) Attribute notation

(c) Instance graph

Figure 3.16: An attributed type graph ATG and an ATG-typed graph

Definition (Typed Attributed graph): Let $ATG = \langle TG, Z \rangle$ be a SIG-attributed type graph. An *ATG-typed graph* $\langle AG, ag \rangle$ *attributed over algebra* A_{AG} consists of a SIG-attributed graph $AG = (G_{AG}, A_{AG})$ equipped with an attributed graph morphism $ag = (ag_G, ag_A)$ where $ag_G : G_{AG} \to TG$ and $ag_A : A_{AG} \to Z$.

In figure 3.16c we have an example of a typed attributed instance graph. In general, attributed instance graphs technically have an infinite number of vertices. This occurs when ever an infinite data type is present such as $\mathbb{N}$ since the graph must possess a vertex for each $n \in \mathbb{N}$. As will be seen shortly however, it is assumed that data vertices are never deleted by rules. Therefore it is only necessary to draw the involved data vertices $d \in IDV_{AG}$. In our example, the instance graph has infinite number of string data vertices. However, we only specify the string data vertices `"M1","M2"`.

[7] Z is called a final SIG-algebra because it is a final object in the category **CAT(SIG)**.

Definition (Typed Attributed graph morphism): Let ATG be a SIG-attributed type graph and $\langle AG_1, ag_1\rangle, \langle AG_2, ag_2\rangle$ be two ATG-typed graphs where

$$\begin{aligned} AG_i &= (G_{AG_i}\ ,\ A_{AG_i}), && \text{for } i = 1,2 \\ ag_i &= (ag_{i,G}\ ,\ ag_{i,A}), && \text{for } i = 1,2 \end{aligned}$$

A *typed attributed graph morphism* $h : \langle AG_1, ag_1\rangle \to \langle AG_2, ag_2\rangle$ is an attributed graph morphism $h = (h_G, h_A)$ which preserves typing, meaning

$$\begin{aligned} ag_{2,G} \circ h_G &= ag_{1,G} \\ ag_{2,A} \circ h_A &= ag_{1,A} \end{aligned}$$

The class of ATG-typed graphs together with the class of ATG-typed graph morphisms defines the category $\mathbf{Graph_{ATG}}$.

Definition (Typed Attributed graph transformation rule): Let ATG be a SIG-attributed type graph and $X = \{X_s | s \in S\}$ be *a sort-indexed family of sets of variables*. An *ATG-typed graph transformation rule over* X, $t = \langle L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R\rangle$, consists of three ATG-typed graphs L, K, R attributed over the term algebra $T_{SIG}(X)$ and two ATG-typed graph morphisms l, r whose homomorphism components, l_A, r_A are identities on $T_{SIG}(X)$. This is summarized by the following equations:

$$\begin{aligned} L &= \langle AG_L,\ ag_L : AG_L \to ATG\rangle \\ K &= \langle AG_K,\ ag_K : AG_K \to ATG\rangle \\ R &= \langle AG_R,\ ag_R : AG_R \to ATG\rangle \\ AG_L &= (G_L, T_{SIG}(X)) \\ AG_K &= (G_K, K_{SIG}(X)) \\ AG_R &= (G_R, T_{SIG}(X)) \\ l &= (l_G : G_K \to G_L\ ,\ l_a : T_{SIG}(X) \to T_{SIG}(X)) \\ r &= (l_R : G_K \to G_R\ ,\ r_a : T_{SIG}(X) \to T_{SIG}(X)) \\ l_a &= r_a = id_{T_{SIG}(X)} \end{aligned}$$

As such the data vertices of the interface and left- and right-hand sides of a rule are actually the elements of $T_{SIG}(X)$. This includes all basic and composite terms over X with and without variables. We say the graph transformation rule t *has variables* if $(IDV_L \cup IDV_R) - T_{SIG} \neq \emptyset$.

When an attributed graph transformation rule is applied according to the double push out approach, the homomorphism components of matchings m, d, m^*, given by

$m_A, d_A, m^*{}_A : T_{SIG}(X) \to A$, are in fact extended assignments and $l^*{}_A, r^*{}_A$ are identities on A (refer to figure 3.10 for double pushout diagram).

Recall from section 3.1.3 that an extended assignment $\overline{\alpha} : T_{SIG}(X) \to A$ uniquely determines a variable assignment $\alpha : X \to A$ and vice versa. Therefore, given a matching m for a rule t, there is only one valid comatch $m^*{}_A$ such that $m^*{}_A \circ u = \alpha = m_A \circ u$. Therefore, to apply a rule t with respect to a matching m one must first extract the assignment α from m_A and then use this assignment to determine the unique comatch m^* for which $\alpha = m^*{}_A \circ u$.

When implementing graph transformation of attributed graphs within a GTT the matchings $m : L \to G$ and $m^* : R \to H$ involved in a derivation $G \Rightarrow_{t,m,m^*} H$ are not explicitly represented. This is because $T_{SIG}(X)$ often has an infinite number of terms in which case an infinite amount of information is required to explicitly represent the SIG-homomorphism components $m_A : T_{SIG}(X) \to A_G$ and $m^*{}_A : T_{SIG}(X) \to A_H$. Fortunately, only the values of attributes are of interest. That is, only the the images of the involved data vertices of L through m_A and R through $m^*{}_A$ are of concern. As a result, only the images of IDV_L and IDV_R through m_A and m^*_A respectively are explicitly specified in the application of a transformation rule.

One issue with this is that the partial specification of the extended assignment m_A does not generally determine a unique variable assignment. As a result, there may be several different possible comatches to chose from (which potentially map different values to IDV_R).

To avoid this added complexity and other non-trivial issues in this work, we impose the following two constraints on the involved data vertices of left- and right-hand sides of typed attributed graph transformation rules:

i. $(\forall d \in IDV_L)\ d \in T_{SIG}(X) - T_{SIG} \implies d \in \bigcup_{s \in S} X_s$, and

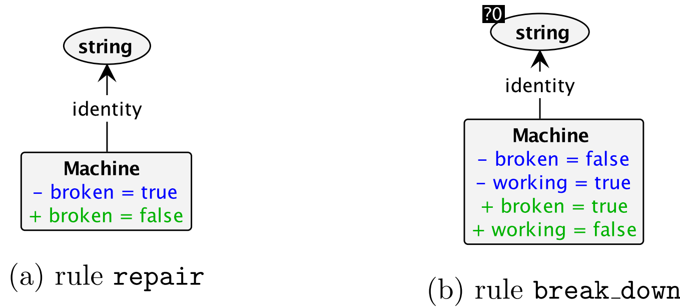
ii. $(\forall d \in IDV_R)(\forall x \in \bigcup_{s \in S} X_s)\ d \in T^x_{SIG}(X) \implies (\exists d' \in IDV_L)\ \ d' = x.$

Intuitively, the first constraint states that the involved data vertices of L may not be compounded terms with variables, that is all variables must appear as basic terms. The first clause simplifies the task of determining whether a matching m is applicable to a rule. To mention one example, this avoids the need to solve a boolean satisfiability problem when attempting to decide the applicability of a match $m : L \to G$ when one of the involved data vertices on the left-hand side is a boolean function (i.e. composite term involving exclusively boolean variables). The second constraint states that if a variable x appears in a term which is an involved data vertex of R, then the variable x must itself be an involved data vertex of L. This clause guarantees only one assignment and in turn one comatch is possible for a given matching.

In figure 3.17 we give two examples of typed attributed graph transformation rules. To easily distinguish attributes with constant values (i.e. terms in T_{SIG}) from those associated to variables

(i.e. terms in $T_{SIG}(X)$), attribute notation is used to represent those with constant values and explicit notation is used for those associated to variables. The same color scheme is used to indicate whether attributes are erasers, readers and writers. In figure 3.17a and 3.17b the attribute `identity` is a reader and the sort of its associated variable (which is `string`) is inscribed within the data node.

Note that since l_A, r_A are required to be identities on $T_{SIG}(X)$, data vertices cannot be deleted by transformation rules. To change the value of an attribute a rule must delete the attribute edge and create a new edge with the same source node pointing to a new data vertex. This is exactly what the rule `repair` does with the attribute `broken` in figure 3.17a.



(a) rule `repair`

(b) rule `break_down`

Figure 3.17: Two typed attributed graph transformation rules

Definition (Typed Attributed graph transformation system): Let ATG be a SIG-attributed type graph and $X = \{X_s | s \in S\}$ be *a sort-indexed family of sets of variables*. An *ATG-typed graph transformation system* $atgts = (SIG, ATG, X, \mathcal{R})$ consists of data type signature SIG, an attributed type graph ATG, a family of variables X and a set of ATG-typed graph transformation rules $\mathcal{R}$ over X.

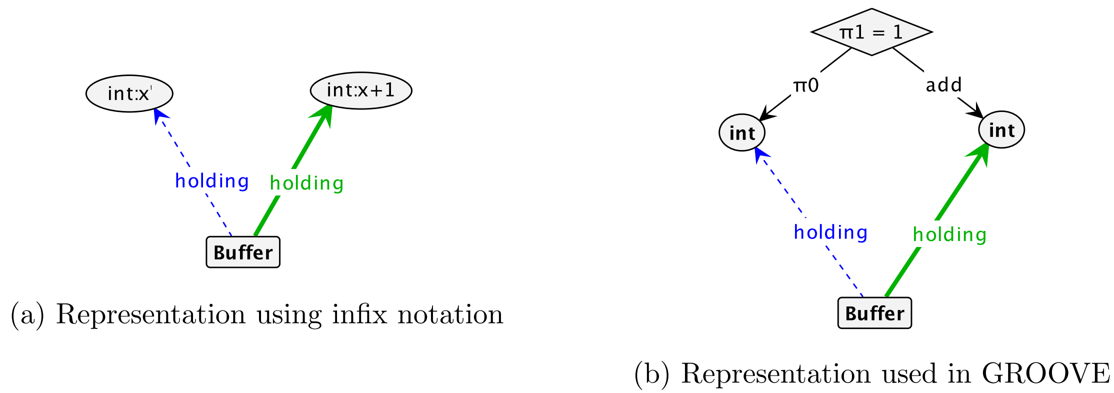
Representation of composite terms in GROOVE

In the exemplary typed attributed transformation rules depicted in figure 3.17a and 3.17b the involved data vertices are both basic terms. We now briefly explain how rules involving composite terms can be represented.

Perhaps the most intuitive way of representing composite terms is by using *infix notation* as illustrated in figure 3.18a. This rule increments the holding attribute of a buffer by one.

GROOVE does not represent composite terms in this manner. Instead it uses so-called *product nodes* to represent operations which are applied to basic terms. In figure 3.18b we illustrate how the exact same rule is constructed within GROOVE.

The product node is drawn as a diamond. The edge labelled π_0 is called an *argument edge*. Argument edges are used to indicate the arguments of operations. In this case, the second



(a) Representation using infix notation

(b) Representation used in GROOVE

Figure 3.18: Representation of composite terms within GTR

argument is inscribed within the product node itself. [8] The edge labelled `add` is a so-called *operator edge*. Its role is to indicate what operation the product node carries out. It is also used to point to the data vertex which represents the result of the operation.

As an example, applying the rule depicted in figure 3.18b to the following graph shown in figure 3.19 would yield the derivation illustrated in figure 3.20.

Figure 3.19: Source graph

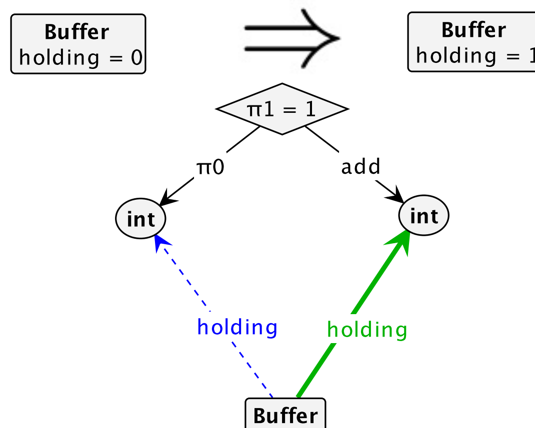


Figure 3.20: Derivation involving rule with a product node

A list of all the operations available within GROOVE can be found in its user manual which can be download from GROOVE's SourceForge Project Page.

[8] When an argument of a product node is a constant, it is sometimes inscribed within the product node itself. This is another one of GROOVE's particular conventions.

3.4.3 Parametrized rules

With the introduction of rules with variables it is possible to model systems with fewer transformation rules. This is because rules with variables are equivalent to a set of rules without variables resulting from the substitution of the variables with the values of the appropriate data type. However, when the induced graph transition system is generated there is an inherent ambiguity associated to transitions corresponding to rules with variables. This is because the values assigned to the variables are not explicitly specified within the graph transition system.

In order to remediate this issue but still retain the advantages provided by rules with variables, the variables of rules may be designated as *parameters*. When a rule with parameters, occurs as a transition the values assigned to its parameters are explicitly specified within the induced graph transition system.

Definition (Parametrized transformation rules): Let ATG be a fixed SIG-attributed type graph and $n \in \mathbb{N}^+$. An *ATG-typed parametrized graph transformation rule*, $p = (\langle L \leftarrow K \rightarrow R \rangle, \{y_1, ..., y_n\})$, consists of an ATG-typed rule span $\langle L \leftarrow K \rightarrow R \rangle$ and a set of *parameters* $\{y_1, ..., y_n\} = Y \subseteq (IDV_L \cup IDV_R) - T_{SIG} \subseteq T_{SIG}(X) - T_{SIG}$. Note that since $Y \subseteq (IDV_L \cup IDV_R) - T_{SIG}$, only rules with variables may be parametrized.

