OSKI KERVINEN
AN ARCHITECTURE FOR VERIFICATION WITH EXTENDED LABELED TRANSITION SYSTEMS
Master’s Thesis

Examiners:
Antti Valmari & Henri Hansen
The examiners and topic of the thesis were approved by the Council of the Faculty of Natural Sciences on 2014-12-03
TIIVISTELMÄ

TAMPEREEN TEKNILLINEN YLIOPISTO
Teknis-Luonnontieteellinen koulutusohjelma
OSKI KERVINEN: AN ARCHITECTURE FOR VERIFICATION WITH EXTENDED LABELED TRANSITION SYSTEMS
Diplomityö, 86 sivua, 5 liitesivua
Tammikuu 2015
Pääaine: Ohjelmistotiede
Tarkastajat: Antti Valmari & Henri Hansen
Avainsanat: LTS, verifiointi, rinnakkaisuus, metaohjelmointi

Tässä työssä johdetaan ja esitellään rakenne rinnakkaisten tilakoneiden verifiointiohjelmistolle. Ohjelmiston tavoitteena on tarjota kehys, johon erilaisia tilakoneiden kutistusoperaatioita ja muita manipulaatioita voidaan toteuttaa ja yhdistellä helposti. Erilaiset kutistukset tarjoavat erilaisia näkökulmia järjestelmän toimintaan.

ABSTRACT

TAMPERE UNIVERSITY OF TECHNOLOGY
Master’s Degree Programme in Mathematics
OSKI KERVINEN : AN ARCHITECTURE FOR VERIFICATION WITH EXTENDED LABELED TRANSITION SYSTEMS
Master of Science Thesis, 86 pages, 5 Appendix pages
January 2015
Major: Mathematics
Examiner: Antti Valmari & Henri Hansen
Keywords: LTS, verification, concurrency, metaprogramming

This thesis lays out a common architecture for a software system meant for verification of correctness of concurrent state machines. The purpose of the architecture is to enable implementation and combination of different reductions and other manipulations, that give perspectives into the behaviour of the system.

The LTS model and associated techniques are presented for a mathematical basis for the system. The system is made flexible using extensive modularization. Even techniques traditionally contained inside the parallel composer, such as cut states and stubborn sets are extracted into modules. Techniques are presented to avoid the performance loss of modularization using compile time processing.
CONTENTS

1. Introduction ............................................. 1
2. Background .............................................. 5
   2.1 Notation ............................................ 5
   2.2 LTS ................................................ 5
      2.2.1 Motivation .................................... 5
      2.2.2 Parallel Composition ....................... 10
      2.2.3 Renaming ..................................... 12
      2.2.4 Hiding ....................................... 14
   2.3 Semantics .......................................... 16
      2.3.1 Sameness ....................................... 16
      2.3.2 Trace semantics ................................ 17
      2.3.3 Congruence ..................................... 18
      2.3.4 Divergence ..................................... 22
      2.3.5 CFFD-preorder ................................ 24
   2.4 Bisimilarity ......................................... 24
   2.5 Reduction .......................................... 26
   2.6 Variables ........................................... 29
      2.6.1 Motivation ....................................... 29
      2.6.2 RLTS ........................................... 30
      2.6.3 RLTS:s and LTS:s ............................ 35
   2.7 Synchronization rules .............................. 36
      2.7.1 For LTS:s ....................................... 36
      2.7.2 For RLTS:s ..................................... 37
   2.8 Composition machinery ............................ 38
      2.8.1 Stubborn sets .................................. 38
      2.8.2 Symmetry reduction ......................... 43
      2.8.3 Cut states ..................................... 43
   2.9 Tester Processes ................................... 44
3. Computation ............................................ 47
   3.1 A Concrete Point of View ......................... 47
   3.2 Transformations ..................................... 47
      3.2.1 Bisimulation minimization .................. 47
      3.2.2 Unfolding ...................................... 48
      3.2.3 CFFD Reduction by Acceptance Graph minimization 48
   3.3 Parallel Composition .............................. 49
   3.4 Composition Augmentations ...................... 49
      3.4.1 Cut States ...................................... 49
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.4.2 Tester Processes</td>
<td>50</td>
</tr>
<tr>
<td>3.4.3 Stubborn Sets</td>
<td>50</td>
</tr>
<tr>
<td>3.4.4 RLTS</td>
<td>50</td>
</tr>
<tr>
<td>4. Architecture</td>
<td>52</td>
</tr>
<tr>
<td>4.1 Wirings and Blocks</td>
<td>52</td>
</tr>
<tr>
<td>4.2 State machine interface</td>
<td>54</td>
</tr>
<tr>
<td>4.3 Annotations</td>
<td>55</td>
</tr>
<tr>
<td>4.3.1 Annotated Graphs</td>
<td>55</td>
</tr>
<tr>
<td>4.3.2 Basic Annotations</td>
<td>57</td>
</tr>
<tr>
<td>4.4 Parallel composition framework</td>
<td>57</td>
</tr>
<tr>
<td>4.5 Annotation Propagation</td>
<td>61</td>
</tr>
<tr>
<td>4.5.1 Blocks Manipulating Annotations</td>
<td>61</td>
</tr>
<tr>
<td>4.5.2 Annotations Manipulating Blocks</td>
<td>62</td>
</tr>
<tr>
<td>4.6 Wiring blocks together</td>
<td>62</td>
</tr>
<tr>
<td>4.7 Parallelization</td>
<td>64</td>
</tr>
<tr>
<td>5. Implementation Ideas</td>
<td>67</td>
</tr>
<tr>
<td>5.1 The Cost of Abstraction</td>
<td>67</td>
</tr>
<tr>
<td>5.2 Static Polymorphism</td>
<td>68</td>
</tr>
<tr>
<td>5.3 Pseudo code notation</td>
<td>73</td>
</tr>
<tr>
<td>5.3.1 Motivation</td>
<td>73</td>
</tr>
<tr>
<td>5.3.2 Notation</td>
<td>73</td>
</tr>
<tr>
<td>5.4 The Architecture</td>
<td>74</td>
</tr>
<tr>
<td>5.4.1 Graph</td>
<td>74</td>
</tr>
<tr>
<td>5.4.2 Annotations</td>
<td>74</td>
</tr>
<tr>
<td>5.4.3 Wiring</td>
<td>75</td>
</tr>
<tr>
<td>5.4.4 Composition</td>
<td>75</td>
</tr>
<tr>
<td>5.4.5 Augmentations</td>
<td>77</td>
</tr>
<tr>
<td>5.4.6 A Colorful Example</td>
<td>77</td>
</tr>
<tr>
<td>6. Conclusions</td>
<td>82</td>
</tr>
<tr>
<td>Sources</td>
<td>83</td>
</tr>
<tr>
<td>A. Appendix</td>
<td>i</td>
</tr>
<tr>
<td>A.1 Example Static Interface in D</td>
<td>i</td>
</tr>
<tr>
<td>A.2 Example Static Interface in C++</td>
<td>iii</td>
</tr>
</tbody>
</table>
# DEFINITIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Term</th>
</tr>
</thead>
<tbody>
<tr>
<td>AG</td>
<td>Acceptance Graph</td>
</tr>
<tr>
<td>DFA</td>
<td>Deterministic Finite Automaton</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>DSL</td>
<td>Domain Specific Language</td>
</tr>
<tr>
<td>LTS</td>
<td>Labeled Transition System</td>
</tr>
<tr>
<td>RLTS</td>
<td>Relational Labelled Transition System</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

Informally, concurrency means that different things happen at the same time and sometimes interact with each other. As far as anyone can tell, reality is concurrent. Traditional programming languages and models of computation are not. In the semantics of traditional sequential programming languages the execution is always at some specific point, a Turing machine is always in a single state and lambda calculus is evaluated a single term at a time.\footnote{Admittedly, the order of evaluation of function parameters in lambda calculus is undefined, which leaves one free to evaluate them simultaneously. However, the evaluations do not interact, therefore this is not concurrency.}

This causes great trouble when we try to connect computers to reality. Concurrency is often discussed in terms of threads, multi-core processors and physically separate machines, but in fact concurrency issues emerge from any form of communication with the real world. Even a user typing while the computer is computing is a concurrent situation.

Another aspect of reality that is not handled well by traditional models of computation is reactivity. Reactivity means that the system does things in response to the actions of a user. The goals of such systems are not easy to describe using formalisms focusing on mapping an input to an output.

Luckily reactivity is subsumed by concurrency. By viewing the user as merely another entity performing concurrent actions, we can describe the user’s actions with any tool made to describe interaction between entities.

Therefore, we need models that deal with concurrent phenomena. Unfortunately concurrency does not only force us to abandon our cherished models of computation, but is genuinely harder even when appropriate models are available.

Let us look at a simple example to understand why this is so. Imagine two threads. We shall call them thread $A$ and thread $B$. $A$ wishes to increment $x$ by 1 and $B$ wishes to increment $x$ by 2. To do this they need to perform three actions.

1. Read $x$

2. Compute what the new value is

3. Write the value to $x$
1. Introduction

You may already be familiar with the fact that this may fail horribly. If both threads read \( x \) before either has written it, only one of the additions will take effect. If one is familiar with concurrency, this may seem obvious, but the fact is that this very mistake has been made, is being made and will be made countless times.

Why? Because the writer of the program did not come to think of that the events may occur in that order. This is entirely reasonable. We generally write and read programs linearly and therefore we naturally think about programs linearly. Learning to think concurrently is hard, if not impossible. We shall illustrate this further.

|    | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|----|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| A1 | A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1| A1|
| A2 | A2| A2| A2| A2| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1| B1|
| B2 | B3| B3| B3| B3| B3| B3| B3| B3| B3| B3| B3| B3| B3| A3| A3| A3| A3| A3| A3| A3|

Table 1.1: All possible interleavings of three actions by two parties and resulting values of \( x \) when the parties are thread \( A \) and thread \( B \) and \( x \) is initially 0

Table 1.1 shows all the possible interleavings of the three actions taken by threads \( A \) and \( B \) with resulting values of \( x \). There are 20 different interleavings. In general, if two threads have \( a \) and \( b \) independent actions, there are \( \frac{(a+b)!}{a!b!} \) different interleavings. This number grows quickly. It is relatively easy to detect the error in this example, since it functions correctly only rarely, but it is possible that very few of the possible interleavings lead to errors, which makes the errors very hard to detect.

Therefore, formal and automated methods for verification of concurrent systems are necessary. Such tools have been developed since the late 70’s [39].

According to [12, 19] they are used extensively in verifying computer hardware. Even a single core computer is rich in concurrent phenomena thanks to the layers of caches, pipelining and interruptions. In these applications great expenditure of time and effort is justified, since early detection of errors is especially valuable when creating physical products. The methods used are based on modeling the system using a logical equations, formulating the requirements using more logical equations and then proving that they are true with respect to the system. Unfortunately it is usually impossible to prove useful requirements automatically; such approaches require extensive human intervention. In fact, the work required is so large, that efforts have been made to crowd-source parts of it [4].

Some methods approach the problem by modeling the system in some way that allows you to investigate all possible interleavings and check none of them leads to an
error. The approach is called explicit model checking. In practice there are usually too many interleavings to check every one and something must be done to avoid visiting or even constructing them all. When the requirements are fully known in advance and formulated in a manner that allows them to be analyzed, it is possible to determine fairly precisely what information can be thrown away in the model without affecting the correctness of the requirements. Throwing away the irrelevant information is called abstraction.

These tools are very powerful and useful, but they have some weaknesses. Firstly, you need to give them the requirements to check very specifically. Usually this means some form of temporal logic \[37\]. Therefore the user of such a tool must know precisely what they want and be able to describe it using such a method. Expertise like this is not available for all use cases of verification.

Methods like this have been used in hardware verification and other applications where great verification effort is cost-effective, for example telecommunication protocols. With concurrency being ever more visible in ever wider a range of programs, however, methods that could be more widely and easily usable are required. One technique meant to require less expertise is Visual Verification.

Presented by Valmari \[34\], the idea of visual verification is to create projections of the system into different points of view. For example, one may look at a communication protocol from the point of view of a single party or look only at actions release and lock to make sure that a lock never follows another lock before a release.

When the user chooses sufficiently small points of view and a tool is sufficiently good at leaving out things that do not matter from that point of view, the results are simple enough for a human to actually understand by looking at them. Then they can visually verify that the system behaves in the desired manner. Hence the name visual verification.

The upside is that the user does not have to know precisely what they want or describe it formally. They simply have to know the points of view from which they are interested in the system. This approach is naturally limited to checking issues that can be detected from such small points of view.

The computational challenges of verification are so severe and fundamental that it is unlikely they will be solved in a general sense. Therefore they must be combated with a wide variety of techniques suited for specific situations. This requires a very flexible framework that can be used to create and wire together these techniques.

Tampere Verification Tool (TVT) \[38\], created in co-operation with the Nokia Research Center, was a toolkit seeking to fulfill these goals and succeeded fairly well. It has some weaknesses, though. In TVT the separate processing blocks are separate programs. They communicate by reading and writing files. This adds unnecessary overhead not only as the file IO, but also as the parsing and unnecessary
reconstruction of data structures.

In addition the file format was not flexible enough, even though it did have a section system that allowed programs to ignore what they did not understand.

The purpose of this thesis is to lay out an architecture that addresses these issues. It should allow the user to create a system out of pieces that can be combined in many ways. The pieces should also be as easy as possible to create.

We will proceed by describing the mathematical framework used to describe concurrent processes in Chapter 2. There we will also explain different semantics, that is, ways of thinking what a system means. These allow us to reduce the size of the system without changing its meaning, especially when combined with a specific point of view. We will also take a brief look into a few of the aforementioned various techniques to get an idea of what we must support.

In Chapter 3 we will look at the techniques from a more algorithmic perspective and Chapter 4 will formulate a framework for their co-existence. Finally, Chapter 5 will explore ideas on how to implement the architecture efficiently.
2. BACKGROUND

In this section we will describe the mathematical structures underlying our approach to verification. We will begin by presenting the LTS and its operators. Then we will discuss their semantics and some reduction techniques they allow. After that we expand our definitions to support variables and then present some more advanced reductions.

2.1 Notation

First some useful notations that shall be used in the following.

We shall need to express the condition \( i \in \{1, 2, \ldots, n\} \) many times so we will denote that with the more convenient \( i^\pi \).

When we have any relation of the form \( \cdot \triangleleft \cdot \) we shall denote \( (a_1 \triangleleft a_2) \land (a_2 \triangleleft a_3) \land \cdots \land (a_{n-1} \triangleleft a_n) \) by \( a_1 \triangleleft a_2 \triangleleft \cdots \triangleleft a_n \). For example \( 1 < 20 < 11 \iff (1 < 20) \land (20 < 11) \).

Take a sequence of tuples \( s^1, \ldots, s^k \) with possibly different types and lengths \( s^i \in S^1_1 \times S^2_2 \times \cdots \times S^i_{r_i} \). We define an operation to create a single tuple with all their members, \( \text{Concat}(s^1, \ldots, s^k) = (s^1_1, \ldots, s^1_{r_1}, s^2_1, \ldots, s^2_{r_2}, s^3_1, \cdots, s^k_1, \ldots, s^k_{r_k}) \). For example \( \text{Concat}((1, a), (b, 8)) = (1, a, b, 8) \).

2.2 LTS

We will now proceed to illustrate the motives behind the definition of Labeled Transition Systems (LTS), define them and define their basic operators.

2.2.1 Motivation

To apply mathematics to concurrency problems, we must model concurrent systems using some mathematical construct. We want it to be as simple as possible while containing all the information we need. It should also be fairly intuitive.

Let us ground the discussion using a concrete example. We shall describe a shared printer. The printer has two clients and only one of them may be printing at a time. The printer never prints forever.

Let us model this system top down, that is, begin as vaguely as possible and add details piece by piece as they are required. Since we wish to model a printer and two clients, our model must contain something to denote them. We begin by modeling
the fact that they exist and are separate by drawing a box for each one.