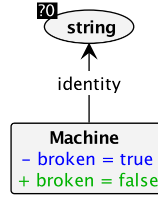


Figure 3.21: Parametrized graph transformation `repair`

In figure 3.21 an example of a parametrized graph transformation rule is given. This rule which is also named `repair` is a parametrized version of the rule shown in figure 3.17a. The black box labelled `?0` hovering over the top left of the data vertex indicates that this string variable is the the rule's first parameter. The nth parameter of a rule is labelled `?(n-1)`. This zero based indexing of parameters is adopted as it is the indexing scheme employed by GROOVE.

Definition (Direct derivation via a parametrized rule): Let $G, H \in \mathcal{G}_{ATG}$, and $p = (\langle L \leftarrow K \rightarrow R \rangle, \{y_1, y_2, ..., y_n\})$ be an ATG-typed parametrized graph transformation rule. In this case, $G \Rightarrow_{p,m,m^*} H$ iff $G \Rightarrow_{p',m,m^*} H$, where $p' = \langle L \leftarrow K \rightarrow R \rangle$ is a non-parametrized ATG-typed transformation rule with the same rule span as p. We call $G \Rightarrow_{p,m,m^*} H$ a *parametrized direct derivation*.

In figure 3.22a we give an example of a parametrized direct derivation via the parametrized rule `repair`. In a parametrized direct derivation, the target graph is derived from the source graph in the same manner as in the non-parametrized case.

Definition (interpretation of a parametrized rule): Let $G, H \in \mathcal{G}_{ATG}$ and $p = (\langle L \leftarrow K \rightarrow R\rangle, \{y_1, ..., y_n\})$ be an ATG-typed parametrized graph transformation rule. Let d be a direct derivation via a parametrized rule p given by $G \Rightarrow_{p,m,m^*} H$. The *interpretation of p with respect to d*, is a (non-parametrized) ATG-typed graph transformation rule given by:

$$p_{z_1,...,z_n} = \langle L' \leftarrow K' \rightarrow R'\rangle$$

where

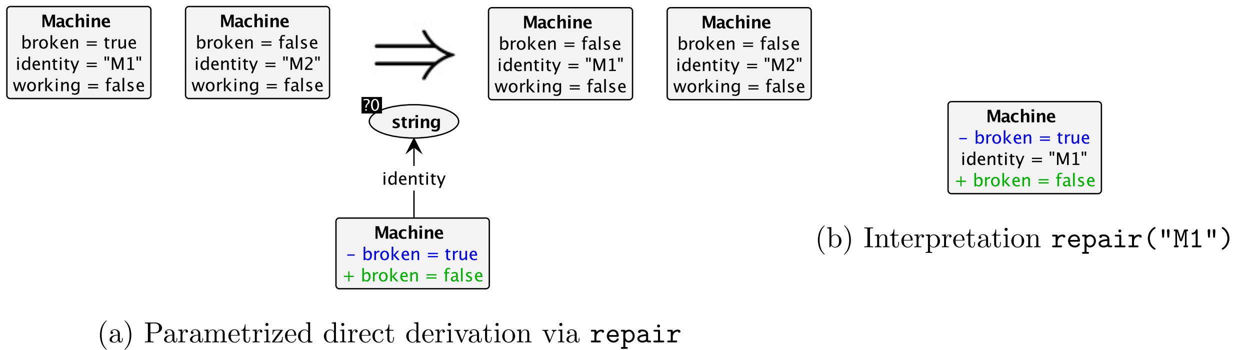
$$\begin{aligned} L' &\cong m(L), \\ K' &\cong m(l(K)) \cong m^*(r(K)), \\ R' &\cong m^*(R), \end{aligned}$$

and

$$z_i = \begin{cases} m_A(y_i) & \text{if } y_i \in IDV_L \\ m^*{}_A(y_i) & \text{if } y_i \in IDV_R \end{cases}$$

In the context of a parametrized direct derivation via a parametrized rule p, the assignment of the values $\{z_1, ..., z_n\}$ to the the parameters of p yields the interpretation $p_{z_1,...,z_n}$. Note that different derivations inducing the same parameter assignment for a given parametrized rule yield the same interpretation. In figure 3.22b we display the interpretation of `repair` with respect to the derivation shown in figure 3.22a.

As the interpretation of a rule is determined by the rule's parameter assignment, we refer to the interpretation via the rule's name followed by a comma separated list of the assigned values. For example, the interpretation depicted in figure 3.22b is referred to as `repair("M1")`.

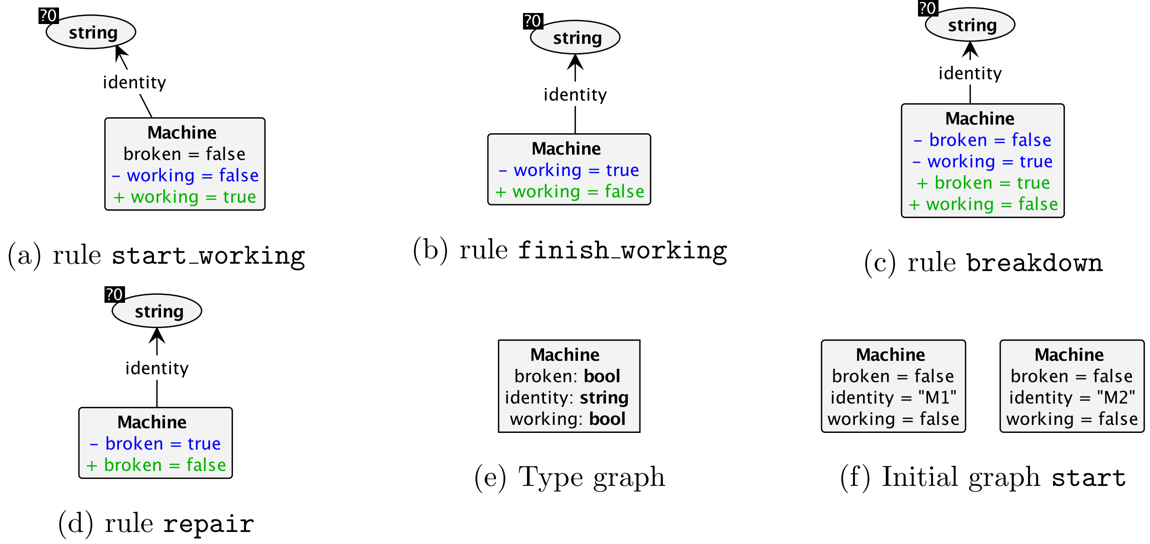


(a) Parametrized direct derivation via `repair`

(b) Interpretation `repair("M1")`

Figure 3.22: A Parametrized direct derivation & the corresponding interpretation

Definition (Parametrized graph transformation system): An ATG-typed *parametrized graph transformation system* (PGTS) $pgts = (SIG, ATG, X, \mathcal{R})$, consists of data type signature SIG, an attributed type graph ATG, a family of variables X and *a mixed set of ATG-typed graph*



(a) rule `start_working` (b) rule `finish_working` (c) rule `breakdown`

(d) rule `repair` (e) Type graph (f) Initial graph `start`

Figure 3.23: A parametrized graph transformation system `Parametrized_Machines` and an initial graph `start`

transformation rules over X, $\mathcal{R} = \mathcal{PR} \sqcup \mathcal{NPR}$ partitioned into parametrized rules $\mathcal{PR}$ and non-parametrized rules $\mathcal{NPR}$.

In figure 3.23 we give an example of a parametrized GTS named `Parametrized_Machines` equipped with an initial graph named `start`. In this example we have:

$$\begin{aligned}
\texttt{Parametrized_Machines} &= (SIG, \texttt{type}, X, \{\texttt{start_working}, \texttt{finish_working}, \texttt{breakdown}, \texttt{repair}\}) \\
\mathcal{PR} &= \{\texttt{start_working}, \texttt{finish_working}, \texttt{breakdown}, \texttt{repair}\} \\
\mathcal{NPR} &= \emptyset
\end{aligned}$$

Let $\mathcal{G}_0$ be a set of start graphs, $pgts = (SIG, ATG, X, \mathcal{R}, ATG)$ be a PGTS and $p \in \mathcal{PR} \subseteq \mathcal{R}$ be one of its parametrized rules. Define the *set of parametrized direct derivations involving* p as $PD(pgts, \mathcal{G}_0, p) = \{G \Rightarrow_{p,m,m^*} H | G, H \in R(pgts, \mathcal{G}_0)\}$. Next define $\mathcal{I}(pgts, \mathcal{G}_0, p)$ as the *set of interpretations of* p *with respect to derivations in* $PD(pgts, \mathcal{G}_0)$. Now we can define the *set of interpretations of* $\mathcal{PR}$ *induced by* $(pgts, \mathcal{G}_0)$ as $\mathcal{I}(pgts, \mathcal{G}_0, \mathcal{PR}) = \bigcup_{p \in \mathcal{PR}} \mathcal{I}(pgts, \mathcal{G}_0, p)$.

Definition (Induced graph transition system of a parametrized graph transformation system): The *graph transition system* $T = (S, \rightarrow, S_0)$ *induced* by an ATG-typed parametrized graph transformation system $pgts = (SIG, ATG, X, \mathcal{R}, ATG)$ equipped with initial graphs $\mathcal{G}_0 \subseteq \mathcal{G}_{ATG}$ consists of a *state set* $S = \{G | G \in R(pgts, \mathcal{G}_0)\}$ of reachable graphs, a *transition relation* $\rightarrow \subseteq S \times [\mathcal{I}(pgts, \mathcal{G}_0, \mathcal{PR}) \cup \mathcal{NPR}] \times \mathcal{MOR}_{\mathbf{Graph_{ATG}}} \times S$, where $(G, t, m, H) \in \rightarrow$ iff $G \Rightarrow_{t,m,m^*} H'$ with $H' \cong H$ and an *initial set* $S_0 = \mathcal{G}_0 \subseteq S$.

Figure 3.24 illustrates the graph transition system induced by (`Parametrized_Machines`, `start`). In this example, the set of interpretations of $\mathcal{PR}$ induced by (`Parametrized_Machines`, `start`) is

given by:

$$\mathcal{I}(\texttt{Parametrized_Machines}, \texttt{start}, \mathcal{PR}) = \{\texttt{start_working("M1")}, \texttt{start_working("M2")}, \texttt{finish_working("M1")}, \texttt{finish_working("M2")}, \texttt{breakdown("M1")}, \texttt{breakdown("M2")}, \texttt{repair("M1")}, \texttt{repair("M2")}\}$$

As Parametrized_Machines possesses only parametrized rules, all of the transitions are interpretations.

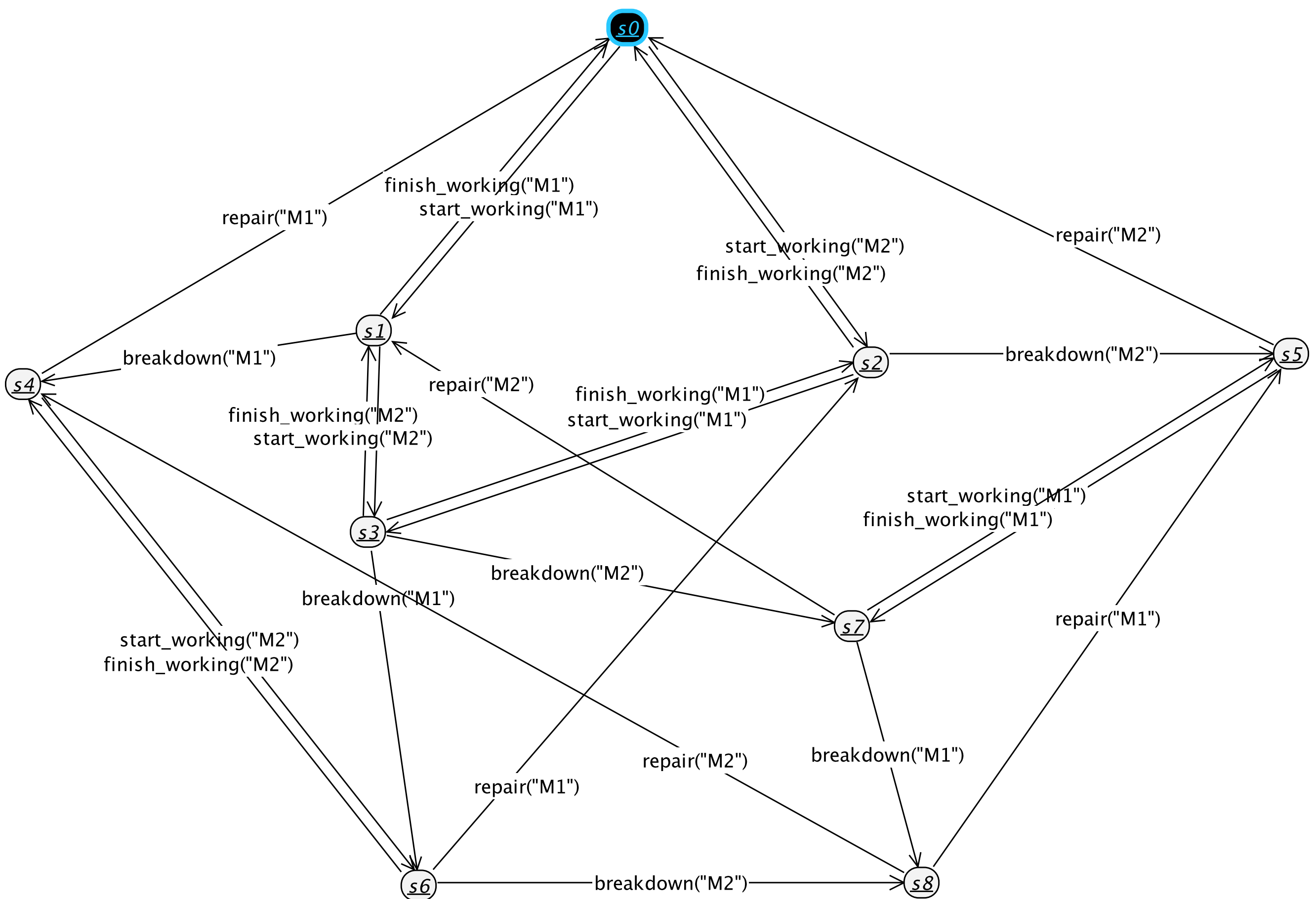


Figure 3.24: A graph transition system induced by a parametrized graph transformation

It is important to note that the parametrization of a graph transformation system's rule set causes no structural changes to its induced graph transition system. Its only purpose is to allow for the customization of transition based on the associated matches and comatches of the derivations underlying each transition.

3.4.4 Graph constraints

In many applications it is useful to determine what subset of graphs produced by a graph transformation system share a certain property. For example, one could be interested in determining the set of graphs which contain a particular subgraph G or the set of graphs that have more than 2 nodes. This can be accomplished using so-called graph constraints. In this work, graph constraints will be used firstly, to specify the subset of states of the induced graph transition system which should be marked as *final*. Secondly, to specify the subset of states which should be flagged as forbidden states in state-based synthesis.

The degree of generality with which this concept is defined determines its ability to express properties concisely. The definition one chooses to employ ultimately depends on the needs imposed by the application at hand. In this work we present perhaps the simplest definition of graph constraints. More complex and powerful formulations of graph constraints can be found in [10, 24].

Definition (Graph constraint): A *positive (negative) atomic graph constraint* is of the form $PC(C)$ $(NC(C))$ where C is some graph. A graph G *satisfies* the positive (negative) atomic graph constraint $PC(C)$ $(NC(C))$, written $G \models PC(C)$ $(NC(C))$, iff there exists a (total) morphisms $m : C \to G$. Note that m need not be injective.

Atomic graph constraints may be combined via logical connectives to form *graph constraints*. As such, every atomic graph constraint is a graph constraint and for every graph constraint c, $\neg c$ is a graph constraint and for every index set I and family $(c_i)_{i \in I}$ of graph constraints, $\wedge_{i \in I} c_i$ and $\vee_{i \in I} c_i$ are graph constraints. A graph G satisfies a graph constraint $\neg c$, written $G \models \neg c$ iff G does not satisfy c, written $G \not\models c$. G satisfies $\wedge_{i \in I} c_i$ $(\vee_{i \in I} c_i)$, written $G \models \wedge_{i \in I} c_i (\vee_{i \in I} c_i)$, iff all (some) c_i with $i \in I$ are satisfied by $\langle AG, ag \rangle$.

Note that the positive (negative) atomic graph constraint characterized by the graph of order zero, $PC(K_0)$ $(NC(K_0))$ is satisfied by all (no) graphs G because a total empty morphism always exists from K_0 to G.

For the sake of generality, the above definition of graph constraints does not reference any overarching category. In practice however, the graph C characterizing a positive or negative atomic graph constraint is an object from a fixed category $\mathcal{C}$ such as $\mathbf{Graph}_\Lambda$, $\mathbf{Graph_{TG}}$ or $\mathbf{Graph_{ATG}}$. In this context, a positive (negative) atomic graph constraint can be viewed as a (characteristic) function $PC(C)$ $(NC(C)) : Ob_\mathcal{C} \to \{true, false\}$ defined by

$$PC(C)(G) = true \text{ iff } \mathcal{C}[C, G] \neq \emptyset$$
$$NC(C)(G) = true \text{ iff } \mathcal{C}[C, G] = \emptyset$$

Recall from section 3.1.2 that $\mathcal{C}[C, G]$ is the set of all morphisms in $Mor_\mathcal{C}$ with domain and codomain C and G respectively.

This functional interpretation of atomic graph constraints can straightforwardly be generalized for non-atomic graph constraints.

In figure 3.25 an example of a graph constraint over the category $\mathbf{Graph_{ATG}}$ is given. This constraint consists of the logical conjunction of two atomic negative graph constraints. Intuitively, a graph G (which is ATG-typed in this case) satisfies this constraint as long as it contains neither:

i. a node of type `Machine` with a `broken` attribute set to `true`

ii. nor a node of type `Machine` with a `working` attribute set to `true`.

$$NC\left(\begin{array}{|c|}\hline \textbf{Machine} \\ \text{broken = true} \\ \hline\end{array}\right) \wedge NC\left(\begin{array}{|c|}\hline \textbf{Machine} \\ \text{working = true} \\ \hline\end{array}\right)$$

Figure 3.25: A typed and attributed graph constraint

Chapter 4

Graph rewriting modelling framework

Having presented the fundamental concepts underlying the algebraic approach as well as some key extensions to graph rewriting, this chapter now presents the graph-based modelling framework which will be used to generate the finite state automaton models used within SCT. We begin by defining the graph-based models about which our framework is centred in section 4.1. In section 4.2, we then provide an overview of how these graph-based models are used to obtain finite state automata. In section 4.3 we discuss how they are used to model systems. Finally in section 4.4 we explain how the proposed modelling framework is applied to generate the models required in both event- and state-based supervisory controller synthesis.

4.1 Implicit and induced automata

Our proposed modelling framework revolves around two mathematical models referred to as the *implicit automaton* and the *induced automaton*. Both of these structures are defined in terms of GTS. The definition of these models may be adjusted to accommodate GTS which transform graphs in different categories. To introduce advanced functionalities, we chose to define these models in terms of typed attributed and parametrized graph transformation systems. These GTS manipulate graphs in the category $\mathbf{Graph}_{ATG}$.

Definition (Implicit automaton): An *implicit automaton* typed over ATG is a triple $ia = (pgts, \mathcal{G}_0, c_m)$ consisting of an ATG-typed PGTS $pgts = (SIG, ATG, X, \mathcal{R})$, a set of initial graphs $\mathcal{G}_0 \subseteq \mathcal{G}_{ATG}$ and a special graph constraint c_m over $\mathcal{G}_{ATG}$ referred to as a *marking constraint*.

Figure 4.1 provides an examples of an implicit automaton. Figures 4.1a-4.1d depict the graph transformation rules, figure 4.1e shows its type graph, figure 4.1g illustrates its initial graph and figure 4.1f displays its marking constraint.

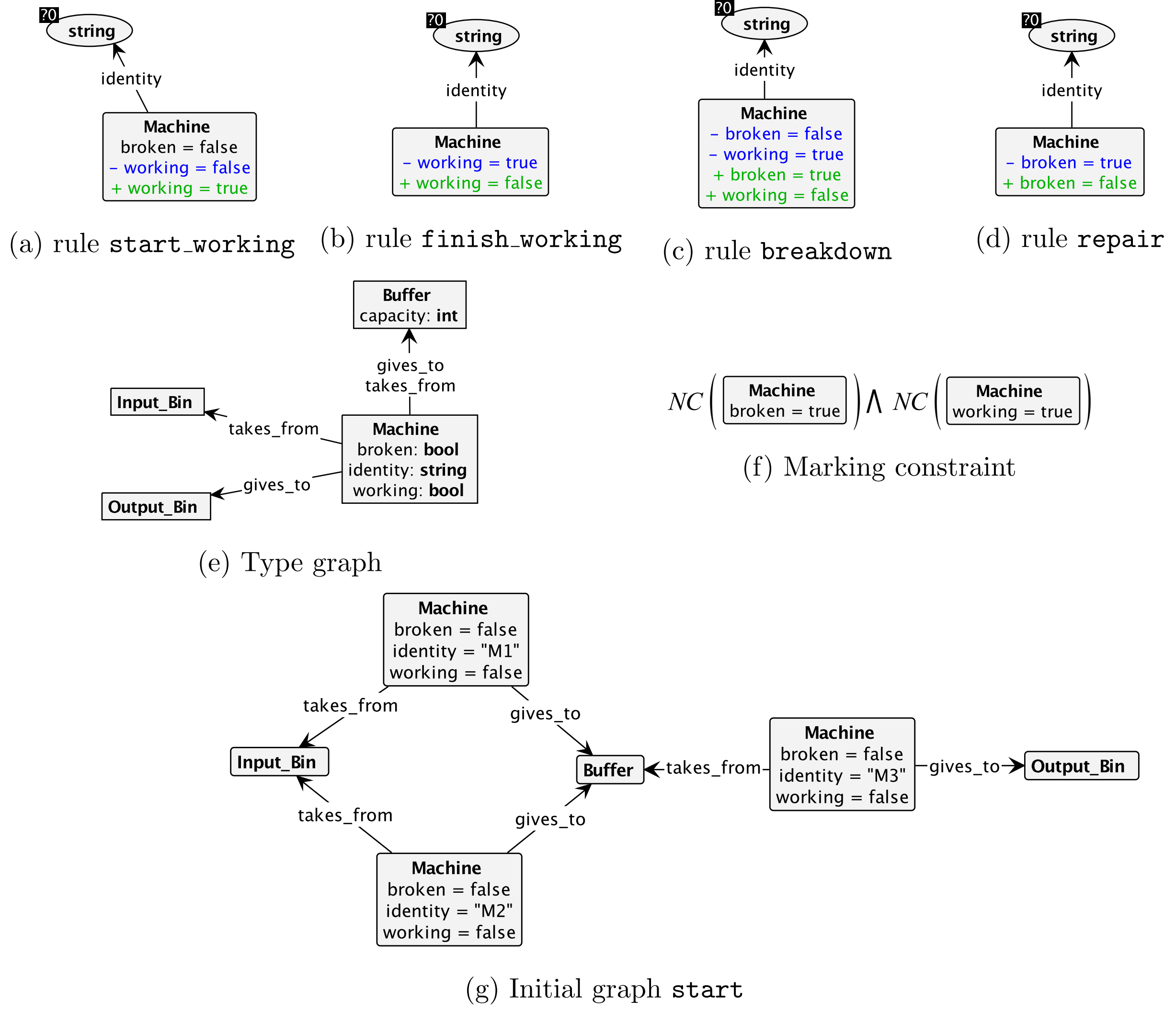


Figure 4.1: An implicit automaton

The reasoning behind the terminology *implicit automaton* stems from the fact that although the triple $ia = (pgts, \mathcal{G}_0, c_m)$ does not explicitly specify the state, transition and marking structure of an automaton, it contains all the information required to construct one. This is demonstrated by the definition of the *induced automaton* which follows.

Definition (Induced automaton): Let $ia = (pgts, \mathcal{G}_0, c_m)$ be an ATG-typed implicit automaton with $pgts = (SIG, ATG, X, \mathcal{R})$. The *automaton* induced by ia, $\mathbf{A} = (Q, \Sigma, \Delta, Q_0, Q_m)$, consists of a *set of states* Q, *event set* Σ, *transition relation* $\Delta \subseteq Q \times \Sigma \times Q$, *initial states* Q_0 and *marked*

states Q_m where

$$\begin{aligned} Q &= \{q|\ q \in R(pgts, \mathcal{G}_0)\}, \\ \Sigma &= \mathcal{I}(pgts, \mathcal{G}_0, \mathcal{PR}) \cup \mathcal{NPR}, \\ Q_0 &= \{q \in Q|\ q \in \mathcal{G}_0\}, \\ Q_m &= \{q \in Q|\ q \models c_m\}, \text{ and} \\ (\forall q, q' \in Q)(\forall \sigma \in \Sigma)(q, \sigma, q') \in \Delta &\iff (\exists m \in \mathcal{MOR}_{\mathbf{Graph}_{ATG}}) q \Rightarrow_{\sigma,m} q'. \end{aligned}$$

Once the induced automaton is generated, the event set may be partitioned as $\Sigma = \Sigma_c \sqcup \Sigma_u$, i.e. into controllable and uncontrollable event sets. This partition of Σ is determined based on the informal description of the system. Note that the set of interpretations of a parametrized rule need not be exclusively controllable or uncontrollable. In other words, given a parametrized rule $p \in \mathcal{PR}$, one may have

$$\mathcal{I}(pgts, \mathcal{G}_0, p) \cap \Sigma_c \neq \emptyset \qquad \& \qquad \mathcal{I}(pgts, \mathcal{G}_0, p) \cap \Sigma_u \neq \emptyset$$

Definition (Partitionability of rule set): Let $pgts = (SIG, ATG, X, \mathcal{R})$ be a PGTS. Let $\mathbf{A} = (Q, \Sigma, \Delta, Q_0, Q_m)$ be the automaton induced by the implicit automaton $ia = (pgts, \mathcal{G}_0, c_m)$ and $\Sigma = \Sigma_c \sqcup \Sigma_u$ be a particular partition of its event set. The parametrized rule set $\mathcal{R}$ is *partitionable* with respect to the partition $\Sigma = \Sigma_c \sqcup \Sigma_u$ if and only if

$$(\forall p \in \mathcal{PR}) \ \ [\mathcal{I}(pgts, \mathcal{G}_0, p) \subseteq \Sigma_c \ \vee \ \mathcal{I}(pgts, \mathcal{G}_0, p) \subseteq \Sigma_u]$$

When $\mathcal{R}$ is partitionable there exists a partition $\mathcal{R} = \mathcal{R}_c \sqcup \mathcal{R}_u$ such that $\Sigma_c = \mathcal{R}_c$ and $\Sigma_u = \mathcal{R}_u$.

Intuitively, a parametrized rule set is partitionable into controllable and uncontrollable events, if each of its parametrized rules are controllable or uncontrollable independent of interpretation, i.e. for each parametrized rule, all of its interpretations are either controllable or uncontrollable.

The state, transition and marking structure of the induced automaton is isomorphic to that of the abstract automata defined in conventional automata theory. However, the induced automaton we have defined is more concrete as its states and transitions are graphs and graph transformation rules respectively.