![Diagram of printer and clients]

The printer and the clients interact. For the clients to print something, they must communicate with the printer. It is also possible for the client to find out if the printing is done or has failed. We shall model these interactions by drawing lines between the boxes. These lines are not directional. We are not saying that the client is doing something to the printer or vice versa. Our lines are more abstract: they just say that the entities take part in an interaction.

To model the fact that the interaction that initiates printing is different from the interaction by which the user finds out that printing is done, lines between boxes shall have names.

![Diagram with names on lines]

This models the relationships between the printer and the clients, but there is no way to check that the two users cannot be printing at the same time. We need to describe what happens inside the boxes.

To fulfill the requirement the printer must know whether it is printing or not. Also, we should model the difference between failing and succeeding. We shall model these things as the printer having different *states*. When it receives a command to print, it moves to the printing state. Then it tries to print and may fail or succeed.

The details which determine whether the printing succeeds or not are beyond the scope of our model. They are a series of events that we cannot affect or observe, but that results in the printer moving to one state or another. The important thing is that they are irrelevant to our requirements. We shall denote such invisible events with the symbol $\tau$. 
The arrow coming from nowhere denotes that the printer is initially not printing.

Our model is beginning to contain quite a lot of information. We shall simplify it a bit. Our main requirement is that two clients may not be printing at the same time. That can be verified by making sure that there are never two print actions without a fail or done between them. This and in fact all our other requirements can be seen from the transition labels alone, we do not need to have any information in the states. There exist models that keep information in the states, but we will not do that for now. The printer is pictured like this in Figure 2.1. The states still have numbers to denote the fact that they have identity; we can tell them apart.

Now we are modeling a single concurrent entity using a Labeled Transition System. This notion is formalized in Definition 1.

**Definition 1 (Labeled Transition System (LTS))**

An LTS is a 4-tuple \((S, \Sigma, \Delta, \hat{s})\), where

\[ S \]

is the (finite) set of states

\[ \Sigma \]

is the (finite) set of transition labels, also called the alphabet.

\[ \Delta \subseteq S \times (\Sigma \cup \{\tau\}) \times S \]

is the set of labeled transitions

\[ \hat{s} \in S \]

is the starting state

The sets of states and transitions are defined to be finite to simplify our presentation. Much of what we present can be applied to infinite LTS:s, refer to the sources for details.

Let us write out the printer LTS according to the definition. We will use the long state names from above for the purpose of clarity.
\( S = \{ \text{not\_} \text{printing}, \text{printing}, \text{failure}, \text{printed} \} \)

\( \Sigma = \{ \text{print}, \text{fail}, \text{done} \} \)

\( \Delta = \{(\text{not\_} \text{printing}, \text{print}, \text{printing}), (\text{printing}, \tau, \text{failure}), (\text{printing}, \tau, \text{printed}), (\text{failure}, \text{fail}, \text{not\_} \text{printing}), (\text{printed}, \text{done}, \text{not\_} \text{printing})\} \)

\( \hat{s} = \text{not\_} \text{printing} \)

Since there is no way for an LTS to reach a state that cannot be reached from the initial state, such states and their transitions are irrelevant. Therefore we essentially always ignore them. Formally:

**Definition 2 (Reachable part)**

Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS. Then the reachable part of \( L \) is the LTS \( \text{Reach}(L) = (S', \Sigma, \Delta', \hat{s}) \), where

- \( S' \subseteq S \) contains the initial state and every state that can be reached from the initial state using any sequence of transitions.
- \( \Delta' = \Delta \cap (S' \times (\Sigma \cup \tau) \times S') \)

We will usually use natural numbers to identify the states, since the state names are not meant to contain any information. When we discuss concrete LTS:s, we will generally describe them pictorially, but the formal form is the basis for the actual mathematics. When a state is identified with a number \( a \) in a picture, we will refer to it as \( s_a \) in the prose.

When an LTS is described with a picture, it can be assumed that its alphabet has only the labels that can be seen in the picture unless otherwise noted.

LTS:s are a cornerstone of the vocabulary we will be using. Therefore we shall define some useful notations about them here.

Let \( L = (S, \Sigma, \Delta, \hat{s}) \) be an LTS, \( s_1, s_2, \ldots, s_m \in S \), \( b \in (\Sigma \cup \{\tau\}) \), \( a_1, a_2, \ldots, a_n \in \Sigma \) and \( \sigma = a_1 a_2 \ldots a_n \). Note that it is possible that \( n = 0 \), then the string \( \sigma \) is the
empty string $\varepsilon$. With these assumptions we denote:

\begin{equation}
\begin{split}
s_1 \xrightarrow{b} s_2 & \iff (s_1, b, s_2) \in \Delta \\
s_1 \xrightarrow{b} & \iff \exists s' : s_1 \xrightarrow{b} s' \\
s_1 \xrightarrow{b} s_2 & \iff \exists s_1 = r_1, r_2, \ldots, r_l, \ldots, r_k = s_2 \in S : \\
& \quad r_1 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_{l-1} \xrightarrow{b} r_l \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_k \\
s_1 \xrightarrow{a} s_2 & \iff (\sigma = \varepsilon \land s_1 = s_2) \lor \\
& \quad (\sigma = \varepsilon \land s_1 \xrightarrow{\tau} s_2) \lor \\
& \quad \exists s_1 = r_1, r_2, \ldots, r_{n+1} = s_2 : \\
& \quad r_1 \xrightarrow{a_1} r_2 \xrightarrow{a_2} \ldots \xrightarrow{a_n} r_{n+1}, \\
en(s_1) & = \{a \in \Sigma \cup \{\tau\}|s_1 \xrightarrow{a}\}
\end{split}
\end{equation}

Now we can discuss transitions a bit more handily. For example, in the LTS of Figure 2.1 $s_1 \xrightarrow{\text{print}} s_2$, $s_2 \xrightarrow{\text{done}} s_1$ and $s_4 \xrightarrow{\text{fail}} \xrightarrow{\text{print}} \xrightarrow{\text{done}} \xrightarrow{\text{print}} s_2$.

We will say that an action $a$ is enabled in a state $s$ iff $s \xrightarrow{a}$.

It is time to describe the behaviour of the clients. The client sometimes wants to print things. After they have told the printer to print things, they must be ready to hear it is done or that is has failed. Furthermore, they may at any point get a tablet computer and stop printing forever. This is shown in Figure 2.2.

![Figure 2.2: The printer client](image)

We now have our first attempts at describing the insides of the entities of our system. We will have to change them later when problems appear, but we have already captured a large part of the behaviour of the system. It is time to make our entities interact.
2. Background

2.2.2 Parallel Composition

There are many forms of interaction, asynchronous and synchronous, with channels, delays, unreliability, etc. It can, however, be shown that most, if not all of them can be emulated using simple synchronization \[28\]. Therefore we will adopt it as our fundamental composition primitive.

We have decided that the labels of transitions are the point of interest in our parallel entities. Therefore we must define interaction in terms of them. We will limit our discussion to interactions between two entities for now.

In our model an interaction shall be modeled with two LTS:s taking a transition with the same label simultaneously. For example: when the printer and the client are interacting, the client can take the transition labeled \texttt{print} if and only if the printer takes it also.

Sometimes parallel composition is defined so that it takes parameters that specify which events the LTS:s must take together and which they may continue to take independently. We shall simply say that any transition with a label that belongs to the alphabets of both machines must be taken together and all others are taken independently. Many other parallel compositions can be simulated with this one, although it requires some extra machinery that has not been discussed yet and may come with an exponential cost \[28\].

The invisible \(\tau\) transitions always remain independent. That is a part of their role. We cannot require them to happen together, since we do not know what happens behind them.

The system that we get by composing the two LTS:s with each other has states, defined by what states the original LTS:s are in, and transitions, defined by what transitions are available to both machines. Hence, the result can be described using an LTS. This means that parallel composition is actually an operation in the set of LTS:s.

We could make it binary, but we shall go ahead and give the \(n\)-ary generalization while we are at it. In all following definitions, the set \(LTS\) is the set of all Labeled Transition Systems.

\textbf{Definition 3 (Parallel Composition)}

Let \(L_1, L_2, \ldots, L_n \in LTS\) and \(\forall i \in \pi : L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i)\).

Then \(L_1||L_2||\ldots||L_n\) is the reachable part of the LTS \((S', \Sigma', \Delta', \hat{s}')\), where

\[
S' = \bigotimes_{i \in \pi} S_i \\
\Sigma' = \bigcup_{i \in \pi} \Sigma_i \\
\hat{s}' = (\hat{s}_1, \hat{s}_2, \ldots, \hat{s}_n)
\]
Figure 2.3: The LTS $L_{\text{client}} \parallel L_{\text{printer}}$ formed by parallel composition from $L_{\text{client}}$ and $L_{\text{printer}}$. Each reachable state $(s_i, s_j) \in S_{\text{client}} \times S_{\text{printer}}$ is labeled $ij$.

\[
\Delta' = \left\{ \left( (s_1, s_2, \ldots, s_n), a, \left( s_{1}', s_{2}', \ldots, s_{n}' \right) \right) \middle| a \in \Sigma' \land \forall i \in \Pi : s_i, s_i' \in S_i \land \left( (s_i, a, s_i') \in \Delta_i \lor (a \notin \Sigma_i \land s_i = s_i') \right) \right\} \cup \left\{ \left( (s_1, s_2, \ldots, s_n), \tau, \left( s_{1}, s_{2}, \ldots, s_{k}', \ldots, s_{n} \right) \right) \middle| (s_k, \tau, s_{k}') \in \Delta_k \land \forall i \in \Pi : s_i \in S_i \right\}
\]

The state the composition is in is determined by the states of the parts and the alphabet is the union of the component alphabets. The transitions of the composition is the union of the visible transitions and the invisible transitions. For a visible transition to occur, all machines with its label in their alphabets must take part in it, while the rest stay still. On the other hand, any machine is free to take its invisible transitions independently of the others.

To illustrate parallel composition, we have pictured the composition of the LTS of the client, $L_{\text{client}}$ with that of the printer, $L_{\text{printer}}$, in Figure 2.3. Let us give our attention to a few details that illustrate the behaviour of parallel composition.

In the first state both components are in their first states. The client is willing to get a tablet and it does not belong to the alphabet of the printer, therefore the client is free to take the tablet and move to state 3, while the printer stays put. This takes the composition to state 31.

On the other hand, both LTS:s have print in their alphabet and both accept it at the starting state. Therefore the composition can move to 22 using print. There the printer has $\tau$ transitions available, which it may take independently, while the client is still free to take its private tablet.

In state 24 we have an example of blocking. The client would be willing to take
2. Background

Figure 2.4: An example of an input to parallel composition that gives a result with exponentially more states. All actions are independent, so they may happen in any order. Therefore any combination of states of components is possible. Hence the result has $2^n$ states.

done, fail or tablet, but the printer does not accept done, so the composition does not have that option. The printer is not aware of tablet, so the client is free to do that.

State explosion

Parallel composition is the operation that manifests concurrency in our model. Therefore it is through parallel composition that the hardness of concurrency turns into trouble for our model.

To be more specific, the trouble comes in the form of the state explosion problem. The number of states in the parallel composition can be up to $|S_1| \cdot |S_2| \cdot \ldots \cdot |S_n| = \Pi_{i \in \pi} |S_i|$. If each component has a constant number $c$ of states, there are $n$ components and every transition label is unique to the component it appears in or invisible, the input has $cn$ states, while the composition has $c^n$ states. See Figure 2.4. This means there can be an exponential growth in the number of states.

While this is the worst case, the growth of the number of states tends to be quite devastating also in practice. Therefore much of verification research focuses on dealing with state explosion.

2.2.3 Renaming

We return to modeling a shared printer. For there to be any sharing, we need to have (at least) two clients. In the name of laziness we want to simply use the client we have twice instead of duplicating the definition.

There is a problem. Since the clients share their alphabet, they would interfere with each other when composed. Imagine if you were allowed to print if and only if every other user of the printer was printing at the same time. That will not do. Instead we shall create a renaming operator and rename their labels. This form of
Figure 2.5: The LTS $\Phi_{C(1)}(L_{\text{client}})$

renaming is from [29].

**Definition 4 (Renaming)**

Let $C$ be any set of pairs $(a, b)$, where $a \neq \tau \wedge b \neq \tau$. We shall denote the domain of $C$ with $D(C) = \{a|\exists b : (a, b) \in C\}$. Then $L$ with the renaming $C$ is the LTS $\Phi_C(L) = (S, \Sigma', \Delta', s)$, where

$$\Phi_C(a, b) \Leftrightarrow (a, b) \in C \lor a = b \notin D(C)$$

$$\Sigma' = \{b|\exists a \in \Sigma : \Phi_C(a, b)\}$$

$$\Delta' = \{(s, b, s') | (\exists a : (s, a, s') \in \Delta \land \Phi_C(a, b)\}$$

The operation replaces each label $a$ with a corresponding label $b$. Every transition whose label was not in $D(C)$ is left intact. Note that this operator may be used to rename a single label into multiple labels by having the same label $a$ appear in more than one pair.

You may notice that this will not do anything if $D(C) \cap \Sigma = \emptyset$. We do not rule that out to make it easy to apply the same renaming to many LTS:s where not all of them necessarily have any members of $D(C)$ in their alphabet.

We can now start manipulating our system with renaming. We will separate the clients by sub-indices. The function $C(i) = \{(\text{print}, \text{print}_i), (\text{fail}, \text{fail}_i), (\text{done}, \text{done}_i), (\text{tablet}, \text{tablet}_i)\}$ gives us the necessary renaming sets. The LTS $\Phi_{C(1)}(L_{\text{client}})$ is pictured in [Figure 2.5].

Next we must do something to the printer to connect it to both clients. Let us try the simplest way we can think of. It shall not know or care whose stuff it is printing, it shall simply accept any print order and send acknowledgment to anyone who is listening. We can accomplish this by renaming the actions in $L_{\text{printer}}$ into both names they have in the clients. That is $\{(\text{print}, \text{print}_1), (\text{print}, \text{print}_2)\} \subseteq C$ and so on for each label. This is pictured in [Figure 2.6].
Now we can put the pieces together and compute the parallel composition of two clients and a printer. Take a look at Figure 2.7. It should be immediately apparent that this picture is hard to read. How can we check, for example, that two clients are never printing at the same time from this?

Well, we could. Compared to real world problems this is still small. Sometimes one cannot avoid this kind of complexity. Luckily, in this situation, there is plenty of information irrelevant to our question in the picture. Therefore the information about the system relevant to our question can be depicted with something simpler.

### 2.2.4 Hiding

For example, when checking that both clients do not print at the same time, we do not really care why one of them abandons the printer. That information is just noise when thinking about this. We would be better off just knowing that something can happen that makes them stop printing, without knowing precisely what that something is.

Luckily we already have a concept for that: the invisible transition. We shall define an operator that allows us to turn labels invisible to declare that we no longer care about them.

**Definition 5 (Hiding)**

Let \( L = (S, \Sigma, \Delta, \delta) \) be an LTS and \( A \) be a set. Then \( L \) with \( A \) hidden, \( L \setminus A = (S, \Sigma \setminus A, \Delta', \delta) \) is an LTS where

\[
\Delta' = \{(s_1, a, s_2) \mid \exists b : (s_1, b, s_2) \in \Delta \land ((b \notin A \land a = b) \lor (b \in A \land a = \tau))\}
\]

Figure 2.8 shows our total LTS with the tablet events hidden. Sadly hiding does not reduce the number of states or transitions. This figure really is simpler from an information content viewpoint, but that does little to help mere human minds grasp
Figure 2.7: The LTS $\Phi_{C(1)\cup C(2)}(L_{\text{printer}}) || \Phi_{C(1)}(L_{\text{client}}) || \Phi_{C(2)}(L_{\text{client}})$. Note how confusing this is.