As with conventional automata, the induced automaton may generally have several initial states, possess an infinite state set and be non-deterministic. The induced automaton $\mathbf{A} = (Q, \Sigma, \Delta, Q_0, Q_m)$ is a *deterministic finite-state automaton* (DFA) if

i. $|Q| \in \mathbb{N}$ (finite-state),

ii. $|Q_0| = 1$ (single initial state), and

iii. $(\forall q \in Q)(\forall \sigma \in \Sigma)[(\exists q' \in Q)(q, \sigma, q') \in \Delta \implies (\exists! q' \in Q)(q, \sigma, q') \in \Delta]$ (deterministic).

As such, an implicit automaton $ia = (pgts, \mathcal{G}_0, c_m)$ is called *DFA-inducing* if the automaton it induces $\mathbf{A} = (Q, \Sigma, \Delta, Q_0, Q_m)$, is a DFA. ia is DFA-inducing precisely when

i. $|R(pgts, \mathcal{G}_0)| \in \mathbb{N}$,

ii. $|\mathcal{G}_0| = 1$, and

iii. $(\forall G, G' \in R(pgts, \mathcal{G}_0))(\forall \sigma \in \Sigma)$
$[(\exists m \in \mathcal{MOR}_{\mathbf{Graph}_{ATG}})\ G \Rightarrow_{\sigma,m} G' \implies (\exists! m \in \mathcal{MOR}_{\mathbf{Graph}_{ATG}}) G \Rightarrow_{\sigma,m} G']$.

Intuitively, clause iii is satisfied when each event is involved in at most one derivation from each state. Note that here, an event $\sigma \in \Sigma = \mathcal{I}(pgts, \mathcal{G}_0, \mathcal{PR}) \cup \mathcal{NPR}$ is either a non-parametrized graph transformation rule or the interpretation of a parametrized transformation rule (which is itself a rule). As such, a parametrized transformation rule may be involved in several derivations from a given state as long as each of its interpretations in $\mathcal{I}(pgts, \mathcal{G}_0, \mathcal{PR})$ are involved in at most one derivation from that state. Therefore, parametrization of rules can eliminate non-determinism in the induced automaton.

4.2 Generating deterministic finite-state automata

The main objective of our modelling framework is to produce DFA models of systems for use by the SCT framework. In this section we provide a high-level description of the procedure used to obtain a DFA model starting from an informal description of the system.

The first step is translating the informal description into an implicit automaton model. The implicit automaton is designed based on the modeller's intuitive understanding of the informal description. The modeller manually creates and edits the implicit automaton model using the graphical user interface of a graph transformation tool.

Once its design is complete, the implicit automaton is used by the GTT to generate the corresponding induced automaton. These automaton models can then be exported from the GTT and submitted to SCT synthesis tools to synthesize supervisory controllers. Obviously, the files exported by the GTT may need to be converted to the file format supported by the employed SCT tool.

Depending on the implicit automaton model received as input, the resulting induced automaton may have an infinite state space. When this is the case, the GTT will continue to compute the

states and transitions of the induced automaton ad infinitum. It may also be the case that the generated automaton is non-deterministic. In this particular application, where the goal is to produce DFA models, both these scenarios indicate there are errors in the design of the implicit automaton. As such, one objective when designing implicit automata is ensuring they are DFA-inducing.

The process describing how deterministic finite-state automata are generated is summarized by the process diagram shown in figure 4.2.

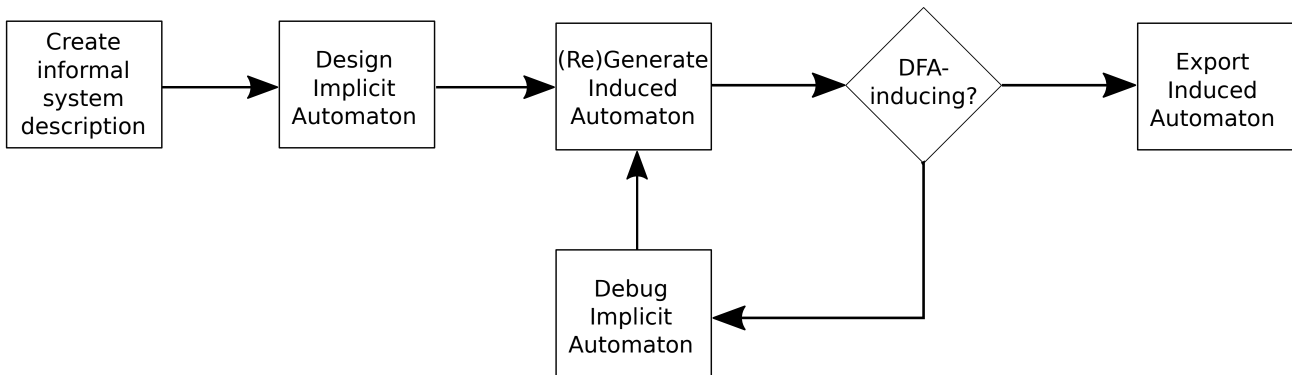


Figure 4.2: Generating DFA via implicit automata

4.3 Modelling systems with implicit automata

In this section we focus on a particular step involved in the generation of DFA models, namely the design of the implicit automaton. There is no systematic procedure which can be followed to translate a system's informal description into an implicit automaton. The design of the implicit automaton is therefore a creative process which relies upon the modeller's ability to intuitively interpret the informal system description. We begin by discussing how attributed graphs should be used to model the state of a system. We then discuss more generally how entire systems are modelled with implicit automata by implicitly adopting so called *modelling conventions*. We then elaborate on the role played by each of the implicit automaton's components.

4.3.1 Best modelling practices for attributed graphs

A typed attributed graph consists of *object vertices*, *data vertices*, *links* and *attributes* (see section 3.4.2). Object vertices are used to represent the actual entities which comprise the system (at that moment). Links, which are simply edges connecting two object nodes, are used to represent relationships between two entities. Data vertices on the other hand, are used to express information about system entities. Attributes, which are edges directed from object vertices to

data vertices, associate information expressed by data vertices to specific system entities. In addition, they also interpret the manner in which the associated information characterizes the entity.

In physical systems, object vertices can be used to represent *physical components* and links can be used to represent relationships among these physical components. For instance, in the machine and buffer examples presented throughout chapter 3, object vertices were used to represent machines, buffers and workpieces. However, object vertices are not limited to representing physical components. They may also represent *virtual components* such as clocks which track system time, or memories which track system history.

Before the introduction of attributed graphs, we used normal vertices and edges to associate information to system entities. For example, in figure 3.12e a `Machine` vertex is connected to an `Idle` vertex via an `is` edge to represent the fact that the machine is idle. Although this is not strictly forbidden, it is considered bad practice to use object vertices and links in this manner. Instead, data vertices and attributes should be used to represent information related to system entities. An example of proper object metadata representation is illustrated in figure 3.23d, where two `Machine` (object) vertices each have `broken` attributes with the values false to represent the fact that both machines are not broken.

4.3.2 Modelling conventions

When modelling systems via implicit automata, the modeller implicitly creates *modelling conventions* which guide him throughout the design. In this work, a *modelling convention* refers to the designation of graph theoretical elements (such as vertices and or edges) to represent an aspect of a system's state. Listed below, are some of the modelling conventions implicitly relied upon to design the implicit automaton illustrated in figure 4.1.

i. `Machine` vertex represents a machine in the system.

ii. `Input_Bin` vertex represents the source of workpieces.

iii. `Output_Bin` vertex represents the destination of workpieces.

iv. `Buffer` vertex represents a buffer in the system.

v. `gives_to` and `takes_from` edges indicate the direction in which a workpiece is transferred.

vi. `Machine`'s `working` attribute of type `bool` keeps track of whether a machine is working on a workpiece or if it is idle.

vii. `Machine`'s `identity` attribute of type `string` distinguishes different machines in the system.

viii. `Machine`'s `broken` attribute of type `bool` keeps track of whether a machine is broken or not.

Ideally, all modelling conventions should be defined in accordance with best modelling practices discussed in section 4.3.1.

There are often several modelling conventions which may be used to model the exact same aspect of a system's state. For example, in the GTS shown in figure 3.12, the presence of an `Idle` vertex incident to a `Machine` vertex is used to indicate the machine is idle. However, in the implicit automaton example just discussed, a machine is understood to be idle when its `working` attribute has the value `false`.

4.3.3 Creating implicit automata models

Formally, an implicit automaton consists of a pgts, a set of initial graphs and a marking constraint. The pgts itself comprises a data type signature, a type graph, a set of parametrized graph transformation rules and a sort-indexed set of variables. In practice, when the modeller creates an implicit automaton he must only produce a type graph, a set of initial graphs, a set of graph transformation rules and a marking constraint. This is because the data type signature is technically defined by the GTT which provides a pre-established set of supported data types and operations. When an implicit automaton is created, the modeller employs the data types and operations necessary for that specific model. As for the sort-indexed set of variables, it is implicitly defined via the creation of the graph transformation rules.

Type graph

The first component of the implicit automaton which is created is the type graph. The type graph is essentially a metamodel the modeller creates prior to modelling the actual system which stipulates the types of entities and relationships that can be present within the system's state. That is, it ensures that all graphs derived by the implicit automaton comprise only vertex and edge types approved by the type graph. It accomplishes this by requiring initial graphs, transformation rules and the marking constraint be properly typed.[1] When the modeller creates the remaining components of the implicit automaton, i.e. initial graphs, transformation rules and marking constraint, the GTT verifies if they are properly typed and flags them as erroneous if they are not. The GTT prevents the user from generating the induced automaton, if the implicit automaton has any errors. Overall, type graphs make modelling less error-prone by informing the modeller if he has included vertex or edge types in an initial graph, transformation rules or marking constraint which he has not predefined in the type graph.

[1] Although rules deleting and or reading illegal vertex and or edge types cannot produce illegal graphs (if the source graph is legal), they are never applicable and therefore prohibited.

Initial graph

The initial graphs should be created second. They are primarily used to represent the initial states of the system. When the objective is to generate a DFA, only one initial graph should be specified. However, the specification of several initial graphs is permitted to make possible the generation of state machines with several initial states such as *holons*. Initial graphs may also be used to demonstrate a system's architecture. For example, the initial graph in figure 4.1g illustrates how a system comprising three machines, an input bin, a buffer and an output bin is structured.

Graph transformation rules

After the initial graphs are complete, graph transformation rules are created to express the system's dynamics. Intuitively, they describe the different kinds of events which may occur in the system. As the implicit automata only explicitly specifies the initial states, the transformation rules together with the initial graphs are used to determine the state and transition structure of the induced automaton. The transformation rules are also used to define the event set of the induced automaton. However, there is generally not a one-to-one correspondence between transformation rules and events due to rule parametrization. Graph transformation rules can generally be used to represent a group of similar events sharing some commonality. For example, the rule depicted in figure 4.1d, represents the generic event of a machine being repaired but it can be interpreted as the specific event of repairing machine `"M1"`, `"M2"` or `"M3"`. This type of versatility exhibited by transformation rules is possible due to the data variables introduced in section 3.4.2. When ever possible, variables should be used to minimize the number of rules created to describe a system's dynamics. This results in a more compact implicit automaton model.

Although rules with variables can model events which are more generic, non-parametrized rules with variables are also more ambiguous. This is because when a non-parametrized rule with variables is applied as a transition in the induced automaton, the values assigned to its variables are not specified. The application of this type of rule is therefore more ambiguous. To illustrate this point more concretely, consider the non-parametrized version of the implicit automaton displayed in figure 4.1, i.e. the implicit automaton obtained by unparametrizing all of its rules. The automaton induced by this hypothetical implicit automaton would be defined over the event set

$$\Sigma = \{\texttt{start_working},\ \texttt{finish_working},\ \texttt{breakdown},\ \texttt{repair}\}.$$

As such, transitions representing the occurrence of one event with respect to different machines are no longer differentiated. For example, the `start_working` event indicates that either Machine M1, M2 or M3 has started working.

Perhaps this methodology of creating ambiguous system models may prove to be useful in supervisory control under partial observation. However, in applications in which this type of ambiguity is inappropriate it can always be removed by designating variables in a rule as parameters(as shown in section 3.4.3). As such, the modeller can use rule parametrization to take advantage of the compactness benefits provided by variables without introducing ambiguity in the resulting automaton.

Marking constraint

The marking constraint is created last. It is used to concisely express which states are marked in the induced automaton without having to explicitly specify a subset of its state set. When the marking constraint is complex, the modeller can create it in a modular fashion by first creating atomic graph constraints and then combining them via logical connectives.

4.4 Application to supervisory control

Heretofore, this chapter has simply described how graph rewriting can be leveraged to generate DFA models. In this section, it is discussed how this modelling framework is used to generate the models used as input in the standard SCT synthesis algorithms described in section 2.8. In particular, in section 4.4.1 it is discussed how the proposed modelling framework should be used with event-based SCT. Finally, in section 4.4.2 it is discussed how to do so with the state-based approach.

4.4.1 Event-based synthesis

As mentioned in chapter 2 both the plant and specification are practically represented as automata in event-based SCT. As a result, the modeller creates an implicit automaton for both the plant and specification to generate their induced automaton models. Recall that the standard SCT synthesis algorithms receive as input deterministic finite state automata, hence the modeller must ensure the implicit automata are DFA-inducing as discussed in section 4.2. Once the induced automata are generated the modeller partitions their events sets into controllable and uncontrollable subsets as necessary.

To illustrate how the implicit plant and specification automata are obtained from the system's informal description we use the "Big Factory" example presented in chapter 4 of [29]. To this end, we first provide the informal description of the Big Factory and its specification.

Description of Big Factory:

The Big Factory consists of three machines `M1`, `M2` and `M3` and a single buffer. The input and output relationships of the machines and buffer are shown in figure 4.3.