Figure 2.8: The LTS $(\Phi_{C(1)\cup C(2)}(L_{\text{printer}})) || (\Phi_{C(1)}(L_{\text{client}})) || (\Phi_{C(2)}(L_{\text{client}})) \setminus \{\text{tablet}_1, \text{tablet}_2\}$. Maybe a bit less confusing. At least if you are not a human being.
2. Background

Surely some of this tangle is unnecessary. We must find a smaller LTS that contains essentially the same information. To recognize that two LTSs contain the same essential information, we must decide what information we care about. To say that two state machines mean the same system, we must define what “mean” means. We must define the semantics we are using.

2.3 Semantics

In this section we shall present the idea of semantics and then define a number of them.

2.3.1 Sameness

The semantics of a state machine are its meaning. They tell how you wish to interpret an LTS. They tell what we are trying to say with it. This is how semantics allow us to say that two LTSs mean the same. Semantics essentially define an equivalence relation and split the set of LTSs into equivalence classes whose members have the same meaning.

There exist many widely used semantics for LTSs. To some extent the differences arise simply from historical reasons, such as different people having started using different mathematics to approach the same problem. Compare for example the fixed point thinking of CSP to the fundamentally LTS-based thinking we have used.

Other differences arise more directly from different needs. For example, sometimes it only matters that an LTS does nothing visible, at other times it is necessary to know whether that is because it is stuck or because it is in an infinite loop of invisible actions. Once decisions like these have been made, also the set of operators used to combine and manipulate LTSs must be chosen. It is usually required to behave in some desirable fashion in relation to the semantics, which causes different sets of operators to require different semantics even when the user wants to preserve the same information. For at least these reasons there has been no convergence into a single semantics.

Let us look at Figure 2.9. The LTSs $L_1$ and $L_2$ are clearly different, since they have a different number of states, but intuitively there is no difference in their behavior. We shall attempt to formalize this notion.

The first thing we require for any two LTSs to be equivalent by any semantics is that they have the same alphabet. Without this requirement supposedly equivalent LTSs could block different transitions of other components in parallel composition.
2. Background

2.3.2 Trace semantics

In the theory of traditional (Non-)Deterministic Finite Automata, the meaning of an automaton is its language; the set of strings it accepts. LTS:s lack the concept of accepting states, but we can simply consider all states accepting states, which makes all sequences of visible labels of consecutive transitions that start from the initial state into words of the language. With LTS:s this set is called the set of traces. The set of traces is a useful concept, but we shall see that it does not alone give a widely applicable semantics.

**Definition 6 (Traces)**

Let \( L = (S, \Sigma, \Delta, s) \) be an LTS. Then the set of its traces, \( \text{Tr}(L) \), is the set of all finite sequences of visible transitions of \( L \).

\[
\text{Tr}(L) = \{ \sigma \in \Sigma^* \mid \exists s' : s \xrightarrow{\sigma} s' \}
\]

The traces of an LTS can sometimes be concisely described using regular expressions. For example the language of the printer is \((\text{print} \mid \text{done} \mid \text{fail})^* (\text{print} \mid \varepsilon)\). This is however not always possible, since LTS:s may be infinite and therefore have irregular languages.

We can define the meaning of an LTS to be its set of traces. This approach is called trace semantics. We can formulate it as the trace equivalence relation.

**Definition 7 (Trace equivalence)**

Two LTS:s \( L_1 \) and \( L_2 \) are trace equivalent, \( L_1 \equiv_{Tr} L_2 \) if and only if \( \text{Tr}(L_1) = \text{Tr}(L_2) \) and they have the same alphabet.

Relations of the form \( A \equiv B \iff f(A) = f(B) \), that is, relations where two things are equivalent if and only if their some functional property is equal are always equivalence relations. Semantics acquired by such a mapping are called denotational...
semantics. Denotational semantics say that the essential behavior of any LTS can be encapsulated by some value from the domain of $f$. For trace equivalence the domain of $f$ is the set of pairs $\Sigma, A$, where $A \subseteq \Sigma^*$. We will be seeing many denotational semantics.

Unfortunately traces do not contain a sufficient amount of information for our purposes. For example, in Figure 2.10 both of the LTS:s have the same set or traces, $\text{Tr}(L_1) = \text{Tr}(L_2) = (\text{print done})^*(\text{print|e})$. The LTS $L_1$ is a perfect printer. It always succeeds and is then ready to print more. The LTS $L_2$, other the other hand, may suddenly stop doing anything; never sending any reply (it is dead) or accepting any further commands (it is locked). This kind of situation is called a deadlock.

Deadlocks essentially provide the answer to the question: can it get stuck? That question is frequently interesting and therefore the inability of traces to answer it is serious and a through treatment of deadlocks important. Their formal definition follows.

**Definition 8 (Deadlocks)**

Let $L = (S, \Sigma, \Delta, \hat{s})$ be an LTS. We define the set of deadlock traces, or simply deadlock states, of $L$

$$\text{Dl}(L) = \{ \sigma \in \Sigma^* | \exists s \in S : \hat{s} \xrightarrow{\sigma} s \land \forall a \in \Sigma \cup \{\tau\} : \neg(s \xrightarrow{a}) \}.$$  

We shall call the locked states $s$ in the definition deadlock states. We shall say that a state $s$ rejects a label $a$ when $\neg(s \xrightarrow{a})$. Then a deadlock state is a state that rejects the entire alphabet and the invisible action.

By applying the concept of denotational semantics, we can simply add deadlock traces to our semantics and get a new equivalence relation.

**Definition 9 (Tr-Dl equivalence)**

Two LTS:s $L_1, L_2 \in \text{LTS}$ are trace-deadlock equivalent, $L_1 \equiv_{\text{Tr,Dl}} L_2$ if and only if

$$\text{Tr}(L_1) = \text{Tr}(L_2) \land \text{Dl}(L_1) = \text{Dl}(L_2)$$

and they have the same alphabet.

This equivalence can discern the difference between the LTS:s $L_1$ and $L_2$ in Figure 2.10. However, there is another problem. To grasp it, we must discuss congruence.

### 2.3.3 Congruence

A single LTS alone does not model a concurrent system. We must compose them using the parallel composition operator. Sometimes making this composition properly requires use of the renaming and hiding operators.

Therefore, when we declare two LTS:s to be equivalent, we want them to behave equivalently when used in composition, renaming and hiding. This idea is formalized by congruence.
Definition 10 (Congruence)
Let \( \equiv \) be an equivalence relation and \( A = \{\square_1, \square_2, \ldots, \square_n\} \) be a set of operators over LTS:s and \( r_1, r_2, \ldots, r_n \) be their respective arities. Then \( \equiv \) is congruent with respect to \( A \) if and only if for any LTS:s \( L_a, L_b \) it holds that

\[
L_a \equiv L_b \Rightarrow \\
\forall i \in \pi : \forall L_1, L_2, \ldots, L_{r_i} \in LTS : \forall k \in \pi_i : \\
L_k \equiv L_a \Rightarrow \Box_i (L_1, L_2, \ldots, L_{r_1}, \ldots, L_{r_i}) \equiv \Box_i (L_1, L_2, \ldots, L_{r_1}, \ldots, L_{r_i})
\]

Congruence essentially says that if two LTS:s are equivalent, replacing one with the other in any expression must give an equivalent result. The definition requires it only from a single application of a single operator at a time, but since the results are always LTS:s, we can keep applying the definition recursively and get the requirement for any expression formed from operators in \( A \).

This allows us to manipulate LTS:s in a compositional manner. We can for example find a smaller but equivalent component LTS before computing parallel composition and save ourselves from generating the entire state space.

We can now see a problem in trace-deadlock equivalence: it is not congruent with respect to parallel composition.

Observe Figure 2.11. The LTS:s \( L_1 \) and \( L_2 \) have the same traces and deadlocks, but \( L_2 || L_2 \) can deadlock after \( c \), while \( L_1 || L_2 \) cannot. Therefore \( L_1 || L_2 \) and \( L_2 || L_2 \) are not trace-deadlock equivalent.

Since all machines must accept a shared event for it to be enabled, parallel composition can generate new deadlocks where none existed previously. On the other hand, since the composition does not deadlock as long as any component can take actions, deadlocks can also be removed. After any trace \( \sigma \) the composition is in a deadlock if all components are in stable states and some machine rejects all events that the others would accept. Therefore to compute the deadlocks after parallel composition, we must know what labels each machine that cannot reject \( \tau \) can re-
2. Background

Figure 2.11: LTS:s $L_1, L_2$ with the same deadlocks, but different deadlocks after substitution into $F(L) = L_1||L_2$.

ject after a given trace. This information is contained in the set of stable failures $Sf$.

Definition 11 (Stable Failures)

Let $L = (S, \Sigma, \Delta, s)$ be an LTS. Then the set of stable failures of $L$, $Sf(L)$ is defined

$$Sf(L) = \{(\sigma, A) | A \subseteq \Sigma \land \exists s : s \Rightarrow s \land \forall a \in A \cup \{\tau\} : \neg(s \xrightarrow{a})\}$$

In this name (and in LTS:s in general) the word stable means that the invisible action $\tau$ cannot be executed. Unstable states are excluded here because nothing can lock the invisible action and therefore there cannot be deadlocks with them present. The word failure comes from the fact that these states fail to execute any member of $A$. We can extract the deadlock traces of $L$ from its stable failures, since a deadlock is a trace that can lead to a state where no label can be executed.

$$Dl(L) = \{\sigma | (\sigma, \Sigma) \in Sf(L)\}$$

From the stable failures of the components we can also compute the stable failures of the parallel composition. The composition rejects a visible label $a$ if and only if one of the components rejects it.

When discussing results of parallel composition, it is often necessary to look at a string only as one of the components sees it. This is made fluent using the projection notation.
Definition 12 (Projection)
The projection of a string $\sigma = a_1a_2 \ldots a_n \in A^*$ to the set $B$, $\sigma \downarrow B$, is defined by

$$\sigma \downarrow B = b_1b_2 \ldots b_n, \text{ where } b_i = \begin{cases} a_i & : a_i \in B \\ \varepsilon & : a_i \notin B \end{cases}$$

Projection replaces any label in $\sigma$ not in $B$ by the empty string $\varepsilon$. Essentially it removes everything not in $B$ from $\sigma$. So it always holds that $\sigma \downarrow B \in A^*$.

With this we can state the stable failures of a composition as a function of the stable failures and alphabets of the components:

$$\text{Sf}(L_1 \mid L_2) = \{(\sigma, A) \mid \exists B_1, B_2 : (\sigma \downarrow \Sigma_1, B_1) \in \text{Sf}(L_1) \land (\sigma \downarrow \Sigma_2, B_2) \in \text{Sf}(L_2) \land \sigma \in (\Sigma_1 \cup \Sigma_2)^* \land A = B_1 \cup B_2\}$$

Consider Figure 2.11 again. The stable failure $(c, \{a, c\})$ belongs to $\text{Sf}(L_2)$ and the stable failure $(c, \{b, c\})$ belongs to $\text{Sf}(L_2)$. Therefore $(c, \{a, c\} \cup \{b, c\}) = (c, \Sigma)$ belongs to the stable failures of $L_2 \mid L_2$ and $c$ is a deadlock. For $L_1$, however, the only stable failures after $c$ are $(c, \emptyset)$ and $(c, \{c\})$, therefore there is no deadlock after $c$ in $L_1 \mid L_2$.

Since we can compute the stable failures of the result LTS of composition from the stable failures and alphabets of the component LTS:s, it is clear that if two component LTS:s have the same stable failures and alphabets, the result LTS:s will have the same stable failures and alphabets. Therefore stable failures are congruent with respect to composition.

In fact, it holds in general that if you can compute some set of properties of a result LTS from the same set of properties of the component LTS:s, the denotational semantics of those properties is a congruence with respect to the operators used. This is stated formally in Theorem 1.

**Theorem 1**
Let “$\equiv$” be the equivalence relation of some denotational semantics and $f : \text{LTS} \rightarrow X$ be a function that extracts the properties that define it. Then, “$\equiv$” is congruent with respect to some set of operators $A$ if for every operator $\Box \in A$ with arity $r$, there exists a function $g_\Box : X^r \rightarrow X$ such that for any LTS:s $L_1, L_2, \ldots, L_r$

$$f(\Box(L_1, L_2, \ldots, L_r)) = g_\Box(f(L_1), f(L_2), \ldots, f(L_r))$$
Proof: Let \( f, \Box, g \) be as assumed above. Then

\[
L_i \equiv L_i' \Rightarrow f(L_i) = f(L_i') \text{ implies that }
\]

\[
f(\Box(L_1, L_2, \ldots, L_i, \ldots, L_n)) = g(\Box(f(L_1), f(L_2), \ldots, f(L_i), \ldots, f(L_n))
\]

\[
= g(\Box(f(L_1), f(L_2), \ldots, f(L_i'), \ldots, f(L_n))
\]

\[
= f(\Box(L_1, L_2, \ldots, L_i', \ldots, L_n))
\]

and therefore

\[
\Box(L_1, L_2, \ldots, L_i, \ldots, L_n) \equiv \Box(L_1, L_2, \ldots, L_i', \ldots, L_n)
\]

Since this holds for an arbitrary operator \( \Box \) and an arbitrary parameter position of it, the equivalence “\( \equiv \)” is indeed congruent with respect to the set of operators \( A \).

q.e.d.

Now, trace-stable failure-equivalence is a denotational semantics with \( f(L) = (\Sigma, \text{Tr}(L), \text{Stf}(L)) \). Above we saw that stable failures of parallel composition can be computed from the stable failures of the arguments. Let us now look at traces.

\[
\text{Tr}(L_1||L_2) = \{ \sigma \in (\Sigma_1 \cup \Sigma_2)^* \mid \sigma \downarrow \Sigma_1 \in \text{Tr}(L_1) \land \sigma \downarrow \Sigma_2 \in \text{Tr}(L_2) \}
\]

Since trace-stable failure-equivalence is a denotational semantics with \( f(L) = (\Sigma, \text{Tr}(L), \text{Stf}(L)) \) and we can compute all three properties of a resulting LTS from the properties of the arguments LTS:s (the alphabet is trivially the union of the two component alphabets), Theorem 1 tells us that it is congruent with respect to parallel composition \([29]\).

### 2.3.4 Divergence

Tr-Sf semantics are already a perfectly serviceable way to look at LTS:s. As we said before, there is no “one true way” to compare LTS:s. There is nonetheless one more potentially interesting phenomenon that we can grasp by following our chain of reasoning further. Rest your eyes on the unfortunate state of affairs in Figure 2.12.

The premise here is obvious: we want a puppy. We must find an LTS to give us one using only the information our semantics can give us. The LTS:s \( L_1 \) and \( L_2 \) are equivalent as far as we can see. Unfortunately, one of them may choose to never give us our puppy and hide its treachery by doing something else forever. Since it never reaches a stable state, the branch of betrayal is invisible to stable failures.

We can deal with this by adding another component to our semantics: **divergences.**

Definition 13 (Divergences)

Let \( L = (S, \Sigma, \Delta, \delta) \) be an LTS. Then the set of divergences of \( L \), \( \text{Div}(L) \), is defined
Figure 2.12: Two LTS:s that Tr-Sf semantics sees as equivalent. There is however a world of difference between them to us, since one of them always gives us a puppy and the other one may never do it.

by

\[ \text{Div}(L) = \{ \sigma \mid \exists s : \hat{s} \xrightarrow{\sigma} s \wedge s \xrightarrow{\tau^{\omega}} \} \]

, where \( s \xrightarrow{\tau^{\omega}} \) means that it is possible to execute an infinite number of \( \tau \) transitions starting from \( s \).

The divergences of parallel composition can be computed from the arguments. The parallel composition diverges for any trace that it can execute and that allows one of the arguments to diverge.

\[ \text{Div}(L_1|L_2) = \{ \sigma \mid \sigma \downarrow \Sigma_1 \in \text{Tr}(L_1) \wedge \sigma \downarrow \Sigma_2 \in \text{Tr}(L_2) \wedge \\
\quad (\sigma \downarrow \Sigma_1 \in \text{Div}(L_1) \vee \sigma \downarrow \Sigma_2 \in \text{Div}(L_2)) \} \]

Therefore divergences are congruent with respect to parallel composition. We now have a traces-stable failures-divergences -semantics, which is congruent with respect to parallel composition.