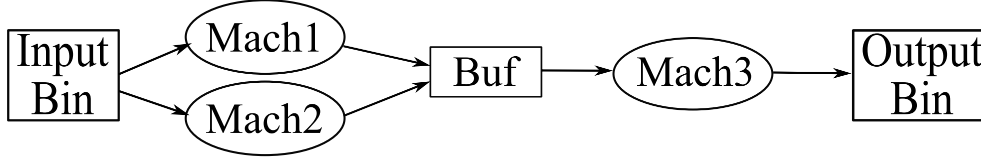


Figure 4.3: Architecture of Big Factory

A machine can take a workpiece from its corresponding input and begin work on it. Once it has begun work a machine can transfer its workpiece to its corresponding output or it can break down, in which case it must wait to be repaired before it can begin work on a new workpiece. Machine `M1` and `M2` take their workpieces from an infinite input bin and machine `M3` gives its workpiece to an infinite output bin. A machine must be authorized by the Big Factory's control system before it can begin working on a new workpiece. Once a machine has begun working the system has no control over when the machine finishes working.

Big Factory Specification:

i. The buffer has a capacity of two workpieces and must not overflow or underflow.

ii. Machine `M3` has priority of repair over machines `M1` and `M2`.

The implicit automaton of the plant was provided earlier in chapter 4, in figure 4.1. The induced automaton generated by this implicit automaton is equivalent to the shuffle of 3 machine automata. It has 27 states and 109 transitions. In this example, all four transformation rules are parametrized and each rule has three interpretations, one for each machine. Hence there are 12 different events in the induced automaton of the Big Factory. In this case, the rule set is partitionable into controllable and uncontrollable event sets as each rule is either controllable or uncontrollable regardless of what machine it is applied to. The partition of $\mathcal{R}$ is given by

$$\mathcal{R}_c = \{\texttt{start_working}, \texttt{repair}\}, \text{ and}$$
$$\mathcal{R}_u = \{\texttt{finish_working}, \texttt{breakdown}\}.$$

Σ is given by

$$\Sigma_c = \{\texttt{start_working("M1")}, \texttt{start_working("M2")}, \texttt{start_working("M3")},$$
$$\texttt{repair("M1")}, \texttt{repair("M2")}, \texttt{repair("M3")}\}, \text{ and}$$
$$\Sigma_u = \{\texttt{finish_working("M1")}, \texttt{finish_working("M2")}, \texttt{finish_working("M3")},$$
$$\texttt{breakdown("M1")}, \texttt{breakdown("M2")}, \texttt{breakdown("M3")}\}.$$

We now turn our attention to the specification. It is possible to create one implicit automaton model which accounts for the entire specification. However, it is generally undesirable to do so as the resulting model will typically be overwhelmingly complex and difficult to interpret. Furthermore, the automaton induced by this monolithic implicit specification automaton may take the GTT too long to generate. Instead, the specification automaton may be obtained in a modular fashion as follows[2]:

1. Design a separate implicit automaton for each requirement listed in the specification.
2. Generate the induced automaton of each implicit automaton using a GTT.
3. Compute a minimal state realization of each generated automaton.
4. Self-loop each induced automaton with its associated complementary event set.
5. Compute the product of the constituent automata to obtain the complete specification automaton.

To demonstate this modular approach an implicit automaton is designed for each of the Big Factory's sub-specifications. The implicit automaton for sub-specification i is illustrated in figure 4.4.

Intuitively, the implicit automaton of a (sub)specification models relevant aspects of the plant and introduces additional memory elements when necessary. In this example, the initial graph only includes elements of the model which pertain to the buffer specification. The attribute `holding` of type `int`, is a memory element introduced to keep track of the number of workpieces the buffer is holding and detect when underflow and overflow occurs. Furthermore, only rules relevant to the sub-specification are included, namely `start_working` and `finish_working`.

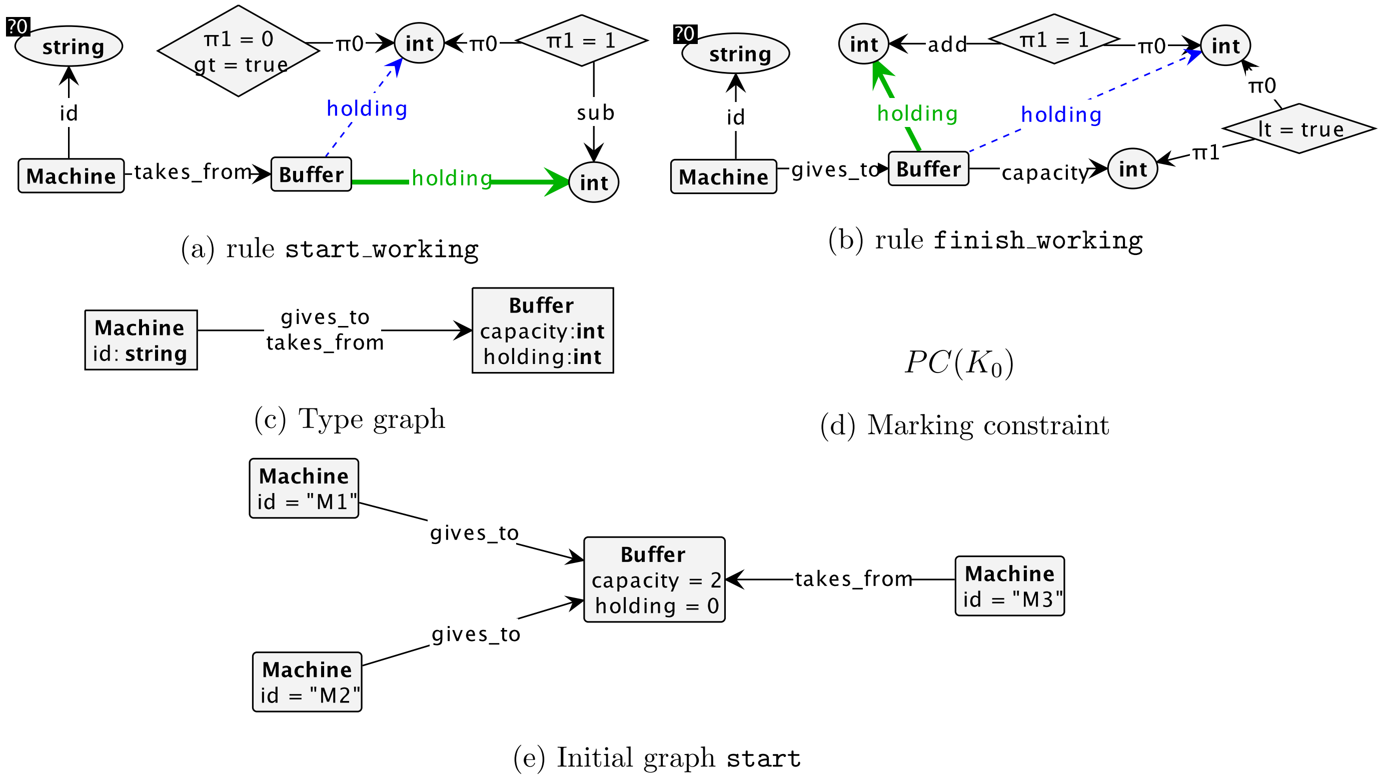
Note that rules in an implicit plant and (sub)specification automaton, which are intended to represent the same action must be given the same rule name. However, they may possess different rule spans as is the case for rules `start_working` and `finish_working`.

In the context of buffer specification, the rule `start_working` can decrement the `holding` attribute of the buffer as long as `holding` > 0. Conversely, `finish_working` increments `holding` as long as `holding` $<$ `capacity`.

The marking constraint is chosen to be $PC(K_0)$ so that every state in the induced automaton of the buffer specification is marked. This allows the plant to determine the marking.

The automaton induced by the implicit automaton of the buffer specification is depicted in figure 4.5.

[2]This approach is applicable so long as the specification can be decoupled into independent sub-requirements.



(a) rule `start_working`

(b) rule `finish_working`

(c) Type graph

(d) Marking constraint

(e) Initial graph `start`

Figure 4.4: Implicit automaton of buffer specification

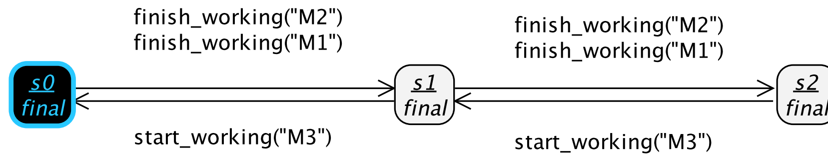


Figure 4.5: Induced automaton of buffer specification

Shifting our focus to sub-specification ii, we depict its implicit automaton model in figure 4.6. In this example a `priority` attribute of type `int` is introduced to keep track of repair priority. The underlying modelling convention being that machines with greater priority attribute values are repaired first and machines with equal values can be repaired in any order. In accordance with this convention, the rule `repair` toggles a machine's `broken` attribute from `true` to `false` if its priority attribute value is greater than or equal to every other broken machine's priority value. Note that `repair` uses an advanced feature of GROOVE known as *nested rules* which is not explained within this thesis. Readers interested in advanced features such as this, are referred to GROOVE's user manual which can be found at GROOVE's SourceForge Project Page.

The priority attribute values of machines 1,2 and 3 are set to `1`,`1` and `2` respectively to give machine 3 repair priority over machines 1 and 2 and give machines 1 and 2 equal priority (as per the specification).

The automaton induced by the implicit automaton of the repair priority specification is illustrated

(a) rule `breakdown`

(b) rule `repair`

(c) Type graph

$$PC(K_0)$$

(d) Marking constraint

(e) Initial graph `start`

Figure 4.6: Implicit automaton of buffer specification

in figure 4.7.

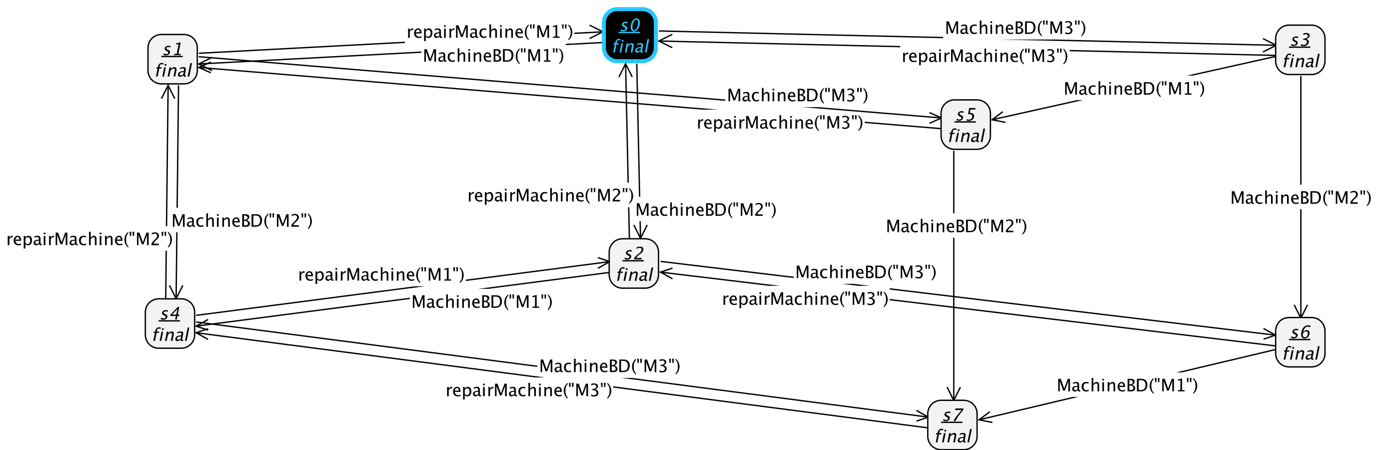


Figure 4.7: Induced automaton of Repair priority specification

4.4.2 State-based synthesis

As is the case in the event-based approach, the plant is also represented as an automaton in state-based SCT (see section 2.9). As such, when the state-based approach is employed the plant is once again modelled as an implicit automaton. The specification however, is no longer represented as an automaton. Instead, it is expressed in terms of mutual state and state-event exclusions. In this subsection it is illustrated how mutual state exclusion specifications are represented using the tools available within the proposed modelling framework. The representation of state-event exclusions will be left as future work.

Mutual state exclusions which are essentially state predicates can be represented using graph constraints. This is formalized in the definition which follows.

Definition (State predicate induced by a graph constraint): Let $A = (Q, \Sigma, \Delta, Q_0, Q_m)$ be the automaton induced by an implicit automaton $ia = (gts, \mathcal{G}_0, c_m)$ and let c be a graph constraint. The state predicate P_c for A, induced by graph constraint c is defined by

$$(\forall q \in Q)\ q \models P_c \Longleftrightarrow q \models c$$

To identify those graph constraints used to model mutual state exclusions we refer to them as *forbidden state constraints*.

To demonstrate how forbidden state constraints can be used to model mutual state exclusion specifications we use the toy "Guideway" example taken from chapter 6 of [29]. To this end, let us provide a brief informal description of the Guideway and its specification.

Description of Guideway:
Two stations `A` and `B` are situated along a guideway. Two vehicles `V1` and `V2` travel along a one-way track from `A` to `B`. The track is separated into two sections with stoplights($*$) installed at a subset of the section junctions as illustrated in figure 4.8.