Even better, traces can be computed from stable failures and divergences. For every trace \( \sigma \), it diverges or does not (or both, with different states). If it diverges, we get it from the divergences. If it does not, we get it from the stable failures. The initial state the trace \( \sigma \) takes you to may not be stable, but if it does not diverge, there is only a finite chain of invisible transitions from it. We can follow that chain to the end without affecting the trace.

\[ \text{Tr}(L) = \text{Div}(L) \cup \{ \sigma | (\sigma, \emptyset) \in \text{Sf}(L) \} \]

In a similar manner it can be shown that this equivalence is a congruence with respect to renaming \[35\]. For hiding it is a congruence only for finite LTS:s. That
can be fixed by adding an additional component, the infinite traces, to the model, but since we have excluded infinite LTSs from our presentation we will not need them.

The semantics we have described is called the Chaos-Free Failures Divergences (CFFD) semantics. The chaos-free int the name contrasts CFFD with the failures-divergences model in which a divergence is considered chaos, after which behavior is undefined.

**Definition 14 (Chaos-Free Failures Divergences equivalence)**

Two LTSs $L_1, L_2 \in \text{LTS}$ are CFFD-equivalent, $L_1 \equiv_{CFFD} L_2$ if and only if $\text{Sf}(L_1) = \text{Sf}(L_2)$ and $\text{Div}(L_1) = \text{Div}(L_2)$ and they have the same alphabet.

### 2.3.5 CFFD-preorder

Sometimes we may want to have a specification that leaves some details unspecified or explicitly allows several alternate ways of doing something. If such a specification is formulated as an LTS, we would like to be able to say that another LTS implements it.

It turns out that this idea of an implementation relation is captured by the CFFD-preorder, at least when you are interested in the properties CFFD preserves.

**Definition 15 (CFFD-preorder)**

The LTS $L_1$ is CFFD-smaller or equivalent to the LTS $L_2$, $L_1 \preceq L_2$ iff $\Sigma(L_1) = \Sigma(L_2) \land \text{Div}(L_1) \subseteq \text{Div}(L_2) \land \text{Sf}(L_1) \subseteq \text{Sf}(L_2)$

At first it may seem like this does not require the implementation to do anything, but you must note that the implementation cannot refuse to do things the specification does not refuse. For example, in Figure 2.13, the LTS $L_3$ does not implement $L_1$, because it rejects error in the initial state, which breaks the inclusion of stable failures. The CFFD-preorder is elaborated on extensively in [25].

### 2.4 Bisimilarity

When creating equivalences that lose all irrelevant information, one must choose what is relevant and what is not. This has lead to the plethora of lossy semantics. At the other end of the spectrum, losing no information anyone could possibly care about, there is more agreement. Strong Bisimilarity (SB), or simply Bisimilarity as it is called here, is fairly universally used in that role [26, 27, 8].

Bisimilarity essentially requires that for any state in an LTS there must be a state in the other LTS that behaves just like it. The idea “behaves just like it” is captured by the bisimulation relation.
2. Background

![Diagram of three LTSs](https://via.placeholder.com/150)

**Figure 2.13**: An example of CFFD-preorder. The specification \( L_1 \) allows divergence, deadlock or recovery after an error. The LTS \( L_2 \) implements it, \( L_2 \preceq L_1 \), but always recovers. To make the LTS \( L_3 \) comparable to others, let \( \text{error} \) be a part of its alphabet despite not appearing in any transition. Even then, LTS \( L_3 \), which never makes errors, does not implement \( L_1 \), because it has an extra stable failure, \( \{ \varepsilon, \{ \text{error} \} \} \).

**Definition 16 (Bisimulation)**

A relation \( \sim \subseteq S_1 \times S_2 \) is a bisimulation if and only if for all states \( s_1 \in S_1, s_2 \in S_2 \) it holds that \( s_1 \sim s_2 \) implies that:

\[
\forall s'_1 \in S_1, a \in \Sigma \cup \{ \tau \} : s_1 \xrightarrow{a} s'_1 \Rightarrow \exists s'_2 \in S_2 : s_2 \xrightarrow{a} s'_2 \land s'_1 \sim s'_2
\]

\[
\forall s'_2 \in S_2, a \in \Sigma \cup \{ \tau \} : s_2 \xrightarrow{a} s'_2 \Rightarrow \exists s'_1 \in S_1 : s_1 \xrightarrow{a} s'_1 \land s'_1 \sim s'_2
\]

The bisimulation relation essentially says that \( s_1 \) and \( s_2 \) behave the same if they can do the same things and end up in similar states when they do them. We do not require \( s_1 \) and \( s_2 \) to be from the same or from different LTSs. It is legal to set \( S_1 = S_2 \) and consider bisimulation inside a single LTS. Armed with this relation it is quite simple to define bisimilarity between LTSs.

**Definition 17 (Bisimilarity)**

Two LTSs \( L_1 \) and \( L_2 \) are bisimilar if and only if they have the same alphabet and there exists a bisimulation \( \sim \) between them such that \( \hat{s}_1 \sim \hat{s}_2 \).

Bisimilarity is a very strong equivalence. It implies all equivalences used in practice, including CFFD. That means that \( L_1 \equiv_{SB} L_2 \Rightarrow L_1 \equiv_{CFFD} L_2 \).

There are two notable semantics closely related to bisimilarity: weak bisimilarity and branching bisimilarity. Weak bisimilarity is defined mostly like bisimilarity, with the main difference being that it uses \( \Rightarrow \) instead of \( \rightarrow \), which means it ignores
2. Background

invisible transitions and therefore loses sight of divergences [22]. Branching bisimilarity on the other hand can see invisible transitions, but only when they lead to different futures [3]. We focus on CFFD and therefore will not expand on these concepts.

2.5 Reduction

Let us explore the use of the above equivalences in reducing the size of the system. We will do that in the context of visually verifying our printer system example.

We begin by trying to find LTS:s equivalent to the one in Figure 2.8. First, in Figure 2.14 we have the smallest LTS bisimilar to the original one. The picture clears up nicely even when only information we purposely hid is lost.

This figure is not that readable yet. Bisimilarity is too strong; it keeps too much information. So let us turn to CFFD equivalence. We have no algorithm for CFFD minimization, but we can use the method of Valmari [35] to reduce the size of the system while staying CFFD equivalent.

We have done that in Figure 2.15. It should be apparent that the simplification is magnificent. From this view it is simple to see that the clients truly do not use the printer simultaneously. Throwing away the details of the tablet transitions is instrumental to this reduction.

You may notice some peculiarities of CFFD, however. For example, the fact that one client may get a tablet, after which it will never request printing again, is not really visible here. That phenomenon boils down to the fact that the system may

Figure 2.14: The smallest LTS bisimilar to the one in Figure 2.8.
Figure 2.15: An LTS that is CFFD equivalent to our printer system with tablets hidden from Figure 2.8, but considerably smaller. This LTS was computed using the acceptance graph technique from [35]. The program used was TVT [38].

begin refusing the other print. Unfortunately CFFD cares about refusals only in stable states and with the tablet transitions turned invisible, the states where that new refusal would appear are not stable, so CFFD does not care.

We could use another, less lossy equivalence to keep that information if we wanted to, but we can also simply change our point of view. We are worried about what happens when one client gets a tablet and the other one does not. So let us hide all but the tablet actions. The result can be seen in Figure 2.16.

Once more we have an image small enough to be read by a human. It may not be even remotely easy to interpret in terms of the unreduced system. Luckily that is not what we are here to do. We wanted to make sure the traces, deadlocks and livelocks are as they should and CFFD has preserved them for us.

We can find a problem. Let us walk through the image. The initial state shows that stuff can go on indefinitely without any tablets. That is correct. Then, after a single tablet action we come to states three and four. Here something can still go on indefinitely. That should be the other client printing. This seems good.

Less good is that there are \( \tau \)-transitions to states five and two, which only accept tablet events. Since printing is invisible and there is no way to perform invisible actions here, that means that no one can print. So it is possible for us to reach a point where nobody can print although one of the clients does not have a tablet yet. This is an error.
Figure 2.16: A CFFD reduced view of our printer system with only the tablet actions visible.

We have thrown away too much information to figure out what exactly went wrong. Let us keep print1 visible. The result is in Figure 2.17. This is considerably larger, but still manageable. We were interested in states that accept only tablet events. Those would be states two and six.

They are both erroneous so we investigate what led to them. The middle path seems to have the most information. Here the error is possible after at least one print1 event followed by a single tablet1 event. Referring to the client specification in Figure 2.2 we can find only one tablet transition that requires a print before it: the one directly after print.

From this the error becomes clear. If the client gets a tablet after it has sent the print event, but before receiving either done or fail, it refuses them and makes the printer stuck. This is not a mistake in our printer system. It is our model of the client that is at fault. You cannot stop a printer from failing simply by refusing to acknowledge the failure even if you have a tablet and do not care.

To fix the client, we must make it accept the printer’s messages after it gets a tablet if it is expecting them. See Figure 2.18.

Let us look at what happens when we replace the old client with this one. Figure 2.19 shows the system with the tablet events hidden and Figure 2.20 shows only the tablet events. The possibility of rejecting all but a tablet is gone. This concludes our Visual Verification example.
2.6 Variables

It is time to consider the advantages of variables in transition systems, expand our model to support them and understand the relationship between the expanded model and the simple one.

2.6.1 Motivation

When using state machines to describe computation, variables are encoded into states. For example, consider a lossy channel with a capacity of one bit. To model that the sender may send a one or a zero, we need separate transitions \textit{send0} and \textit{send1}. They must lead to separate states for the channel to know whether it must relay a one or a zero. This is depicted in Figure 2.21.

To simplify this, we can model variables using separate LTS:s and use composition to attach them to the system.

Figure 2.22 shows the bit channel using this technique. The advantage of this trick may not be apparent in such a small system, so we have also presented a counter variable in Figure 2.23. You could compose it with a system that uses \textit{inc} and \textit{dec}. Any \textit{dec} transitions would be disabled whenever the counter was at zero and \textit{inc} transitions likewise whenever it was at the maximum. Separating the potentially
very large counter would make such a system much clearer.

We see that LTS:s can capture the notion of variables, but this comes at the cost of adding several new LTS:s to parallel compositions. The added LTS:s may be fairly independent of the others, contributing greatly to the state explosion problem.

The full power of state machines is not truly required to describe variables. Variables have constraints and regularities in their behavior, which should allow us to expend less effort on them. By modeling them with full state machines, we lose this extra information.\footnote{It is not actually inconceivable that the regularity could be detected and taken advantage of somehow, perhaps with automorphisms, but we are not aware of any practical algorithm that could do this.}

Therefore it seems advisable to make variables a full member of our model to allow us to exploit the constraints we know for their behavior. A system defined using a formalism like this may look something like Figure 2.24.

This is the motivation for the Relational Labeled Transition System (RLTS) formalism presented by Kokkarinen and Valmari \cite{kokkarinen, valmari}.

\subsection{RLTS}

Valmari and Kokkarinen define variables differently \cite{kokkarinen, valmari}. Kokkarinen uses global variable names, whereas Valmari simply gives every state a tuple of variables. Valmari does use variable names in pictures and explanations, though. They are simply interpreted into tuple positions when moving to the formalism. We adopt this approach, because global variables have a higher risk of having values also when not needed, expanding the state space unnecessarily.

Where does the word relational in the name RLTS come from? A transition in
Figure 2.19: A CFFD reduced and tabletless view of the system with the fixed client.

Figure 2.20: A view of the fixed system, with only the tablet actions visible. The error we saw in Figure 2.16 is gone.
Figure 2.21: A lossy one bit channel.

Figure 2.22: A lossy one bit channel with the bit as a separate machine.

an LTS may represent arbitrary computation on the part of the entity. Therefore it may change its variables in any way. It is expected, however, that the resulting values of the variables somehow depend on the previous values of the variables. Since transitions also describe communication, there must also be some way to communicate values. For this, each transition shall have parameters. It is expected that the end result may depend on these parameters and these parameters may depend on the variables in the tail state.

All these dependencies can be described using a relation. It is a general enough notion to describe any dependency between the initial values, parameters and end values.

Let us start figuring out the components of a definition for this concept. Each state shall have a set of variables. Since they shall be nameless they will be identified by their position in a tuple. Therefore we must associate with each state the kind of tuple that stores its variables.

There shall be a global set of types $\Omega$. Each type is a set of possible values for a variable of the type. That is, for every type $A \in \Omega$ a variable of type $A$ can have the value $v$ if and only if $v \in A$. 
Figure 2.23: A simple counter.

Figure 2.24: A lossy one bit channel using a notation with variables. Here send?x means we save the parameter of send into x, while relay!x means we use x as the parameter of relay. The x: bit in state two means that it has a single variable called x of type bit. On the right we have the counter of Figure 2.23 presented as an RLTS. The conditions in square brackets before the action name are guards: requirements that must be fulfilled for the transition to be enabled. The square brackets after the action name describe how the action changes variables. The side with ";" represents the values after the transition.

**Definition 18 (RLTS)**

A Relational Labeled Transition System (RLTS) is a tuple $(S, \mathcal{T}, \Sigma, \Delta, \hat{s}, \hat{v})$, where

- $S$ is the set of states
- $\mathcal{T}: S \to \Omega$ gives the types of variables for each state
- $\Sigma$ is the set of possible messages, that is, labels with parameters.

We will write these as $\text{label}<p_1,p_2>$.

- $\hat{s} \in S$ is the initial state
- $\hat{v} \in \mathcal{T}(\hat{s})$ is the initial values of the variables of the initial state.

Finally

$$\Delta \subseteq \{(s, R, s') | s, s' \in S, R \subseteq \mathcal{T}(s) \times \Sigma \times \mathcal{T}(s')\}$$

is the set of transitions. This says that when we are in a state $s \in S$, with the variables having certain values $v \in \mathcal{T}(s)$, taking a specific transition $a<1>$, we may move to state $s'$ with its variables getting the values $v'$ if and only if there is $R$ such that $(s, R, s') \in \Delta$ and $(v, a<1>, v') \in R$.

This definition is very general. Its full power can easily be necessary, however. A Turing complete programming language with a source of true randomness would be hard to fit in any more restricted model. If no nondeterminism is needed, we could
restrict the relation into being a partial function from some initial values and input parameters to unique resulting variable values and output parameters.

In practice we will specify the relation using some formalism that restricts us to relations of specific forms, specified with some well defined expressions. Different restrictions allow different tools for manipulating the expressions that define the relations.

It is good to find whatever results apply to this general form, since they will be available to any specific form. Kokkarinen has results on both the general form and a restriction he calls a variable transition system (VTS). In VTS:s only relations that can be described using an associated expression language can be used.

All our RLTS examples use Kokkarinen’s expressions (and are therefore VTS:s). The expressions enable arithmetic and comparisons with the past and resulting values of variables as operands. For a formal treatment see Kokkarinen [20].

Let us explore the mapping between this notation and the relations. We will start by looking at the bit channel in Figure 2.24. The transition send creates a relation where the only requirement is that the value of x must be equal to the single parameter after the transition. Since the initial state has no variables, the tuple only has two members: the label and the resulting value of the variable.

\[ R = \{(\text{send}<p>, p) \mid p \in \{0, 1\}\} \]

The relay transition on the other hand only requires that the single parameter equals the single variable in the starting state.

\[ R = \{(x, \text{relay}<x>) \mid x \in \{0, 1\}\} \]

For an example with guards, parameters and variable manipulation, look at Figure 2.25. In particular, here is the relation of the fux transition.

\[ R = \{(a, a, \text{fux}<a: >, a: , b: ) \in \{0, 1\}^2 \times \Sigma \times \{0, 1\}^2 \mid b := a + a\} \]

\[ = \{(1, 1, \text{fux}<0>, 0, 0), \]
\[ (1, 1, \text{fux}<1>, 1, 0), \]
\[ (0, 0, \text{fux}<0>, 0, 0), \]
\[ (0, 0, \text{fux}<1>, 1, 0) \} \]

Our architecture should support plugging in arbitrary ways to specify the relation
2. Background

and arbitrary tools to handle them. We should of course also have either a very simple or a widely usable form that could be used by default.