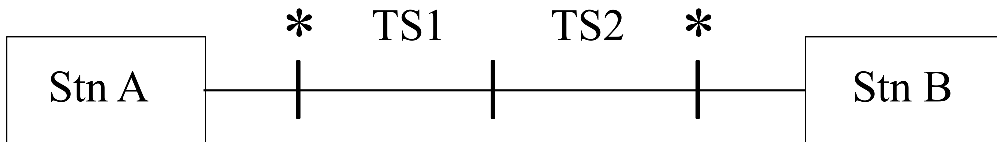


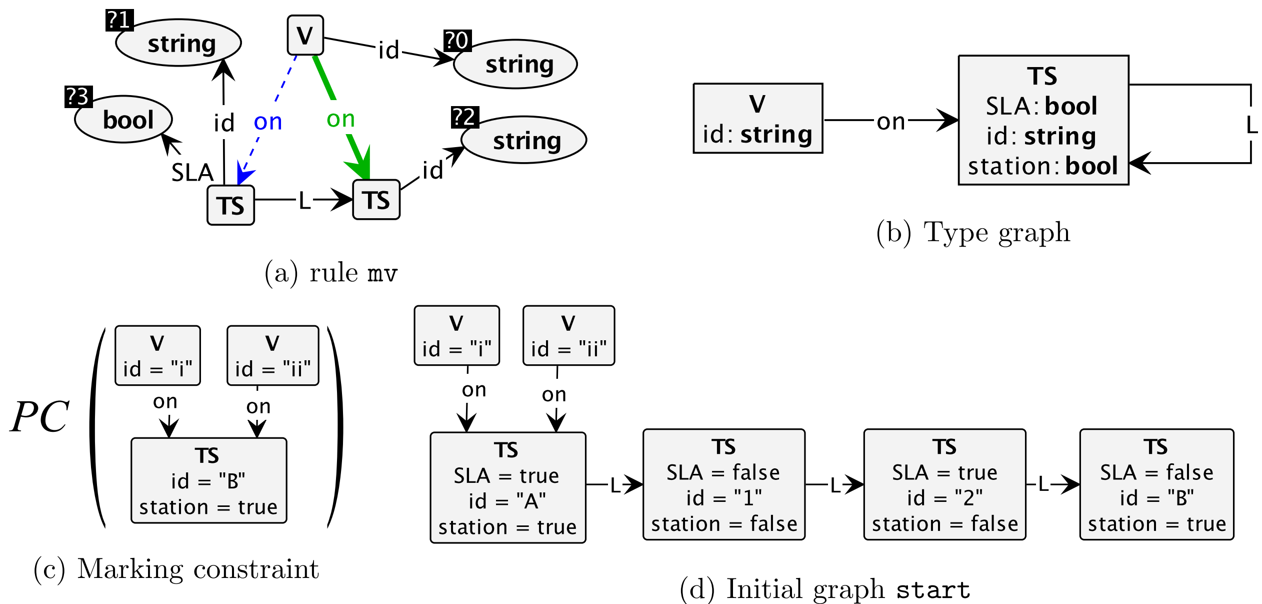
Figure 4.8: Topology of track

Both vehicles begin at station `A` and travel along the track to reach station `B`. The guideway control system may only prevent a vehicle from crossing a section junction at which a stoplight is installed. Detectors are installed at section junctions to enable the guideway's control system to detect when a vehicle enters a new track section. This allows it to maintain what track each

vehicle occupies. In this instance of the Guideway example, it is assumed that detectors are installed at every junction.

Specification:
The guideway's control system should operate the stoplights to prevent both vehicles from travelling along the same section of track simultaneously.

An implicit automaton representing the plant of the Guideway is depicted in figure 4.9. The initial graph depicted in figure 4.9d, encodes the topology of the track and initial positions of the vehicles. The track sections are represented by `TS` nodes. The `L` edges joining the `TS` nodes express how the track sections are linked as well as the direction of travel. To minimize the number of rules needed to model dynamics, the stations are also represented by `TS` nodes. The vehicles are represented using `V` nodes and their positions are expressed via the `on` edges.



(a) rule `mv`

(b) Type graph

(c) Marking constraint

(d) Initial graph `start`

Figure 4.9: Implicit automaton of Guideway

The `id` attributes are used to distinguish different vehicles, track sections and stations. The `station` attributes indicate which `TS` nodes represent stations. A `TS` node represents a station if it has `station=true`. Each `SLA` attribute expresses whether or not a stoplight is installed at the section junction which lies ahead. For instance, there is no stoplight installed between track sections 1 and 2 because `SLA=false` for track section 1.

All of the guideway's dynamics are expressed by the rule `mv` depicted in figure 4.9a. The rule `mv`, which stands for move vehicle, moves a vehicle from one `TS` node to the next along the direction of travel (dictated by the `L` edges). As both stations and track sections are represented by `TS` nodes, `mv` can also moves vehicles from stations to adjacent track sections and vice versa.

The rule `mv` has four parameters. The first is used to identify which vehicle the rule is applied to.

The latter two respectively indicate the vehicle's origin and destination. The last parameter is used to identify whether a stoplight is installed at the junction the vehicle has crossed.

The marking constraint for this example depicted in figure 4.9c, consists of a positive atomic graph constraint. Intuitively this constraint is satisfied when both vehicles reach station B.

Next we discuss the Guideways specification which is modelled as the forbidden state constraint shown in figure 4.10.

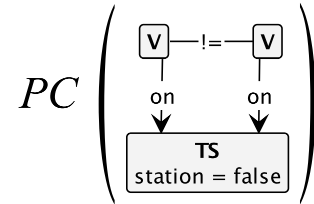


Figure 4.10: Forbidden state constraint

Recall that a positive atomic graph constraint $PC(C)$ is satisfied by a graph G if there exists a graph morphism $m : C \to G$, where m is not necessarily injective. However, in certain instances, it is convenient to impose *injectivity constraints* on pairs of nodes in C which prevent them from being mapped to same node in G. In GROOVE an injectivity constraint may be specified by including a special edge labelled "!=" between the two nodes that should have different images in G. As such, the forbidden state constraint illustrated above is intuitively satisfied when at least two separate vehicles are travelling along the same section of track.

The above implicit automaton and forbidden state constraint are used by GROOVE to generate the induced automaton and forbidden state predicate which are illustrated within the same view in figure 4.11. The states of the induced automaton satisfying the forbidden state predicate (states `s3, s11`) are automatically flagged by GROOVE as forbidden states during the generation of the induced automaton. Remember that each state in the induced automaton is really a graph, state `s6` for example, being illustrated in figure 4.12.

The event set of the resulting induced automaton consists solely of interpretations of `mv`. Event `mv("ii","A","1",true)` for instance, represents the transition of vehicle `ii` from station `A` to track section `1`. Moreover, the last parameter value indicates whether the vehicle crosses a stoplight during the occurrence of the event. This last parameter is used to determine whether the event is controllable or not. More specifically, any event whose last parameter is equal to `true` is controllable and otherwise is uncontrollable. Note that the rule set $\mathcal{R} = \{$`mv`$\}$ is not partitionable with respect to the aforementioned partition of the event set. This is because some interpretations of `mv` are controllable while some are not.

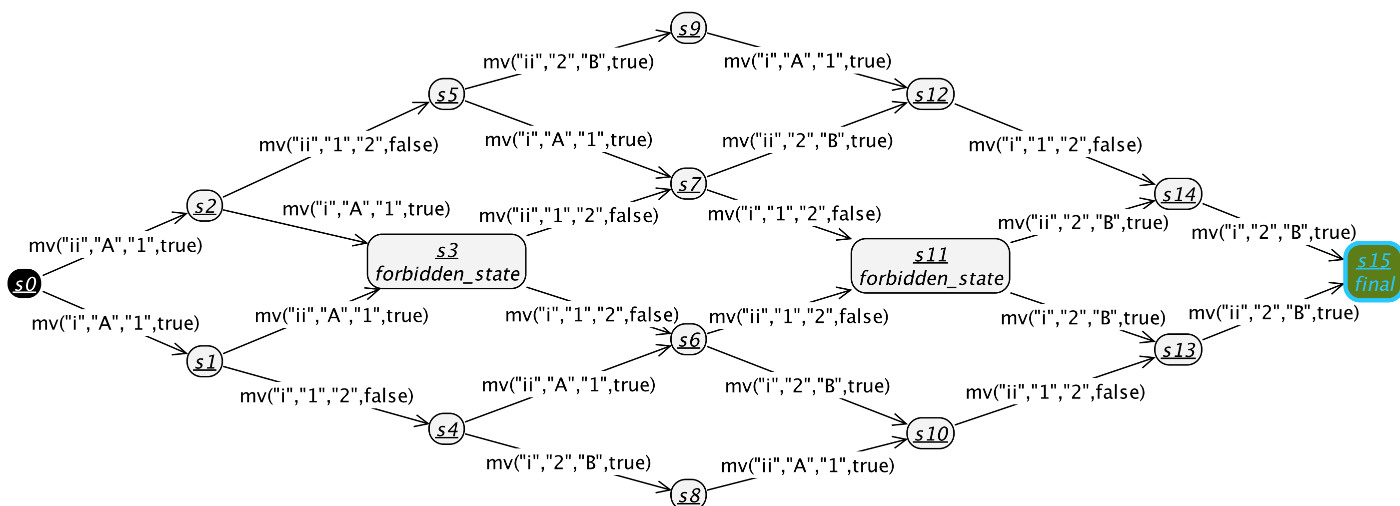


Figure 4.11: Induced plant automaton of Guideway

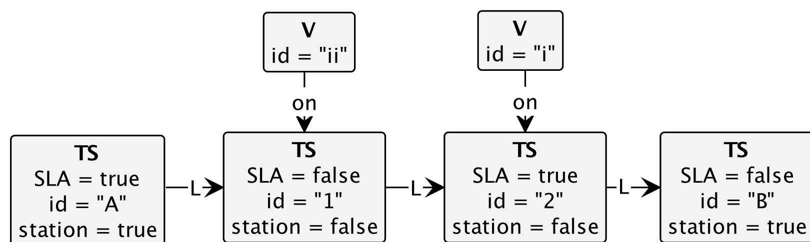


Figure 4.12: State `s6` of the Guideway's induced automaton

Chapter 5

Discussion

In chapter 4 it was illustrated how the proposed graph rewriting modelling framework can be used to generate the models needed as input in both event- and state-based supervisor synthesis algorithms. In this chapter we discuss the advantages and disadvantages of the proposed modelling approach over the manual automaton design methodology described in chapter 2. In doing so, we address the issues of user-friendliness, model intelligibility, model modifiability, susceptibility to error as well as computational complexity. We then conclude the chapter by discussing the feasibility and benefits of employing this modelling approach in an industrial setting.

To avoid confusion, in the remainder of this chapter we will refer to the graph rewriting modelling framework as the "proposed modelling framework" and refer to the manual automaton programming methodology as the "traditional modelling framework".

5.1 User-friendliness

In both the event- and state- based approach, implicit automata are used to model the plant. The implicit automaton model of a system is naturally derived from the informal description of the system. Much like in object oriented programming, the modeller begins by defining the different types of objects and relationships involved in the system's state via the type graph. Using instances of these specified objects and relationships, the initial state of the system is then expressed via the initial graph. The initial graph is trivial to create as a system's state is naturally represented via (attributed) graphs.

The dynamics of the system are represented by a set of graph transformation rules. As they simply specify how the entities and relationships of a system's state must be modified, transformation rules are also intuitively obtained from the informal system description. For example, in the

Guideway problem presented in section 4.4.2, the crossing of a vehicle from one section of track to the next is naturally represented by redirecting the vehicle's position pointer (i.e. `on` edge) to the `TS` node representing the section of track it has just entered (see rule `mv` depicted in figure 4.9a).

One of the major advantages brought about by representing system states as graphs is the ability to probe the existence of substructures within these graphs using graph constraints. This enables the modeller to concisely reference a subset of states which possess a common substructure of interest. A good example which demonstrates the generality of graph constraints is the forbidden state constraint displayed in figure 4.10. This forbidden constraint references all states in which two or more vehicles are travelling along the same section of track irrespective of the number of vehicles or the topology of the track. In the traditional framework, it is not possible to reference subsets of the state space in terms of common substructures because the states of the constituent automata used to construct the plant are atomic, i.e. they lack underlying structure. Instead, one must exploit structure within the state space of the composite automaton[1] that is the plant.

The inherently intuitive process of creating implicit automata is made even more user-friendly by the graphical user interfaces provided by graph transformation tools such as GROOVE.

All of these aforementioned benefits are key as experience with applying SCT in an industrial setting has shown that its successful application is contingent on the *user-friendliness* and *expressiveness* of the modelling formalism[27, 14, 20, 2].

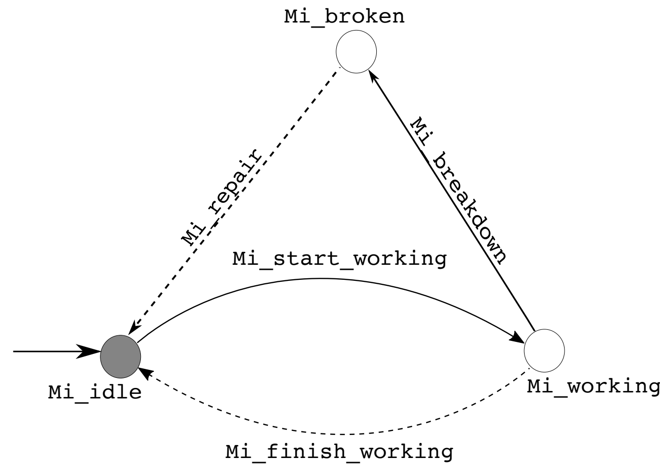
5.2 Model intelligibility

In the proposed modelling framework, the modeller can use the implicit automaton model to understand the system under representation instead of the generated automaton model itself. One of the main advantages of the implicit automaton is that it always explicitly expresses the overall architecture/topology of the system via the initial graph.

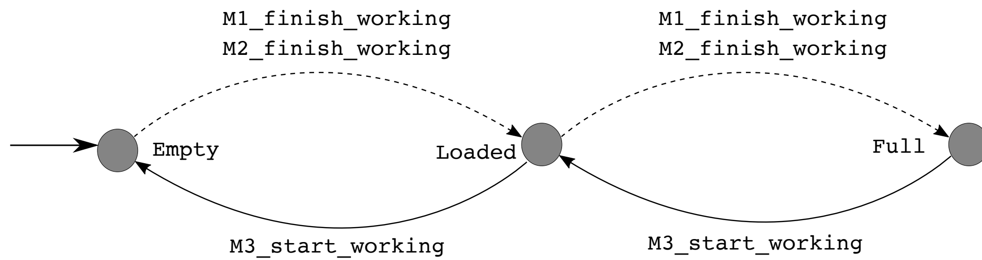
In the traditional modelling framework, constituent automata are used to model different aspects of the plant's behaviour (and specification in event-based SCT). These constituent automata are used to understand the system as the complete plant automaton obtained from their synchronous composition is typically unintelligible. For instance, the Big Factory example presented in section 4.4.1, can be modelled by the constituent automata depicted in figure 5.1. The automata illustrated in figure 5.1a represent the three machines and the automaton shown in figure 5.1b represents the buffer specification.[2]

[1] A composite automaton is one constructed via a synchronous composition of many constituent automata. Its state space comprises tuples of states of the constituent automata used to construct it.

[2] The automaton of the repair priority is omitted as it does not pertain to the argument at hand.



(a) DFA of machine $i = 1, 2, 3$



(b) DFA of buffer specification

Figure 5.1: Constituent automata of the Big Factory

In this case, it is clear from the constituent automaton representation that the Big Factory comprises three machines and one buffer. However, this representation does not explicitly express how these subsystems of plant are related to one another. As a result, it is not immediately clear how the machines and buffers are connected. By contrast, a quick glance at the initial graph of the implicit automaton of the Big Factory illustrated in figure 4.1g immediately reveals the overall architecture of the system.

It is worth noting that the relative advantages held by the two modelling approaches under discussion differ depending on the particular example at hand. Even though the proposed modelling approach might yield a more intelligible model for the buffer and machine example it may not be the case for all other examples. To gain more perspective on the matter, let us examine the Guideway example presented in section 4.4.2.

Using the traditional modelling approach, the two-vehicle Guideway example can be modelled by the two constituent automata depicted in figure 5.2. Automaton **V1** models the dynamics of one vehicle and **V2** models the dynamics of the second.

The overall topology of the track is clearly represented within the automaton models of the Guideway shown in figure 5.2. Hence for the Guideway example, the implicit automaton model

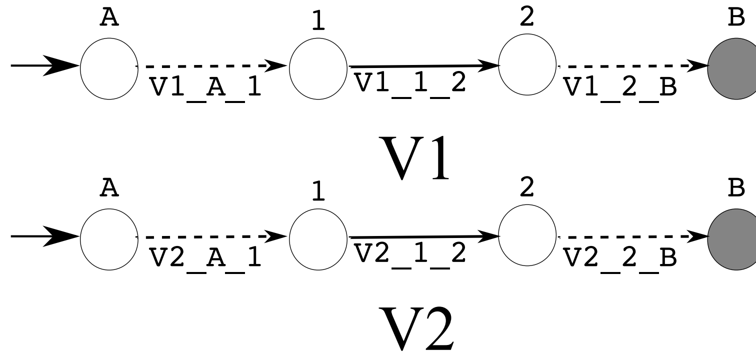


Figure 5.2: Modular automaton model of Guideway

(figure 4.9) does not really provide any advantage over the traditional automaton model (figure 5.2) from an intelligibility perspective.

From these two examples we see that proposed modelling approach does not always provide a clear advantage over the traditional automaton-based modelling approach with respect to model intelligibility. Instead, the relative advantages of these two modelling approaches differs depending on the system being modelled.

5.3 Model modifiability

When modifying or extending existing system models the modeller must translate informally specified changes into concrete model modifications. In the proposed modelling framework, the translation is usually straightforward and the changes are typically easy to make. This is particularly true when making structural changes to the system which do not affect its general dynamics. This is because the system architecture is decoupled from the dynamics and is exclusively embodied within the initial graph. As a result, such structural changes lead to simple and intuitive modifications in the initial graph.

Moreover, as graph constraints provide such general means of specifying subsets of the state space, it is often possible to define them with enough generality that they need not be modified following changes to an initial graph. As such, the modeller can define general marking and forbidden state constraints when the system is initially modelled in order to alleviate model maintenance.

To demonstrate these claims, suppose we wish to modify the model of the Guideway by adding new sections of track and an additional station according to figure 5.3 and an additional vehicle. Note that this is a purely structural change which does not change the Guideway's general dynamics (i.e. how vehicles move from one section of track to next etc.).

To effect this change in the Guideway's implicit automaton, one must simply update the initial graph to reflect the new topology. The Guideway's new initial graph is illustrated in figure 5.4.

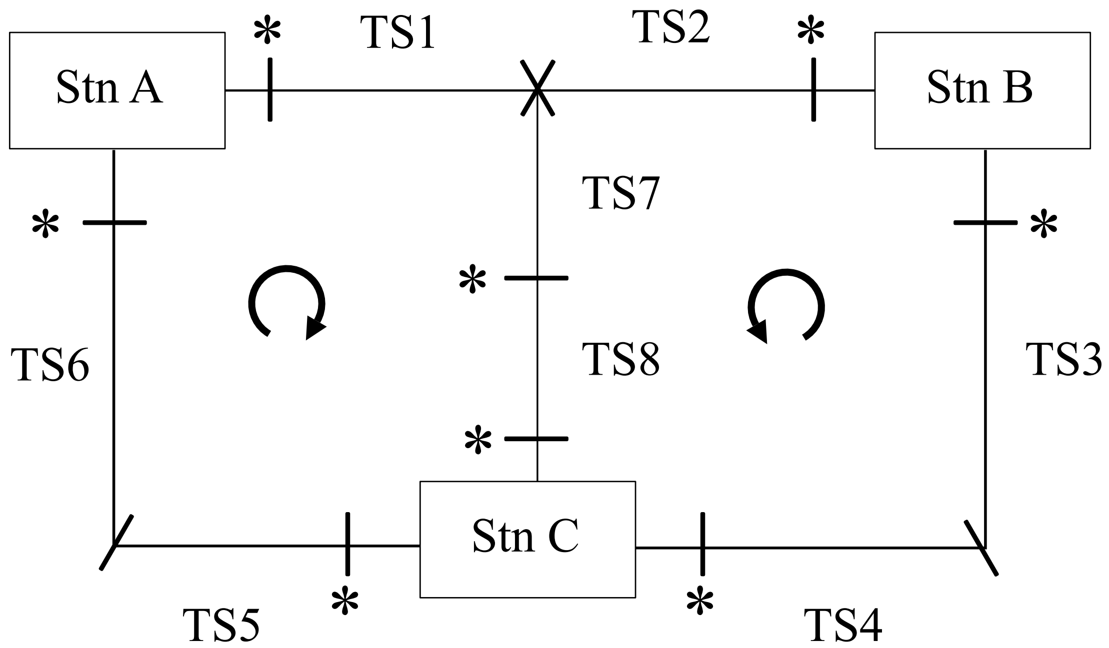


Figure 5.3: New topology of Guideway

There is no need to modify the type graph(figure 4.9b), the rule `mv` (figure 4.9a) or the forbidden state constraint (figure 4.10).

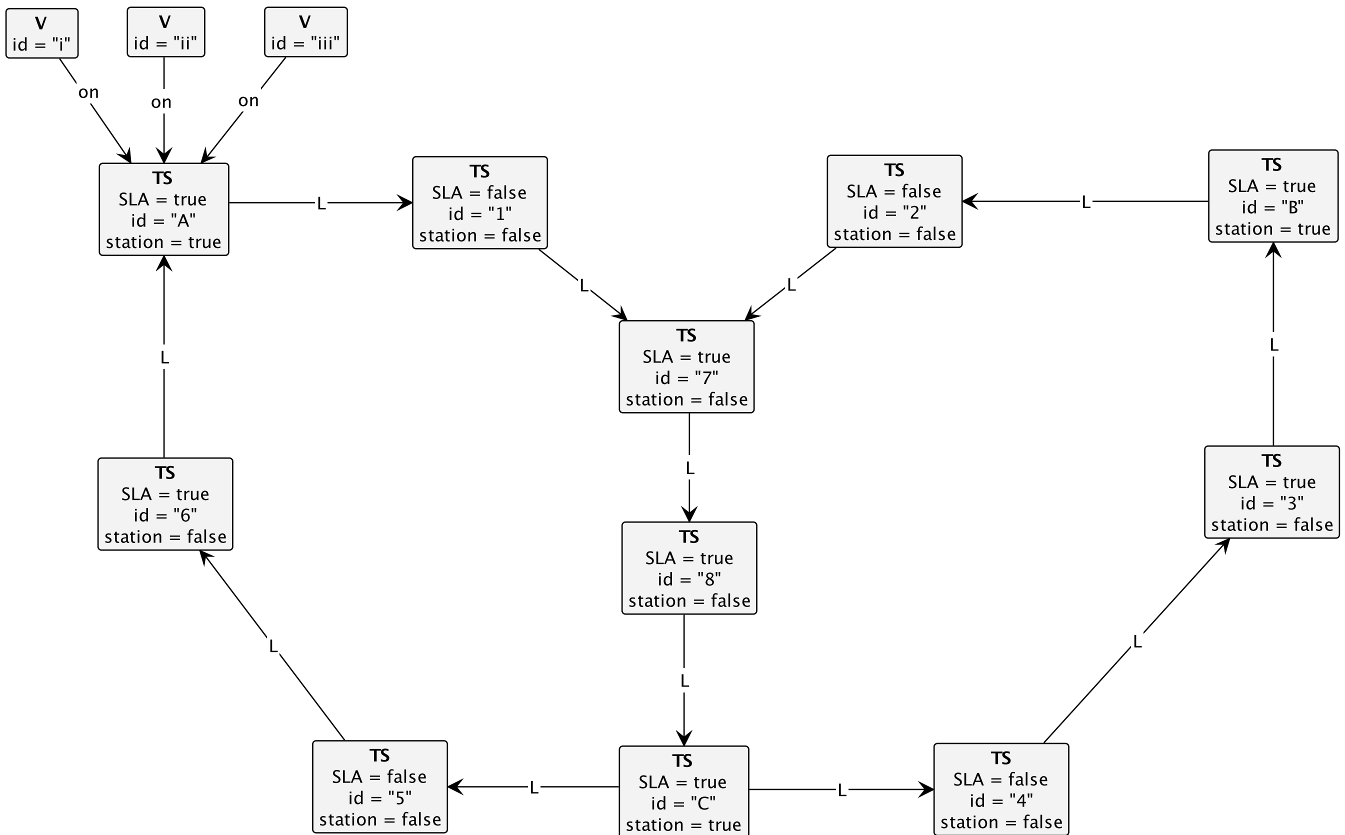


Figure 5.4: New initial graph of Guideway

For argument's sake, suppose we would like to update the marking constraint to ensure that all vehicles are able to end up at a station (not just station B). To effect this change the Guideway's marking constraint is simply changed to the graph constraint depicted in figure 5.5.

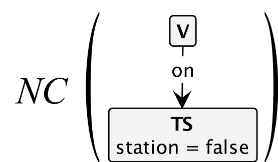


Figure 5.5: New marking constraint of Guideway

Intuitively this graph constraint is unsatisfied whenever a vehicle is located on a section of track which is not a station. In other words it is satisfied whenever all vehicles are located at stations. Note that this new marking constraint is more general than the last and would not need to be modified if additional changes were made to the the topology of the track.

Once all of these necessary changes are made to the implicit automaton, the modeller may simply generate the new automaton model of the Guideway using the GTT. As such, this modelling approach eliminates the need to make tedious, time-consuming and potentially nontrivial manual changes to an automaton's transition structure.

In contrast, suppose one wants to effect these types of changes in the Guideway's automaton model shown in figure 5.2. In this case, the modeller needs to first modify the state and transition structure of the existing automaton models to reflect the changes in the topology of the track. He then needs to create a copy of the automaton model for each newly introduced vehicle and customize its transition labels to differentiate it from the automaton models of the other vehicles.

In general, if there are k sections of track and n vehicles, the implicit automaton model representation has order $k + n$ vertices because the implicit automaton only stores one copy of track in the form of the initial graph. On the other hand, the automaton representation has order $k * n$ states because it stores one copy of the track for each vehicle in the system. In this particular example, we see how it is easier to modify the topology of the track and increase the number of vehicles in the implicit automaton model than in the standard automaton model.

5.4 Susceptibility to error

In this section, we argue that the process of obtaining automaton models via implicit automata is inherently less error prone than manually creating them. To convince the reader of this claim we first analyse the process of manually designing automata and compare it to the proposed modelling approach in which implicit automata are manually designed and in turn used to automatically generate induced automata.

The design of any automaton model can be decomposed into the following two main steps.

Manual construction of automata:

1. The modeller reads an informal description of the system and then fabricates a general and informal model of the system's state in his mind.
2. Using this general and informal model of the system's state he constructs the explicit state, transition and marking structure of the automaton.

The first step is an intuitive process which is common to the design of any type of formal model which cannot be circumvented. The second step is specific to the design of formal models which explicitly express the state and transition structure of the system. In this step the modeller typically follows a design procedure which closely resembles how the GTT constructs the induced automaton. That is, he begins by creating the initial state and gradually adds new states and transitions. He decides whether or not an event is possible from a given state based on an abstract understanding of the state's internal structure. This is analogous to determining the applicability of a graph transformation rule. To determine whether the occurrence of an event leads to a new or already existing state he subconsciously performs the following two step procedure:

i. On the basis of his abstract understanding of the event, he modifies his informal model of the source state to obtain a model of the target state of the transition. This step is analogous to the construction of the target graph based on the source graph and transformation rule.

ii. He then compares this newly obtained informal model of the target state with that of all the other states to verify that it does not already exist. If the model does not match that of any other state, the transition leads to a new state. On the other hand, if the model does match that of another state, the transition leads to this already existent state. This step is analogous to the merging of isomorphic states in the construction of the induced automaton.