2.6.3 RLTS:s and LTS:s

We constructed the left RLTS in Figure 2.24 to have a more succinct way to present the LTS in Figure 2.21. We think they do the same thing. We will now formalize their equivalence by presenting a way to construct and equivalent LTS for any RLTS.

We can construct such an LTS by taking every state with variables in the RLTS and creating a separate state for any combination of values the variables could have. Then we can describe the relationships between the variables using plain LTS transitions. Following the presentation of [20] this transformation is called unfolding.

Definition 19 (Unfolding)

Let $M = (S, T, \Sigma, \Delta, s, \delta, \bar{v})$ be an RLTS whose types are finite. Then the unfolding of $M$ is the LTS $U(M) = (S', \Sigma, \Delta', \delta')$, where

$S' = \{(s, v) | s \in S \land v \in T(s)\}$

$\Delta' = \{((s_1, v_1), a, (s_2, v_2)) | \exists R : (s_1, R, s_2) \in \Delta \land (v_1, a, v_2) \in R\}$

$\delta' = (\delta, \bar{v})$

Kokkarinen [20] shows that for several concepts explored before, extending the concept from LTS:s to RLTS:s is equivalent to requiring the LTS definition of the concept from the unfolding of the RLTS. This is shown at least for parallel composition, hiding and CFFD equivalence. These results allow us to manipulate RLTS:s as far as we can and then unfold them if we want to apply tricks we know only for pure LTS:s.
In accordance with our choice to leave infinities out of our presentation, we require each type $A \in \Omega$ of the RLTS to be finite for it to be unfolded. Otherwise unfolding may produce infinite LTS:s.

2.7 Synchronization rules

Synchronization rules are a compact way to describe composition of state machines. In this section we will present them for LTS:s and RLTS:s.

2.7.1 For LTS:s

As we saw above, we can use parallel composition, relational renaming and hiding to wire together components in arbitrary ways. The expressions we need to do this do however grow a bit messy fairly quickly. In addition, implementing these steps separately is not very efficient.

For these reasons the synchronization rule concept as presented in [29] is handy. It allows us to do a complex synchronization in one step.

The idea is that we have a set of rules that map sets of original transitions into transitions in the result system. They are of the form: the result taking the transition $send0$ equals $L_{\text{channel}}$ taking the transition $send$ and $L_{\text{bit}}$ taking the transition $set0$.

For each transition in the result, there must be a rule that tells what transition each of the components takes. When a component does nothing, we mark it with a "-". There cannot be a rule where no component does anything, however, since we do not want to be able to create new transitions out of thin air. On the other hand, the invisible actions of the components must be able to keep running independently.

Definition 20 (Synchronization rules for LTS:s)

Let $\forall i \in \pi : L_i = (S_i, \Sigma_i, \Delta_i, \delta_i) \in \text{LTS} \land "-" \notin \Sigma_i$. Let $\Sigma$ be a set such that $\tau \notin \Sigma \land "-" \notin \Sigma$. Let $\Upsilon \subseteq (\times_{i \in \pi} (\Sigma_i \cup \{"-"\}) \setminus \{"-"\}^n) \times (\Sigma \cup \{\tau\})$ be a set of synchronization rules.

Then $\Upsilon(L_1, L_2, \ldots, L_n)$ is the reachable part of the LTS $(S, \Sigma, \Delta, \delta)$, where

- $S = \times S_i$
- $\Delta = \{(s_1, s_2, \ldots, s_n), a, (s'_1, s'_2, \ldots, s'_n) \in S \times (\Sigma \cup \{\tau\}) \times S | \exists (a_1, a_2, \ldots, a_n; a) \in \Upsilon : \forall i \in \pi : (a_i = "-" \land s'_i = s_i) \lor (s_i, a_i, s'_i) \in \Delta_i)\} \cup \{(s_1, s_2, \ldots, s_n), \tau, (s'_1, s'_2, \ldots, s'_n) | (s_i, s'_i) \in \Delta_i\}$
- $\delta = (\delta_1, \delta_2, \ldots, \delta_n)$

In [29] it is shown that any expression formed with composition, renaming and hiding can be implemented using synchronization rules and vice versa; they are
2. Background

<table>
<thead>
<tr>
<th>$L_{\text{printer}}$</th>
<th>$L_{\text{client}}$</th>
<th>$L_{\text{result}}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>print</td>
<td>print</td>
<td>print$_1$</td>
</tr>
<tr>
<td>print</td>
<td>-</td>
<td>print$_2$</td>
</tr>
<tr>
<td>done</td>
<td>done</td>
<td>done$_1$</td>
</tr>
<tr>
<td>done</td>
<td>-</td>
<td>done$_2$</td>
</tr>
<tr>
<td>fail</td>
<td>fail</td>
<td>fail$_1$</td>
</tr>
<tr>
<td>fail</td>
<td>-</td>
<td>fail$_2$</td>
</tr>
<tr>
<td>-</td>
<td>tablet</td>
<td>$\tau$</td>
</tr>
<tr>
<td>-</td>
<td>-</td>
<td>$\tau$</td>
</tr>
</tbody>
</table>

Figure 2.26: Synchronization rules to implement the expression $(\Phi_{C(1)\cup C(2)}(L_{\text{printer}})||\Phi_{C(1)}(L_{\text{client}})||\Phi_{C(2)}(L_{\text{client}})) \setminus \{\text{tablet}_1, \text{tablet}_2\}$, which is depicted in Figure 2.8.

equivalent. For example, Figure 2.26 shows the rules to implement the printer system studied earlier.

2.7.2 For RLTS:s

We shall use the same approach as with state machines themselves and see synchronization rules with variables essentially as a quick way to write rules for each possible value of the parameters.

There are no shared variables in the RLTS model, which means that the variables of the components are independent. Therefore the tuple of variable types of a state of the composition must contain all the variable types of the states of the components.

Since our formulation of RLTS:s populated the alphabet $\Sigma$ with the events with the parameter values written in, the above definition for synchronization rules works for them with fairly small changes, although formalizing the way the compound relations are formed from the component relations is non-trivial. Conceptually applying the LTS based definition to RLTS:s just requires you to write out each possible value combination as a separate rule.

Giving the synchronization rules $\Upsilon$ explicitly like that would defeat the point of RLTS:s. Therefore $\Upsilon$ should be generated using some formalism for specifying relations, similarly to the relations on the transitions themselves.

The user needs to be able to express relations like: “when $L_x$ does $\text{push}<a>$ and $L_y$ does $\text{pull}<b>$, the composition is seen to do $\text{move}<a+b>$” which would formally mean that $\forall a, b : \text{push}<a> \in \Sigma_1 \land \text{pull}<b> \in \Sigma_2 \land \text{move}<a+b> \in \Sigma \Rightarrow (\text{push}<a>, \text{pull}<b>, \text{move}<a+b>) \in \Upsilon$.

As with RLTS:s, any particular implementation is free to use whatever restricted form of relations it wishes. In practice this will likely take the form of guards and simple expressions like the ones in VTS:s.
2. Background

38

2.8 Composition machinery

Any congruent semantics can be used to try and avoid the state explosion problem by reducing intermediate systems. Often, however, this is not sufficient.

Another approach is to alter the parallel composition itself to avoid generating the full state space in the first place. This section shall present some of these techniques.

2.8.1 Stubborn sets

As mentioned in our state explosion worst case example, the state explosion is worst when many machines can take actions entirely independently of each other. In some cases this very independence can be used to ignore parts of the state space without losing information. The techniques that do this are often called Partial Order methods \[7\].

We begin by pondering the nature of independent actions. Please examine Figure 2.27. There the actions eat and derivate are independent. That is, executing eat in no way affects your capability to execute derivate or vice versa. Because of this, you can follow either the path eat derivate or derivate eat and the result is equivalent.

If we only care about where we end up (if we only want to know if deadlocks exist, for example), this allows us to ignore one of the paths to state four. Of course, if we care about traces, we cannot do this, since we would be ignoring a trace. We can do this to invisible actions, however, and to actions that will be hidden eventually.

First, we need some terminology to talk about actions that are hidden. Sometimes, to ease discussions like this, authors always have a set of invisible actions and hiding just moves members from the visibles to there. Since we did not do that, we need to assume some context.

In the following we are interested in the equivalence of \(L\) and our pruned LTS \(\tilde{L}\) only in an expression \(L \setminus A\). That is, we will at all times know that some actions, that are still visible to us, will be hidden before we actually look at the result. We will denote the set of these invisible actions by \(\Sigma_I = (\Sigma \cap A) \cup \{\tau\}\) and the visible actions by \(\Sigma_V = \Sigma \setminus A\). Now we can say that it is okay to reorder invisible actions, but not visible ones.

We must also be careful not to leave out other branches that may occur on the way. See Figure 2.29. There the paths eat derivate and derivate eat still lead to the same state, but only the path eat derivate contains the chance to escape. Therefore we must not ignore it, but can still ignore the other one.

Clearly mere independence is not sufficient; we need a more precise concept to manage the pruning. We shall use the concept of stubborn sets as presented by Valmari \[30\].
2. Background

Figure 2.27: A system where the actions \textit{eat} and \textit{derivate} are independent.

Figure 2.28: A system where the actions are independent and invisible.

Figure 2.29: A system where the actions are independent, but only one of the paths is ignorable.
The idea is that in each state we have labels that must be expanded now to not ignore anything interesting and labels that we will have an equivalent chance to expand later.

To put it more mathematically, for each state $s$ there exists a set of action labels $\tilde{A}(s) \subseteq \Sigma$ and only the transitions whose labels belong there are expanded. For this to work, $\tilde{A}$ must be a stubborn set generator.

**Definition 21 (Stubborn set generator)**

Let $L = (S, \Sigma, \Delta, \tilde{s})$ be an LTS. The function $\tilde{A} : S \rightarrow 2^\Sigma$ is a stubborn set generator for it if and only if for every $s_0, s_1, \ldots, s_n, s'_0, s'_1, \ldots, s'_n \in S$, $a \in \tilde{A}(s_0)$ and $b_1, b_2, \ldots, b_n \in \Sigma \setminus \tilde{A}(s_0)$ conditions $\tilde{A}0$, $\tilde{A}1$ and $\tilde{A}2$ hold, where

$\tilde{A}0$ If any action is enabled in $s_0$, some action of $\tilde{A}(s_0)$ is enabled in $s_0$

$\tilde{A}1$ If $s_0 \xrightarrow{b_1b_2\ldots b_na} s_n$ then $s_0 \xrightarrow{a b_1b_2\ldots b_n} s_n$.

$\tilde{A}2$ If $s_0 \xrightarrow{a} s'_0$ and $s_0 \xrightarrow{b_1b_2\ldots b_n} s_n$ then there exists $s'_n$ such that $s'_0 \xrightarrow{b_1b_2\ldots b_n} s'_n$ and $s_n \xrightarrow{a} s'_n$.

A set generated by a stubborn set generator is a stubborn set.

Condition $\tilde{A}0$ is basically a sanity check. Without it the empty set would be a valid stubborn set. It makes sure that no new deadlocks are created by the pruning.

Conditions $\tilde{A}1$ and $\tilde{A}2$ essentially require that all the enabled actions in the stubborn set are in a certain sense independent of the actions outside it. They also ensure that exploring only paths that start with labels in the stubborn set won’t make us ignore branches like escape in Figure 2.29.

For example, let us leave out eat from the stubborn set of the initial state $s_1$ in Figure 2.29. In that case, condition $\tilde{A}0$ requires that we must have derivate in the stubborn set. Then, if escape $\notin \tilde{A}(s_1)$, condition $\tilde{A}2$ would require the existence of the path derivate eat escape. On the other hand, if escape $\in \tilde{A}(s_1)$, condition $\tilde{A}1$ would require the existence of the path escape eat. Neither option is valid, therefore we cannot leave eat out of the stubborn set.

These conditions guarantee that we can safely ignore paths that start with labels outside the stubborn set, since we can do them later. This allows us to create a reduced LTS.

**Definition 22**

A stubborn set generator for $L$ induces a reduced LTS $\tilde{L} = (\tilde{S}, \Sigma, \tilde{\Delta}, \tilde{s})$, where $\tilde{S} \subseteq S$ and $\tilde{\Delta} \subseteq \Delta$ are as small as possible while fulfilling the following conditions:

2Here expanding means inclusion in the resulting graph. The term comes from the graph search nature of the actual algorithm.

3The definition in [30] includes a fourth condition for infinite sequences, but it is only required when the LTS can be infinite.
2. Background

- \( s \in \tilde{S} \)
- If \( s \in \tilde{S} \), \( s \xrightarrow{a} s' \) and \( a \in \tilde{A}(s) \), then \( s' \in \tilde{S} \) and \( (s, a, s') \in \tilde{\Delta} \)

Henceforth we shall denote \( s \xrightarrow{a} s' \) and \( a \in \tilde{A}(s) \) by \( s \xrightarrow{a} s' \) and use double dots similarly in other notations to say that they refer to the reduced LTS.

The conditions \( \tilde{A}_1 \) and \( \tilde{A}_2 \) require a global view; they look at arbitrarily long executions. Luckily when components have actions in their alphabet that some other components do not, we can use a sufficient condition that uses only local information.

**Theorem 2**

Let \( \forall i \in \pi : L_i = (S_i, \Sigma_i, \Delta_i, \hat{s}_i) \in \text{LTS} \). Let \( L = \|_{i \in \pi} L_i \). If the function \( \tilde{A} : S \rightarrow 2^\Sigma \) satisfies the following conditions and \( \tilde{A}_0 \) in \( s = (s_1, s_2, \ldots, s_n) \), it also satisfies \( \tilde{A}_1 \) and \( \tilde{A}_2 \) in \( s \).

- If \( a \in \tilde{A}(s) \) and \( \neg(s \xrightarrow{a}) \) then there exists \( j \in \pi \) such that \( a \in \Sigma_j \land \neg(s_j \xrightarrow{a_j}) \land \text{en}_j(s_j) \subseteq \tilde{A}(s) \)
- If \( a \in \tilde{A}(s) \) and \( s \xrightarrow{a} \) then for all \( j \in \pi \) either \( a \notin \Sigma_j \) or \( \text{en}_j(s_j) \subseteq \tilde{A}(s) \)

These conditions still ensure only deadlock equivalence. For the pruned version to satisfy other equivalences, the stubborn sets must fulfill more conditions. Valmari [30] gives conditions for several equivalences. We will present those needed for trace equivalence and CFFD equivalence.

First we deal with the visible and invisible actions. The stubborn set method reorders the labels in the stubborn set relative to the labels outside it. Therefore if any visible, enabled action is in the stubborn set, all visible actions must be there to ensure they are not reordered with respect to each other.

\( \tilde{A}_3 \) For every \( s \in \tilde{S} \): \( \Sigma_V \cap \text{en}(s) \cap \tilde{A}(s) = \emptyset \) or \( \Sigma_V \subseteq \tilde{A}(s) \)

Alas, that is not enough. There is a problem with loops. If expanding only actions in the stubborn set of each state we encounter never causes some action to enter the stubborn set, that action never gets expanded. Figure 2.30 shows a simple example of this.

Since we are doing mathematics at the moment, we can avoid this simply by saying that it isn’t allowed to happen.

\( \tilde{A}_4 \) \( \forall s \in \tilde{S} : \forall a \in \text{en}(s) : \exists s' \in \tilde{S} : s \xrightarrow{a} s' \land a \in \tilde{A}(s') \)
In practice this condition is a bit troublesome, since it isn’t local. The check can be done in $O(|\hat{S}||\hat{A}|)$ time [31], however, which isn’t too bad. The algorithm uses Tarjan’s algorithm to detect terminal components in which some action is enabled but never a member of the stubborn set.

Valmari [30], [31] shows that if $\hat{A}0$, $\hat{A}1$, $\hat{A}2$, $\hat{A}3$ and $\hat{A}4$ hold, then $\text{Tr}(L) = \text{Tr}(\hat{L})$.

CFFD equivalence preserves also divergences and stable failures. To preserve them in stubborn set reduction, we need two more conditions. See Figure 2.31 and Figure 2.32 for some justification.

\textbf{Å5} For every $s \in \hat{S}$ if $\Sigma_I \cap \text{en}(s) \neq \emptyset$, then $\hat{A}(s) \cap \Sigma_I \cap \text{en}(s) \neq \emptyset$

\textbf{Å6} For every $s_1, s_2, \ldots \in \hat{S}, a_1, a_2, \ldots \in \Sigma$ if $s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots$, then for some $i > 0$ $\Sigma_V \subseteq \hat{A}(s_i)$

\textbf{Theorem 3}

If $\hat{A}0$, $\hat{A}1$, $\hat{A}2$, $\hat{A}3$, $\hat{A}5$ and $\hat{A}6$ hold, $\hat{L} =_{CFFD} L$.

\textit{Proof:} See [30].
Figure 2.32: Parentheses around an action mean that it will be hidden. Choosing $\tilde{A}(s_2) = \{c\}$ is legal by $A0$-$A5$, but the divergencehood of the trace $a$ is lost. This is why we need $A6$.

### 2.8.2 Symmetry reduction

Oftentimes we are interested in situations where a system has multiple identical subsystems. Prominent examples are the dining philosophers, general mutual exclusion and many-to-many communication protocols.

These situations have a tendency to create several essentially identical regions in the full LTS. This causes redundant work. It is possible to take advantage of the redundancy to avoid this work. We will need an additional concept to do that.

**Definition 23 (Symmetry bijection)**

Let $L = (S, \Sigma, \Delta, s)$. A bijection $f : (S \to S) \cup (\Sigma \to \Sigma) \cup (\tau \to \tau)$ is a symmetry bijection for $L$ iff $\forall s_1, s_2 \in S, a \in \Sigma \cup \{\tau\} : s_1 \xrightarrow{a} s_2 \iff f(s_1) \xrightarrow{f(a)} f(s_2)$.

A symmetry bijection is a specific kind of automorphism of the LTS. It maps states to states, label to labels and invisible transitions to invisible transitions. By the properties of bijections and the symmetry in the definition, symmetry bijections form a group under function composition, with the identity function as the identity.

You can use these groups to break the states into equivalence classes and then only investigate a single representative of each. This can be combined with partial order methods for great effect. The main difficulty here lies in finding a canonical representative of the class to recognize when we are in an equivalence class that has already been explored. For a treatment of this approach, see [7].

### 2.8.3 Cut states

Using compositionality to avoid the state explosion problem relies on the assumption that the intermediate compositions are smaller than the final result. It would be natural to expect that composing fewer LTS:s together would give a smaller result.
The fact is, this is sometimes not the case. Subsystems may have a very large state
space on their own, while reaching only a small part of it as a part of the system.
As an extreme example, imagine an unbounded counter. It could easily be used in
a system to count only up to three.

This problem can be dealt with using the interface processes of \cite{9}. We shall
present this idea using the more widely applicable formalism in \cite{28}.

The idea is to formulate the restrictions the correct full system will place on the
subsystem in a manner that allows us not to expand those parts of the intermediate
state space either. This is done using interface processes.

**Definition 24 (Interface process)**

An interface process $I$ is a tuple $(S, \Sigma, \Delta, \check{s}, \text{Cut})$, where

- $(S, \Sigma, \Delta, \check{s})$ is an LTS
- $\text{Cut} \subseteq S$
- No state in $\text{Cut}$ has any outgoing edges
- Every state $s \in S \setminus \text{Cut}$ has a transition by every label $a \in \Sigma$
- There are no invisible transitions in $\Delta$

The states in $\text{Cut}$ are called cut states.

Being a cut state is a dominant property: if any component in a composition is
in a cut state, the composite state is a cut state and none of its outgoing transitions
are in the result.

When the user guesses that the full system will not let a subsystem perform
a particular sequence $\sigma_{\text{never}}$, they can compose the subsystem with an interface
process that reaches a cut state whenever the sequence appears, thereby making the
composer ignore that region.

Since no non-cut state ever rejects any label and the interface process contains no
divergences, the parallel composition does not affect the system’s behavior outside
cut states.

If the user was correct and the system is correct, the sequence never occurs in the
full system and no cut state is ever reached. If there was an error, a cut state may
remain in the final result, which is a handy indication that the guess was wrong.

### 2.9 Tester Processes

Now that we have a toolset for the manipulation and reduction of state machines,
we shall present one tool that is designed for actually answering questions about the
properties of the LTS: tester processes.
Tester processes are used by composing them with the system. They contain special states that place conditions on what the system is allowed to do while there. For example, the system is not allowed to deadlock in a *deadlock rejection state*.

Tester processes can be used to describe a multitude of conditions you expect the system to fulfill. Figure 2.33 shows an example where the tester process forbids deadlocks after a path of the form \((ab)^+\).

A tester process is used to test specific conditions, it is not meant to produce a result that can be used for further composition. Therefore it can safely alter the behavior of the system, for example by ignoring parts of the state space that cannot violate the conditions it is checking. It just must not introduce or remove anything we are interested in.

You may observe that the process \(T\) in Figure 2.33 forces the parallel composition to only contain the traces it is checking and their prefixes. This is potentially very beneficial for performance.

To use tester processes you must augment the parallel composer to notice when a condition is not fulfilled and stop the construction of the state space with an appropriate error message.

Tester processes are very powerful. Using *rejection states*, deadlock rejection states, and *livelock rejection states*, it is possible to check for CFFD preorder \([13]\).

One very useful feature of tester processes is that when the composition results in an error, it will show you a specific state that breaks the requirements. It can be used to determine a trace that leads there, which gives us a counter example that can be used when debugging the system.

Note the similarities between tester processes and the interface processes men-
tioned earlier. The main difference is that interface processes just cut a part of the state space, but let the composition keep running elsewhere, while tester processes abort the composition. It seems best to implement both of these under some generalization.
3. COMPUTATION

3.1 A Concrete Point of View

We had a fairly illustrative look into the corner of verification that we strive to support in Chapter 2. Since our goal is to create a good architecture for using these kinds of ideas together, we must make their implementations communicate with each other. To understand the challenges that arise in this communication, we shall look at the ways these pieces consume and produce information.

3.2 Transformations

First we list techniques that take a single state machine and produce another. We will present each in the following format:

**Purpose** Explains why one may wish to use this transformation.

**Preserves** Tells what kind of equivalence there is between the input and the output.

**Input** Explains what information the transformation reads from the input and the logic by which it wants to read it, for example whether it accesses input as a list of edges, by graph search or something else.

The list will be followed by some free form text describing the items in more detail and providing other relevant information.

3.2.1 Bisimulation minimization

**Purpose** Reduces size. Recognizes equivalence.

**Preserves** Bisimilarity (aka, all useful information)

**Input** A fast algorithm, which requires a list of transitions sorted by labels, the number of states and the initial state (and optionally an initial partition), is known.

Bisimulation minimization finds the smallest bisimilar LTS. This can be done cheaply, both in time and in memory. To be precise, this can have $O(m \log(n))$ time complexity and $O(m + n)$ memory usage, where $m$ is the number of transitions.
and $n$ is the number of states $[27]$. Therefore this is unlikely to make any chain of manipulations considerably slower, while having a chance of making the LTS:s at least a bit smaller.

The natural output of this algorithm is a grouping of the states into blocks of bisimilar states.

### 3.2.2 Unfolding

**Purpose** Converts an RLTS to a pure LTS

**Preserves** Everything we care about

**Input** Graph search an RLTS

We will often need to convert RLTS:s to LTS:s, especially before variable aware versions of other techniques are available. This can be done simply enough by exploring the RLTS and creating states for all possible variable value combinations. This will inevitably take at least one time unit per created state and transition. See Subsection 2.6.3.

This can be implemented lazily. That is, this can provide a graph search interface to its result without having the entire result in memory.

### 3.2.3 CFFD Reduction by Acceptance Graph minimization

**Purpose** Finds a deterministic, hopefully smaller CFFD equivalent LTS.

**Preserves** CFFD-equivalence

**Input** Graph search an LTS

This algorithm, first presented in $[35]$, changes the LTS into a deterministic finite state automaton to describe its traces and stores divergences and stable failures explicitly in the states. For this, it uses a data structure it calls an Acceptance Graph (AG).

Then the algorithm uses basic bisimulation minimization or DFA minimization to get a smaller AG and turns it back into an LTS. The result is always smaller or equal to the deterministic version of the original LTS, but due to the potential exponential growth during the determinization, it could be even exponentially larger than the input.

In practice the result is usually smaller and this is in fact the way ARA $[32]$ and TVT $[38]$ do their CFFD reduction. CFFD is a weak equivalence, so the reduction in size can be dramatic.
The first step of this algorithm is the determinization of the LTS into an Acceptance Graph. To do that it needs to access the input with graph search logic; moving from node to node along edges.

The natural output of this block is the minimized Acceptance Graph. Its expansion into an LTS will increase its size proportional to the number of acceptance sets in each state. That stage can be done lazily.

3.3 Parallel Composition

One of the core components of our system is the implementation of parallel composition. It must produce a new LTS from a sequence of LTS:s and a description of their composition.

We choose synchronization rules as the description of the composition. They are easier to implement and more compact than the operators separately.

3.4 Composition Augmentations

A number of the techniques discussed in Chapter 2 are not used stand alone, but are instead used to augment the parallel composition, usually to avoid having to build the entire state space. We will list a few of these in the following format:

**Purpose** The purpose of the augmentation

**Preserves** The relationship between the result of naive composition and the result when the augmentation is used.

**Extra input** What additional input, if any, this augmentation requires.

The list is again followed by free form text with more details.

3.4.1 Cut States

**Purpose** Reduces size of intermediate composition results.

**Preserves** All behavior inside the bounds of the interface.

**Extra input** One or more interface processes with cut states.

The interface processes described in Subsection 2.8.3 require the parallel composition to be aware of cut states. Whenever it reaches a state it must check if it is a cut state in any of the input components to be able to mark it as such in the result and avoid expanding transitions.

The need to store cut states in the intermediate composition result increases its storage requirements slightly. All in all this is a very cheap technique with great potential advantages, but it requires the user to give good interface processes.
3.4.2 Tester Processes

**Purpose** Uses parallel composition to check some properties of the system.

**Preserves** Only the fact that the tester accepts or does not.

**Extra input** A tester process.

The tester processes introduced in Section 2.9 are not used to get smaller or better composition results. Instead they use the composition machinery to check if the system fulfills some requirements encoded into a tester process.

The costs of this augmentation in memory and time depend on tests used. For example: deadlock rejection can be done locally at each state, but divergence rejection requires graph traversal.

The more complex checks require much from the parallel composition. They may need to explore the state space in specific ways including looking at parts that have not been reached yet.

3.4.3 Stubborn Sets

**Purpose** Try to reduce the size of the composition result.

**Preserves** Can be tuned to preserve various equivalences

**Extra input** Depends on the equivalences preserved.

The stubborn set techniques described in Subsection 2.8.1 require fairly advanced algorithmics. This is especially true if full CFFD equivalence needs to be preserved, but already the local checks that are sufficient for conditions $\tilde{A}_1$ and $\tilde{A}_2$ require information about the components of the composition.

Like the advanced tester process tests, the tests for the conditions $\tilde{A}_4$ and $\tilde{A}_6$ need to manipulate the search order of composition and to search the graph outside the reached area.

3.4.4 RLTS

**Purpose** Expands the parallel composition to support Relational Labeled Transition systems.

**Preserves** RLTS information.

**Extra input** The inputs need to be RLTS:s.
Using the RLTS:s from Subsection 2.6.2 requires support from the composer. This support can be seen as just another augmentation. It too involves maintaining extra information in the LTS and using that information to affect the composition.

The states need to remember what variables they contain and the combined state must have all the variables of the component states. The expressions on the transitions must be similarly combined. The conditions on the same transition of separate components can also be mutually exclusive, which means that that transition can be dropped from the result.
4. ARCHITECTURE

In this chapter we will take a look at how to create a software architecture that allows many techniques like the ones presented in Chapter 3 to be used together. We will create concepts and general thought outlines that can be used to approach the implementation in a coherent manner. Some of these concepts we will present as types at the detail level of UML class diagrams.

4.1 Wirings and Blocks

Since state explosion is the great problem of the field, many of the transformations and augmentations we want to support are designed for reducing the state space. Perhaps the most basic pattern aiming to do this is presented in Figure 4.1. The inputs of parallel composition are reduced before the composition, which hopefully makes them smaller.

This likely makes the LTS between compose and the final reduction smaller, while the result reaching profit will be CFFD equivalent to the one without the reductions, thanks to congruence. This pattern can be used with any congruent reduction technique.

![Figure 4.1: The inputs of composition are reduced before composition to mitigate state explosion.](image)

An example of the usefulness of more advanced combinations is the behavioral fixed point method presented in [33]. It can be used to prove properties of unbounded, but finite systems with finite computation. The idea is that if adding one more component to a composition does not change its CFFD equivalence, induction tells us that we can add any number of them and not change the equivalence. See
A detailed understanding of this method is not relevant for the architecture. What is relevant is that the method has been found useful and that it involves repeating operations without knowing in advance how many repetitions there are, which means support for unbounded looping is desirable in our architecture.

Figure 4.2: An example of the behavioral fixed point method from [35]. First compose two philosophers. Then add new philosophers to the composition until successive compositions are equivalent.

Figures 4.1 and 4.2 presented the information flow in verification tasks using boxes and arrows. It is time to introduce some terminology to talk about these diagrams more precisely. This pattern where information flows from a software component to another in a sequence emerges frequently in software engineering. It is often called a pipeline and the boxes filters [40]. This terminology contains unnecessarily strong connotations of streaming behavior and does not allow loops, however, so we shall call the total a wiring and the boxes blocks.

When the wiring contains an arrow from box A to box B, box B is subsequent to box A. All boxes that can be reached from box A following arrows in the wiring are downstream from it.

With this new terminology we can state our goals as follows: we want the user to be able to try ideas on the level of wirings without having to worry about the internal details of the boxes. On the other hand, we want to make it possible to create various boxes without worrying too much about the wirings they end up being used in.

Thusly we divide the world into box concerns and wiring concerns. Box concerns are techniques that do some specific manipulations to LTS:s. Boxes must be independent of each other. Wiring concerns are ways to solve problems with combinations of these manipulations. They describe the dependencies.

This chapter focuses on creating modularity and flexibility by figuring out how to insulate wiring concerns from box concerns. The following chapter will give ideas on how to implement the architecture efficiently.
4.2 State machine interface

We have seen some techniques that will be implemented as blocks in our design. Our overview was by no means exhaustive, but the techniques we have presented have been diverse enough that a design generic enough to handle them is likely to be able to handle many more.

We will begin by examining the most fundamental part of our system: moving information about LTS:s from block to block. After that we will figure out how to implement parallel composition in a way that allows the user to create and use methods based on tweaking the parallel composition.

In previous closely related efforts the separate pieces of the verification toolset communicated using intermediate text files \[32, 38\]. Since the computational complexity of verification manifests as state explosion, there will potentially be a huge number of states and transitions which makes these intermediate files a serious bottleneck. Therefore we will use an in-process interface instead. The data need not be written to disk, which will vastly improve performance. Our level of abstraction will still leave the user free to store things on disk if necessary.