In this manner the modeller introduces new states and transitions until the automaton model is complete.

Now consider the process involved in obtaining automaton models via implicit automata. This process can be decomposed into the following three main steps.

Generating induced automata via implicit automata:

1. The modeller reads an informal description of the system and fabricates a general and informal model of the system's state in his mind.
2. The modeller then formalizes this general and informal description of the system's state in the form of an implicit automaton.
3. The implicit automaton is then given as input to a GTT which automatically computes the explicit state, transition and marking structure of the induced automaton.

In the second step of this modelling approach, the modeller translates his general and informal understanding of the system into an implicit automaton. This process is intuitive and as a result less error prone because the implicit automaton tends to correspond more closely with the general manner in which the modeller informally understands the system. The exercise of specifying the implicit automaton can also help reveal inconsistencies in the modeller's understanding of the system's internal structure.

Furthermore, the implicit automaton contains a concrete model of the state, which is formally related to the system's dynamics via the set of graph transformation rules. Therefore, the GTT always generates an induced automaton in which these specified relationship between the states and graph transformation rules are respected. That is, the induced automaton is correct by construction up to the validity of the implicit automaton.

To help ensure that the implicit automaton is itself correct, elements included in the initial graphs, transformation rules and marking constraint which are not predefined within the type graph are flagged as erroneous (see section 4.3.3).

Lastly, in certain instances, parametrized graph transformation rules can be used to automate the labelling of the transitions within automata. This technique is demonstrated within the Guideway example presented in section 4.4.2. In this example, the parametrized rule `mv` is used to automatically label the transitions associated to each vehicle/origin/destination combination in the induced automaton depicted in figure 4.11.

5.5 Computational complexity

This structured approach to generating automata does not come without a cost. The main issue with this approach being the large time and space complexities of generating the induced automaton. In this section we first discuss the time complexity associated to generating induced

automata. With these computational limitations in mind, we then discuss the feasibility of employing this modelling approach in an industrial setting and explain the benefits of doing so. Readers seeking additional detail on complexity and implementation are referred to [25].

The generation of the induced automaton can be subdivided into two main steps.

1. Generating the graph transition system
2. Resolving the marked state set

1. Generating the graph transition system

The process of generating the graph transition system, as implemented in GROOVE, can itself be decomposed into the following three phases:

i Computing rule matchings

ii Applying rules

iii Performing isomorphism checks

i. Computing rule matchings

Finding a match of the left-hand side of a rule within a source graph is an instance of the graph matching problem(GMP) [1]. This problem is a generalization of the subgraph isomorphism problem and is NP-complete in the size of the graph being matched. Luckily, the left-hand sides of transformation rules tend be quite small alleviating the computational burden. The complexity of this phase is exacerbated with the use of negative application conditions. In addition, the complexity of this step is evidently linear in the number of transformation rules.

ii. Rule application

The process of actually modifying the source graph to obtain the target graph is typically the least time consuming step. It involves simply deleting and or adding nodes and or edges to the source graph.

iii. Isomorphism check

The task of verifying whether a newly generated state is isomorphic to existing states requires solving an instance of the graph isomorphism problem (GIP). Although, the exact complexity of the GIP is still unknown, it is thought to be NP-intermediate. Like the GMP, its complexity is proportional to the size of the graphs under comparison. In this case, the size of the graphs can

pose more significant problems as they represent the entire system state. It is worth noting however, that much progress has been made in developing algorithms that efficiently solve the GIP in practice [13]. Lastly, the maximum number of potential isomorphism checks grows quadratically with the number of states. However, in practice it is possible to drastically reduce the number necessary comparisons. GROOVE for example, uses so-called *graph certificates*, which are necessary indicators of isomorphism, to reduce the number of executed isomorphism checks[25].

2. Resolving the marked state set

In this step, the GTT must resolve which states are satisfied by the marking constraint. To confirm a particular state is marked the GTT must solve an instance of the GMP making this step identical to the rule matching phase.

Note that although the aforementioned computational tasks are considered separately their executions must in fact be interleaved.

5.6 Industrial relevance

Having considered the computational complexity of generating induced automata, it is concluded that generating a monolithic induced automaton which models an industrial scale system is intractable. However, one can borrow the modular design techniques employed within the traditional framework to help alleviate this issue. This is accomplished in following manner:

i. First several implicit automata are designed to model the different subsystems of the plant.

ii. The corresponding induced automaton models of these subsystems are then generated using a GTT.

iii. The synchronous composition of these induced automata is then computed using an SCT tool to obtain the complete plant model.

As long as the system is decomposed into sufficiently small subsystems the induced automata can be generated in reasonable amounts of time. This allows the modeller to take advantage of the intuitive and user-friendly nature of this high-level modelling language without incurring the associated computational penalty.

Furthermore, as the constituent automata are automatically generated by a GTT, the modeller no longer needs to take into account the level of difficulty involved in constructing them. This gives the modeller more freedom in deciding how to break up the system. This is because he is no longer obligated to “over decompose” it in order to facilitate the manual creation of the

constituent automata. In the proposed approach, the size of the subsystems are only constrained by the amount of time needed to generate them.

To give the reader a general idea of the size of the subsystems which can be generated in reasonable time, GROOVE can generate a graph transition system representing a system of 10 dining philosophers in 199 seconds[25]. This graph transition system has a total of 31,104 states and 116,658 transitions. Each state has on average 37.7 and 113.8 nodes and edges respectively. In the context of modelling, a 199 seconds is relatively small, as the modelling phase can easily take hours if not days.

Chapter 6

Conclusion

Since its theoretical conception, SCT has matured into practical viable supervisory controller design methodology. Despite having been shown to provide numerous advantages such as accelerating the development of supervisory controllers and in turn helping support system evolvability [27], SCT has still not yet found widespread application in industry. This is due in part to its unintuitive automaton-based modelling framework which has rendered it inaccessible to non-SCT experts.

In attempt to increase the user-friendliness of modelling within the framework, this thesis proposes the use of a graph rewriting-based modelling approach. The proposed modelling approach improves the usability of the SCT framework by discharging the modeller from task of manually constructing the explicit state and transition structure of automata. Instead the modeller is entrusted with designing a more intuitive, front-end model, referred to as the implicit automaton. Once the implicit automaton is designed it is given as input to a graph transformation tool which uses it to automatically generate the corresponding induced automaton model. These induced automata can then be used as desired within SCT framework.

As the design of the implicit automaton is more intuitive and natural than that of the automaton itself, modelling is made less error-prone and more user-friendly. Moreover, since the implicit automaton explicitly represents the internal structure of the systems state, model intelligibility is improved. Furthermore, as implicit automata decouple system architecture from system dynamics, they are more amenable to modification. As a result, the proposed graph-based modelling approach facilitates and expedites the process of updating prior plant and specification models, increasing SCTs capacity to support system evolvability.

For these reasons, it is confidently believed that the proposed modelling approach can significantly improve the usability of the SCT framework and in turn make it more accessible to a wide range of potential users.

6.1 Future work

As the proposed modelling approach simply generates automaton models used within SCT, it is (or can easily be made) compatible with the core features of SCT and its extensions. An example of this, mentioned in section 5.6, is its compatibility with the modular design techniques used (in the traditional modelling framework) to create composite plant and specification automata. As another example, the proposed modelling approach could also be used in the context of partial observation[6] by simply partitioning the induced automaton's event set into observable and unobservable events. Therefore, future work could include the integration of the proposed modelling approach within other extensions of SCT. Furthermore, one could verify whether the newly introduced structure of the state could be leveraged in that particular extension. For example, in the state-based approach, the structure of the state is used to concisely express mutual state specifications in terms of forbidden state constraints. One SCT extension in particular, which could benefit from this modelling approach is supervisory control of state tree structures [19].

In this thesis, we have chosen the algebraic approach to graph rewriting as the engine of our proposed modelling approach. We have done so because it provides enough flexibility to express nontrivial system models and specifications. However, the fundamental principle underlying this work, which is to use rewriting systems (not necessarily graph-based) to implicitly specify a system model and in turn generate its explicit counterpart, is independent of the rewriting approach one chooses to employ. As such, another potential direction of future work could be to investigate the use of other less computationally demanding rewriting systems to generate automaton models.

Bibliography

[1] E. Bengoetxea. *Inexact Graph Matching Using Estimation of Distribution Algorithms.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, Paris, France, Dec 2002.

[2] E. Bertens, R. Fabel, M. Petreczky, D. A. van Beek, and J. E. Rooda. Supervisory control synthesis for exception handling in printers. In *Proceedings Philips Conference on Applications of Control Technology*, 2009.

[3] R. H. Bishop. *Mechatronic Systems, Sensors, and Actuators: Fundamentals and Modeling.* The Mechatronics Handbook, Second Edition. CRC Press, 2007.

[4] Francis Borceux. *Handbook of Categorical Algebra*, volume 1. Cambridge University Press, 1994. Cambridge Books Online.

[5] Christos G. Cassandras et al. *Introduction to Discrete Event Systems.* Springer Science & Business Media, 2008.

[6] R. Cieslak, C. Desclaux, A. S. Fawaz, and P. Varaiya. Supervisory control of discrete-event processes with partial observations. *IEEE Transactions on Automatic Control*, 33(3):249–260, Mar 1988.

[7] Stephen A. Cook. The complexity of theorem-proving procedures. In *In STOC*, pages 151–158. ACM, 1971.

[8] Reinhard Diestel. Graph theory, vol. 173 of. *Graduate Texts in Mathematics*, 2005.

[9] H. Ehrig, M. Pfender, and H. J. Schneider. Graph-grammars: An algebraic approach. In *Switching and Automata Theory, 1973. SWAT '08. IEEE Conference Record of 14th Annual Symposium on*, pages 167–180, Oct 1973.

[10] Hartmut Ehrig, Karsten Ehrig, Annegret Habel, and Karl-Heinz Pennemann. Constraints and application conditions: From graphs to high-level structures. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, *Graph Transformations*, volume 3256 of *Lecture Notes in Computer Science*, pages 287–303. Springer Berlin Heidelberg, 2004.

[11] Hartmut Ehrig and B. Mahr. *Fundamentals of Algebraic Specification I.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1985.

[12] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. *Graph Transformations: Second International Conference, ICGT 2004, Rome, Italy, September 28–October 1, 2004. Proceedings*, chapter Fundamental Theory for Typed Attributed Graph Transformation, pages 161–177. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[13] Pasquale Foggia, Carlo Sansone, and Mario Vento. A performance comparison of five algorithms for graph isomorphism. In *Proceedings of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*, pages 188–199, 2001.

[14] Stefan T. J. Forschelen, Joanna M. van de Mortel-Fronczak, Rong Su, and Jacobus E. Rooda. Application of supervisory control theory to theme park vehicles. *Discrete Event Dynamic Systems: Theory and Applications*, 22(4):511–540, 2012.

[15] Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer. *Graph Transformation: First International Conference, ICGT 2002 Barcelona, Spain, October 7–12, 2002 Proceedings*, chapter Confluence of Typed Attributed Graph Transformation Systems, pages 161–176. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[16] Reiko Heckel, Georgios Lajios, and Sebastian Menge. Stochastic graph transformation systems. *Fundamenta Informaticae*, 74(1):63–84, 2006.

[17] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation; 3rd ed.* Pearson, Boston, MA, 2007.

[18] Michael Lwe. Algebraic approach to single-pushout graph transformation. *Theoretical Computer Science*, 109(1):181 – 224, 1993.

[19] C. Ma and W.M. Wonham. *Nonblocking Supervisory Control of State Tree Structures.* Lecture Notes in Control and Information Sciences. Springer Berlin Heidelberg, 2005.

[20] Jasen Markovski, Koen GM Jacobs, Dirk A van Beek, Lou J Somers, and Jacobus E Rooda. Coordination of resources using generalized state-based requirements. In *WODES*, volume 10, pages 300–305, 2010.

[21] John L. Pfaltz and Azriel Rosenfeld. Web grammars. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, IJCAI'69, pages 609–619, San Francisco, CA, USA, 1969. Morgan Kaufmann Publishers Inc.

[22] Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5(6):560 – 595, 1971.

[23] Peter J Ramadge and W Murray Wonham. Supervisory control of a class of discrete event processes. *SIAM Journal on Control and Optimization*, 25(1):206–230, 1987.

[24] A. Rensink. Representing first-order logic using graphs. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *International Conference on Graph Transformations (ICGT)*, volume 3256 of *Lecture Notes in Computer Science*, pages 319–335, Berlin, 2004. Springer Verlag.

[25] Arend Rensink. Time and space issues in the generation of graph transition systems. *Electronic Notes in Theoretical Computer Science*, 127(1):127 – 139, 2005. Proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)Graph-Based Tools 2004.

[26] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[27] R.J.M. Theunissen, D.A. van Beek, and J.E. Rooda. Improving evolvability of a patient communication control system using state-based supervisory control synthesis. *Advanced Engineering Informatics*, 26(3):502 – 515, 2012. Evolvability of Complex Systems.

[28] W. M. Wonham and P. J. Ramadge. On the supremal controllable sublanguage of a given language. *SIAM Journal on Control and Optimization*, 25(3):637–659, 1987.

[29] W. Murray Wonham. *Supervisory Control of Discrete-Event Systems*. Dept. Elect. Comput. Eng., Univ. Toronto, Toronto, ON, Canada, 2012. Available at http://www.control.toronto.edu/DES/.