We can see already from the small sample in Chapter 3 that LTS manipulation algorithms use various specialized data structures to do their work. The bisimulation minimization algorithm we have referred to uses state and transition partitions \[27\], CFFD reduction uses acceptance graphs \[35\] and so on. We cannot just convert the data into a standard form between blocks either, since the state explosion could cause even a constant time conversion at block borders to devastate performance.

The obvious solution to this problem is to make the blocks communicate through a standard interface instead of a specific data structure. Unfortunately no interface can be implemented efficiently by every data structure. For example, it takes a long time to find all edges leaving a particular state from a list of edges ordered by their labels, while storing the edges by their tail states makes finding edges by a label slow.

These issues make a case against a standard interface, but in the end, each block needs to construct its internal data structures somehow and if they cannot be built directly from the output of the previous block, a conversion cost is inevitable. Therefore, the conversion costs associated with a standard interface cannot always be avoided even if we do not use a common interface, which means this is not a compelling reason to avoid the interface approach after all.

To avoid the cost when it can be avoided we will allow the blocks to communicate via short cuts when it is possible between them. The blocks will need to determine if they can use the result structure of the input directly or not. The default interface shall be a minimalistic graph search:
4. Architecture

<table>
<thead>
<tr>
<th>Graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial() : State</td>
</tr>
<tr>
<td>Transitions(State) : [Transition]</td>
</tr>
<tr>
<td>Head(Transition) : State</td>
</tr>
</tbody>
</table>

Figure 4.3: The state machine interface. The notation [Transition] means that the method returns something that can be used to iterate over a list of Transitions.

- Get the initial state.
- Iterate the transitions of a state.
- Get the head from a transition.

This interface should be fairly easily supported by most data structures that describe graphs. Of course some data structures, like a list of edges, make implementing the interface very inefficient. If the internal representation of some block is that far from this, it is best to just transform it into something else once the processing is complete. We will have explicit support for such a final transformation.

The fact that the transitions of a state are iterable means the consumer of this interface will receive them in order. We do not specify anything about what order that will be. The implementer need not even iterate them in the same order every time. An implementation can make promises about the order if it is deemed useful.

One advantage of this interface is that the graph need not exist in memory at all. The state may actually contain information to generate the adjacent states. This leaves us free to lazily stream the graph if we desire, which will be useful. The interface is presented using pseudo-UML [23] in Figure 4.3.

4.3 Annotations

4.3.1 Annotated Graphs

While much of the theory we summarized discusses pure LTS:s, it is an explicit goal of ours to support also variables as seen in Subsection 2.6.2. We saw also many other techniques that add extra information to the graph, for example cut states in Subsection 2.8.3. It would seem best to have some uniform, generic interface for any such extra data. We shall now lay out such a scheme, giving the extra data the name annotations.
Blocks will consume and produce various annotations. Blocks must be able to add them, remove them and change them. They should not cost much when they are not used. If we stored all annotations in the states and transitions they annotate, we would have to change each one when the set of annotations changes. Therefore it is best to store them in separate objects, at least conceptually.

We shall call these objects **Annoters**. An annoter will correspond to one annotation, for example the property of being a cut state or not. It shall have a method `Get`, which takes a graph element (a transition or a state) and returns the value of the annotation for that element. If ids are consecutive, an annoter can be something as simple as an array that contains the annotation value for each element, but it can also be an arbitrarily complex computation or just a constant. An annoter must return something for any graph element of the correct type from the graph it is annotating. However, an annotation is free to specify a value that means “undefined for this element” or something similar that fits its semantics.

An annotated graph will provide a method `GetAnnoter`, which shall take an identifier of an annotation and returns an **Annoter** that corresponds to that annotation. This design is depicted in Figure 4.4.

In addition to the states and transitions, the entire graph may need to have some extra information. For example, the graph may provide a mapping from internal transition labels to human readable names. Also this information may change in operations.

Since each annotation type is associated with either states, transitions or the whole graph, we shall adopt a naming scheme where the first word of the name states the association. For example the annotation **StateVariable** contains the variables in states and the annotation **TransitionExpression** the expressions that manipulate variables in transitions.
4.3.2 Basic Annotations

In this subsection we shall list some basic annotations to help the reader understand some of their possibilities.

**StateVariables, TransitionExpression** These allow you to describe RLTS:s. The StateVariables annotation gives the set of variables for each state and TransitionExpression gives the relation of variables and parameters using some formalism.

**GraphNodeCount, GraphTransitionCount** Whenever you already know the full number of nodes or transitions in the graph, you can store it in an annotation. This annotation is meant for situations where the size is known (or it is very cheap to calculate). Graphs that cannot answer these questions quickly must not claim to have these annotations.

**StateArrayIdentity** This gives an unique index in \([0 : : \text{GraphNodeCount})\) for every node in the graph. This can be used to store information about each node in an array of size \(\text{GraphNodeCount}\) with the identity as index.

**GraphHashSize, StateHash** When the number of nodes is unknown and you wish to store data about each one, using a hash table is often a good solution. The creator of an implementation of Graph may be able to create an excellent hash function for states based on their internal structure. These annotations will expose such a hash function to the users. The hash of each state must be smaller than GraphHashSize. The annotations GraphNodeCount and StateArrayIdentity are in a sense an ideal implementation of these two.

4.4 Parallel composition framework

The interface described in the two previous sections is sufficient for the modular implementation of techniques that take in a state machine and output another one like those described in Section 3.2. In addition to those there exist many techniques like those in Section 3.4: ones that take in a parallel composition and return something that is somehow equivalent to the result of that parallel composition. This subsection discusses our approach to modularizing techniques like these.

The framework must of course provide a naive parallel composer. Some earlier verification tool sets have implemented all the parallel composition augmentations they wanted as features of the single parallel composer [32, 38]. This makes the composer bulky and makes it difficult to develop the augmentations independently. We will try to enable much greater modularity in augmenting the composition.
Modularity requires modules and modules require an interface. Therefore, we must identify common denominators in the augmentations:

- Reject states tell you to abandon the composition if they are reached.
- Cut states tell you to ignore all transitions from them and possibly abort the composition if the user claims none should exist at this point.
- Deadlock and livelock rejection states abort the composition if specific conditions are fulfilled.
- Also stubborn set methods ultimately come down to ignoring specific transitions when specific conditions are fulfilled.

From the point of view of the composition, the effect of all these plugins is very similar. They use conditions to either tell you to abort the composition or to ignore specific transitions.

Unfortunately, the input the techniques require to evaluate their conditions is highly variable. For example, cut states only need the cut states to be marked with an annotation, while the stubborn set method needs to know about the presence of cycles, which is non-local information.

Considering the variability apparent in the small sample of techniques we have looked at, it is likely that some other similar approach would require something entirely different from the reading of the input. Therefore, rather than try to guess all the ways the exploration may need to be guided, we take an opposite approach.

We let the augmentations take full control of the search. We do this by providing a lazy naive parallel composer and putting the composition augmentations on top of it.

What we mean by a lazy parallel composer is that it does not compute the entire state space when called and store the result. Instead it saves the rules and the components and computes the transitions from a state only when asked. You may compare this to the SuperCombinators of FDR.

When the composer is lazy, it performs no search itself. This allows the subsequent block not only to determine the search order but also to not search parts of the graph at all, which is what all the above augmentations wanted to do. This converts composition augmentation from a block concern into a wiring concern. You can see an example in Figure 4.5.

In the figure we represent the fact that Compose is lazy by drawing it with dashed lines. We have also marked Cut States as lazy because it can be implemented lazily by returning a Graph that returns an empty list as the transitions of any cut state and forwards all other queries to the input graph.
4. Architecture

We have also drawn some of the arrows with dashed lines, which represents the fact that the implementations of Graph they represent are lazy. This additional notation is useful, because also eager blocks may produce lazy output. For example an implementation of CFFD reduction using the method mentioned in Subsection 3.2.3 will explore its input fully, but memory can be saved by transforming the resulting acceptance graph into an LTS lazily.

![Diagram](image)

Figure 4.5: By making parallel composition lazy, we can formulate composition augmentations as wirings. The dashed lines indicate lazy graphs.

Since augmentations are essentially filtering operations, they can often be implemented lazily. When augmentations are lazy, they can be added after the composition and the state space won’t be generated until the first eager block downstream. This means the states that get pruned by the augmentations never exist. You can still use augmentations after a block that fully explored the state space, but much of the time and memory gains will be lost.

This means that augmentations that cannot be implemented lazily are less flexible. If an augmentation explores the state space to do its reduction, the augmentations downstream cannot prune the states before they exist. Pruning them after they have been created still makes the final result smaller, but the time and memory to create them will already have been spent. Therefore only one eager augmentation can be used to its full potential after a single composition.

One example of an augmentation that cannot be implemented lazily, at least not naively, is the stubborn set method when conditions Ā4 and Ā6 are checked using the sufficient conditions of Theorem 2. They require non-local information which is only available when the full search is being done. It would be possible to let the stubborn block be lazy by back propagating the non-local information to the stubborn block from wherever it is available. Such a thing is not impossible in our architecture, but we do not explore the idea further.

When the augmentations are just blocks, it is possible to insert other transformations between augmentations. In a sense, the concept of augmentations has disappeared: they are just specific transformations.
They do however require additional information. For example, the stubborn set method needs to see the original labels of freshly hidden actions and the behaviors of the components (if using the sufficient conditions for \( \tilde{A}_1 \) and \( \tilde{A}_2 \) from Subsection 2.8.1). Fortunately we already have a system for passing around extra information: annotations.

The lazy composer will need to provide the following annotations to support all the augmentations we have seen:

**TransitionLabel** The label of the result transition given by the synchronization rules

**StateComponentStates** The states of composition components

**TransitionRule** The synchronization rule that generated this transition

**GraphRules** The full set of synchronization rules

**GraphComponents** The tuple of component graphs

**CompoundAnnotations** Compound annotations give access to the annotations of the component graphs. For each annotation \( A \) in any component graph, there will be a compound annotation which provides access to the value of \( A \) in each component which it is defined in. The resulting **CompoundAnnotation** appears on the same elements as \( A \). This will be elaborated on in the next section.

All this laziness requires care. Some block may wish to search the same part of a graph multiple times. It might be inefficient to reperform all the lazy computations every time, so the framework should provide block that memoizes the graph, a **GraphMemoizer**. It takes a lazy graph and is lazy itself, but constructs the so far explored graph internally and answers queries using it and therefore does not ask the original graph the same question twice. This avoids incurring the cost of performing the lazy computations several times. The **GraphMemoizer** gives considerable speed up if queries are repeated and incurs a significant memory cost if you are not otherwise storing the graph.

It may be useful to provide also a block that eagerly explores the graph and stores it, an **EagerGraphCopier**. It makes sense to use the **EagerGraphCopier** when you know the next step is going to fully explore the graph anyway, since a real copy will be able to answer questions faster and allows you to immediately destroy the previous intermediate results.

\(^1\)There are other useful ways to check them, but they all require some sort of extra information.
4. Architecture

4.5 Annotation Propagation

4.5.1 Blocks Manipulating Annotations

When the graph contains annotations, we need to define how they are treated when the graph is manipulated. For example, think about the parallel composer. Its states are combinations of states of component graphs. Many annotations need to be preserved in some form through that combination.

- For cut states, if a component state is a cut state, the compound state must be a cut state.
- For RLTS:s, the set of variables of a compound state is the disjoint union of variables in the component states.
- For tester processes, if a component state is some kind of a rejection state, the compound must be the same kind of a rejection state.

Thus it would seem like a good idea to allow data to describe how it is manipulated by various operations. Unfortunately, there are many operations, sometimes with subtle differences. Compare combining states in a minimization to combining states in composition. That approach would make it necessary to specify behavior for every element in the Cartesian product of annotation types and operations.

We will resolve these issues by approaching the problem from another perspective. Instead of annotations defining what annotations exist in the result, we will simply collect all component values of an annotation in a compound annotation and let the user use a later block to compute the desired result from them.

For example, composing annotated graphs does not automatically give you an annotated graph with specific annotations. Instead, the graph that the lazy composer returns has compound annotations that give access to the annotations on all the components. You can then use transformations to compute the result annotations any way you like. This, too, becomes a wiring concern.

In fact, many of the annotations we have seen are already associated with some augmentation that can take care of annotating the result correctly. For example, the cut state augmentation will remove the compound cut state annotation and replace it with a regular cut state annotation, which is true if any of the components of the compound annotations was.

For annotations that are not associated with any more complex manipulation it may be good to provide some helper block, which can be given a function for extracting a result annotation from an existing compound annotation. Similarly, discarding an annotation should probably be done using a block.
Also other operations that combine states or transitions, such as minimization, should collect their annotations in compound annotations. It is probably best to provide a way to differentiate between compound annotations generated by different operations.

4.5.2 Annotations Manipulating Blocks

The above approach makes some downstream block worry about how an operation affects annotations. The effect of annotations on an operation cannot be dealt with that way. For example, if the expressions on two transitions in an RLTS are mutually exclusive, their combination should not exist in the result of composition. For another example, cut states need to stop biminimization from combining two states only one of which is a cut state.

Based on these needs, we will give annotations the capability to affect the equatability of elements. For more complex interactions between annotations and operations it is best for the operation to be explicitly aware of the annotation.

4.6 Wiring blocks together

In this section we will think about our framework from the point of view of the final user, who has a collection of transformations and compositions they wish to put together in a specific way. How should they do this? How will the user create the wiring?

The framework must support arbitrarily complex wirings. We want them to be able to contain loops, control structures and preferably also things we haven’t thought of yet. Creating any kind of GUI for this task would be a major undertaking and would make it hard to do things we have not anticipated. That leaves two feasible options: a configuration file written using some specifically designed language or a program written using our framework as a library.

The custom language would allow us to make the core concepts succinct to express, allow good error messages and make usage in general easier. It would also leave open the possibility of automatically optimizing the wiring order if sufficiently was known about the compositionality of the pieces, in a manner reminiscent of what SQL processors do. For example, the technique in Figure 4.1 could be used automatically whenever appropriate. This is the approach taken by a verification tool called FDR \cite{FDR}. Such a system would however require large amounts of work and inhibit the compiler from providing optimizations to the code itself.

Therefore we choose to make our framework a library and the wirings programs. This can be extremely flexible without any extra effort on our part since the user has a full programming language at their disposal. The lack of an intermediate language
decreases overhead and potentially allows you to benefit from the formidable optimization capabilities of modern compilers, especially if implemented using metaprogramming. Since the users are expected to be familiar with programming, this also avoids the need for them to learn a new syntax.

In some implementation languages there is of course the option of getting some of the benefits of a custom language back by using embedded Domain Specific Language (DSL) capabilities available.

It is also worth noting that a framework in the form of a programming library can be easily used later to create a GUI or a custom language if the need and resources arise.

Listing 4.1 shows pseudo code that gives an example of what a wiring might look like as a program. We must now design our framework as a library for these programs.

As discussed before, some blocks may result in data structures that cannot implement the Graph interface as efficiently as required. Such blocks must have a final stage that converts the result data structure into something that can.

The transformation may be costly, which is why we promised to enable shortcuts in case the next block can use the already existent data structure efficiently. We do this by requiring that any block returns not a Graph, but a Graphable. Blocks are free to produce data structures not suitable for the Graph interface as long as they are Graphable.

Figure 4.6 depicts the idea. A Graph can implement the Graphable interface by returning itself. A data structure that cannot implement Graph efficiently, for example a list of transitions, needs to actually generate something new in ToGraph. If the next block in the wiring can use a list of transitions as the input, we can avoid calling ToGraph all together.

We could either ask all blocks to accept Graphables as input and let them check if they are able to use a shortcut, or we could let them have arbitrary parameters and force the user to know when they can take a short cut and when not.

Whenever dealing with performance, explicitness is very useful. Forcing the user to think about what they are doing makes it more likely that they do. On the other hand, the user may just always default to graphing even when savings could be achieved.

The implicit short cutting would also make life easier. Additionally, since the users and the makers of the blocks have the full capabilities of the programming language at their disposal, they are always free to agree to sidestep the general interface if they desire.

Therefore it is best to make the general interface be as easy to use as possible and make it able to increase efficiency without the user’s knowledge. Therefore blocks
4. Architecture

Figure 4.6: The **Graphable** interface and examples of classes that may implement it.

```java
int main([string] args){
  Graph g1 = read_graph_file("file1.g");
  Graph g2 = read_graph_file("file2.g");
  Rules rules = read_synchronization_file("rules.rl");
  // BasicComposer implements IntermediateComposer<Graph,Graph>
  BasicComposer composer = new BasicComposer(rules);
  IntermediateGraph g12_raw = composer.compose(g1,g2);
  Graph g12 = RLTS_augmentation(g12_raw).ToGraph();
  print_graph(g12);
}
```

Listing 4.1: Pseudo code that shows how a wiring is created by writing a program.

shall take as input a **Graphable**, not a **Graph**. Then they can check if the provided interface is useful for them and if not, turn it into a **Graph**. See **Listing 4.2** for an example.

4.7 Parallelization

With so much of modern computing power coming in the form of parallel processing, an application requiring such extensive computation cannot afford to ignore it. Since our design is above the actual implementation, all we can do is make sure our design does not inhibit any parallelism the implementer may wish to exploit.

Firstly, we must avoid hindering the individual blocks from doing things concur-
rently. Since the only contract we have with them are the **Annotated** and **Graph**
Graphable biminimize(Graphable ga) {
    TransitionList tl;
    if (dynamic_cast<TransitionList>(ga) != 0) {
        // We can use the data directly
        tl = dynamic_cast<TransitionList>(ga);
    } else {
        // This will take time
        tl = construct_list(ga.ToGraph());
    }
    // Do the biminimization using tl
    ...
}

Graph g = read_graph_file("graph.txt");

// Some imaginary transformation whose natural output is
// an ordered list of transitions.
TransitionList tl = homonymize_labels(g);

// Some imaginary transformation that naturally results in
// a bitmap that somehow encodes the graph
GraphableBitmap bm = trans_quadratic_hilamarv_decomposition(g);

// Biminimize will notice that it can use tl
// directly and no intermediate Graph is constructed
Graphable result2 = biminimize(tl);

// Biminimize has to call ToGraph(),
// since it cannot read the bitmap interface efficiently
Graphable result1 = biminimize(bm);

Listing 4.2: An example of avoiding the cost of the generic interface.
interfaces, all we need to do is guarantee that they are fully re-entrant and thread safe. Therefore we require that from any implementation. This should not be unreasonable since they are read-only interfaces. Since short cuts are used implicitly, also the interfaces that short cuts provide must be thread safe.

There are two more chances for parallelism. They appear in the relationships of the blocks. We call them independent parallelism and streaming parallelism.

Independent parallelism refers to a situation where you have a branch in the wiring of your blocks and can therefore run two blocks independently of each other. Say you have two input graphs and try to minimize both before composition. You can do the minimizations simultaneously. Thread safety of all communication channels between blocks makes this safe.

Streaming parallelism is somewhat less likely to be useful, but we mention it for the sake of completeness. When functionality is hidden behind a lazy interface, nothing forces the implementation to perform its computation only on demand. It can actually keep producing results in a separate thread so that they are instantly available when they are required. The Graph interface is abstract enough to support this without any modifications, but the actual implementation of it in this manner would require considerable care. The two forms of block parallelism are illustrated in Figure 4.7.
In this chapter we present ideas that should help in implementing the above architecture. We will pay special attention to improving the performance of the end product. The above design can be implemented without using any of the ideas in this chapter. The ideas we shall present are based on template metaprogramming, which may discourage or even prohibit their use, depending on the implementation language and available expertise.

5.1 The Cost of Abstraction

In the above section we presented an architecture with great flexibility and abstraction. It breaks tasks into a large number of more or less independent pieces. This is desirable from a code writability and manageability perspective, but it comes at a cost. Even fairly simple things end up being done through layers upon layers of interfaces. This has severe implications for performance.

Consider a naive C++ implementation of the design. Since the blocks do not know what kind of a Graph they are accessing the data through, the usual thing to use would be dynamic polymorphism. Graph would be an abstract base class and its methods would be called using virtual function calls. Since the computational challenges of the field manifest in the number of states and transitions, these methods are a significant bottleneck. The additional cost of a single virtual call is not great, it is in the order of magnitude of an array lookup. But if the function you call only does an array lookup itself, the virtual call ends up doubling its cost. Multiply that with a hundred million transitions and you will have a significant factor. Additionally, virtual function calls cannot be inlined, which inhibits many compiler optimizations.

The problem is even more pressing in the parallel composition framework. In approaches where all the augmentations are integrated into the composer, all the computations of the augmentation can be put in the same compilation unit, which allows for optimizations and does not have great overhead. In our flexible framework, the result of composition is constructed through many layers of independent augmentations, many of which may be simple. This makes the virtual call overhead very significant.

We solve these issues using template metaprogramming. We replace dynamic
interface StateCutAnnoter{
    bool GetAnnotation(State s);
}

class DummyStateCutAnnoter: StateCutAnnoter{
    bool GetAnnotation(State s){
        return false;
    }
}

bool IsCutStateInComposition(State s_1, State s_2, StateCutAnnoter a_1, StateCutAnnoter a_2){
    return a_1.GetAnnotation(s_1) || a_2.GetAnnotation(s_2);
}

Listing 5.1: Example A. A usage of dynamic polymorphism. Any StateCutAnnoter has a method GetAnnotation: State->bool and IsCutStateInComposition finds it at run time.

polymorphism with static polymorphism. That means that the polymorphism is resolved at compile time. This requires that the compiler knows what kind of a Graph each graph is. This is not a significant limitation since we decided to describe the wiring as a program anyway.

5.2 Static Polymorphism

In this section we will explain what we mean by static polymorphism.

With dynamic polymorphism we may write example A in Listing 5.1. We can call IsCutStateInComposition with any StateCutAnnoter and be confident that it will give us booleans. Unfortunately, this comes with a small performance cost. The virtual function must bee looked up. In a thoroughly abstracted hierarchical system there can be deep chains of functions that do not really do anything but call the next function. The aggregate cost of the calls grows to be significant especially if there is a large number of calls through the interface. In the above example there would be two virtual calls per a state in the composition, which is potentially a huge number due to state explosion.

These issues can be largely avoided using static polymorphism. Behold the revised example A in Listing 5.2. In the version presented there, we can give IsCutStateInComposition anything that has a GetAnnotation with the correct signature and it will work. All the function calls are resolved at compile time. Functions that only call another function can be optimized away by the compiler. In the case of the dummy annoter, the compiler would likely be able to reduce IsCutStateInComposition to the constant false. This technique is used extensively in the c++ standard library, notably in the way the algorithms library accepts iterators. [18]

The advantages of this approach are illustrated by Figure 5.1. It presents the
class DummyStateCutAnnoter{
    bool GetAnnotation(State s){
        return false;
    }
}

bool IsCutStateInComposition<A>(State s_1, State s_2, A a_1, A a_2){
    return a_1.GetAnnotation(s_1) || a_2.GetAnnotation(s_2);
}

Listing 5.2: Example A implemented with static polymorphism using templates. The function IsCutStateInComposition looks up DummyStateCutAnnoter.GetAnnotation at compile time and produces an error if it doesn’t find it.

time to call a very small function through a call chain of ever greater depth. For both cases the nesting is achieved by a recursive function that does nothing but call itself with a decremented nesting if it is above zero and call the final function if nesting is zero. For the static case this is done with template metaprogramming. It should be noted that since many of these decrements are done by the dynamic version, they are a significant part of its computation and our test should not be taken as a pure measure of function call cost.

You can see that the static version is not only faster, but actually remains constant as the number of nested calls grows, since the compiler can see that the intermediate calls do nothing. In fact, we were forced to add a modulo operation to the incrementation to make the static version depend even on the number of times it was called in the loop. Without that the compiler saw what the result of the loop was and optimized the loop away entirely. The fact that this issue arose only serves to prove our point: the compiler has vastly more information at its disposal when using the static method and it can use it for optimization.

In fact, since the call chains are unlikely to be quite this deep in a reasonable implementation, the additional information available to the compiler may be more significant in providing speedups than the avoidance of function calls.

This approach has its own problems. Firstly, if we try to give IsCutStateInComposition something that does not have GetAnnotation, we get an error about a missing function from the line where calling it is attempted. Worse, if it has GetAnnotation with a wrong signature, we may get a warning about incompatible operands for operator "\|\|", or something like that. These error messages do not tell you the real problem, which is that you passed in an incompatible type argument. Due to overload resolution, template matching rules and other features of the language these error messages also tend to become very long and complex.

Even worse, we do not want to talk about the implementations of functions in this thesis, and therefore any constraints on parameters that are determined by the
5. Implementation Ideas

Figure 5.1: The time taken to call a function that increments and takes the modulo of a single variable through a call chain of increasing depth. The function is called nine million times at each point. The static version stays at approximately 0.04s regardless of nesting.

details of how they are used in the implementation should be invisible to us.

Let us explore these problems in the context of more things we wish to do with templates. We shall use types as parameters to templates to move around information at compilation time without any actual instances. This technique is also used in the c++ standard library, for example in numeric limits. 

Look at example B in Listing 5.3. There we specify annotations using the types StateCutAnnotation and TransitionExpressionAnnotation. These structs exist only to provide the types used by Annoter. This technique suffers from the problem that the type system provides no means to express that the parameter A needs to contain the type definitions Type and Target.

Since our programming languages don’t provide a way to describe requirements like the ones in the above examples, we shall define a concept of our own to do it. We shall call it *type interfaces*. A type interface can have two kinds of members. First, there are members that allow you to require static members and typedefs from the type. These allow you to require the presence of members like Type and Target in Listing 5.3.

The second kind of type interface members are *instance members*. They are denoted by prepending a member with the word instance. These allow you to require instance members from the type. They are very much like members of regular interfaces except that the calls are not necessarily virtual. They allow you to express requirements like the ones IsCutStateInComposition has for its type parameter in Listing 5.2. Let us look at the previous examples using them.

---

1We do not disallow using virtual functions to implement type interfaces, we simply avoid requiring it.
5. Implementation Ideas

```c
struct StateCutAnnotation{
    typedef Type = int;
    typedef Target = State;
};

struct TransitionExpressionAnnotation{
    typedef Type = Expression;
    typedef Target = Transition;
};

interface Annoter<A>{
    A::Type GetAnnotation(A::Target);
};

Annoter<StateCutAnnotation> state_cut_annoter;
Annoter<TransitionExpressionAnnotation> transition_expression_annoter;
```

Listing 5.3: Example B. An example of template programming. We have parametrized the `interface Annoter` so that it works for any kind of annotation. Since the parametrization has been done using templates, it can be resolved at compile time.

```c

type interface StateCutAnnoter{
    instance bool GetAnnotation(State s);
};

class DummyStateCutAnnoter: static StateCutAnnoter{
    bool GetAnnotation(State s){
        return false;
    }
}

bool IsCutStateInComposition<A:StateCutAnnoter>(State s_1, State s_2,
    A a_1, A a_2){
    return a_1.GetAnnotation(s_1) || a_2.GetAnnotation(s_2);
}
```

Listing 5.4: Example A revised to use a type interface.
5. Implementation Ideas

```cpp
type interface Annotation{
    typedef Type;
    typedef Target: State | Transition | Graph;
}

struct StateCutAnnotation{
    typedef Type = int;
    typedef Target = State;
}

struct TransitionExpressionAnnotation{
    typedef Type = Expression;
    typedef Target = Transition;
}

interface Annoter<A : Annotation>{
    A::Type GetAnnotation(A::Target);
};

Annoter<StateCutAnnotation> state_cut_annoter;
Annoter<TransitionExpressionAnnotation> transition_expression_annoter;
```

Listing 5.5: Example B revised to use type interfaces.

In Listing 5.4 we have again the example with a state cut annoter. The function `IsCutStateInComposition` can now explicitly state that its parameter `A` must implement the type interface `StateCutAnnoter`, which in turn requires it to have a method `GetAnnotation`. We have used the additional notation `class C : static TypeInterface` to underline that `DummyStateCutAnnoter` implements `StateCutAnnoter` statically, but we will not require this; `DummyStateCutAnnoter` is a valid `StateCutAnnoter` as long as it has the required method. Therefore, the `Annoter` example can be written as in Listing 5.5.

There we have simply defined the type interface `Annotation` and require it from the template parameter `A` of `Annoter`. `StateCutAnnotation` and `StateCutAnnoter` need not be changed to explicitly declare that they implement it. This allows you to define type interfaces for classes you cannot change.

Note that when we require the type to contain a `typedef`, we can require that the type it refers to satisfies another static interface or other condition. Above, the alias `Target` must refer to one of the types `State`, `Transition` or `Graph`.

We have defined type interfaces mostly to improve our presentation. There is no need for them to exist outside the minds of users of this framework. Nonetheless, emulating them in code to some extent will likely be beneficial. It will give the user better error messages at the very least. They can be emulated using `static if` in D [8] and Substitution Failure Is Not An Error (SFNAE) trickery in C++ [41]. Both languages also have static assertions that will be helpful in creating useful error messages.
5. Implementation Ideas

The emulation boils down to creating a compile time function that checks if the parameter type has the members you want to require from it. Appendix A.1 contains a simple example of a type interface in D and Appendix A.2 in C++. The compile time reflection available in D might allow this to be done more fluently [2].

There have been attempts to create a very extensive system like this in the C++ language in the form of concepts [10]. Unfortunately, they have not made it to the standard, which is why we make do with our own lightweight definitions. The type classes of Haskell address similar issues in a very powerful manner [17].

5.3 Pseudo code notation

5.3.1 Motivation

In the following section we will be presenting ideas using static interfaces. For compact presentation and language independence, we will use pseudo code with some notational shortcuts, which shall be presented here.

Much of this notation is created to support implementing the design using extensive template metaprogramming. The extensive type information enabled by this presentation is also very useful in describing the interfaces without going into algorithmic details.

When we use the word informal in the list below, we mean that the concept cannot be used to describe things to the compiler; you will have to be satisfied with using it to describe things to people.

5.3.2 Notation

Function type We have tried to keep our pseudo code somewhat close to C++ for familiarity, but the syntax for the type of variables that contain functions (function pointers) is so cryptic there, that we use one derived from mathematics instead. The type of a function taking parameter types \(P_1, P_2, \ldots, P_n\) and returning \(R\) is written \((P_1, P_2, \ldots, P_n) \rightarrow R\).

Parametric interfaces and methods are denoted by angle brackets.

\begin{verbatim}
interface Foo<T>{
    T bar<V>(V grue);
}

Foo<int> f;
int i = f.bar<string>("rikk");
\end{verbatim}

Enumerables of a type \(T\) are denoted by \([T]\). Whatever idiomatic way to to iterate through a collection can be used as long as it is opaque enough to allow laziness. In c++ this entails two members, begin and end.
Template parameter types We will sometimes use the notation $T:I$ in template parameter lists to denote that the parameter type $T$ must conform to the interface $I$. In languages lacking this capability, the requirement is informal.

Type lists In template parameters lists, $[<T>]$ shall stand for a list of types. Depending on the implementation language, such a construct may have to be left to the runtime in practice.

Type list concatenation When $T$ is a type and $L$ is a list of types, $T[]L$ shall denote a list of types with $T$ prepended to $L$.

Type list Select If every item in a type list $L$ has a member $Name$, we can get a type list of those members by the expression $Select<Name, L>$.

Type inclusion When $A$ and $B$ are types, we shall use the notation $A$ is $B$ to mean that $A$ can be used as a $B$. This usually means that $A$ inherits from $B$, but in metaprogramming the requirement may have to be informal.

Static function The notation $T bar<K x>$ as a member of a type shall denote a static function. That is, given some $K a$, we can evaluate $bar<a>$ to get an instance of $T$ at compile time.

5.4 The Architecture

In this section we will illustrate how implementing our architecture can be approached using the idea of static polymorphism.

5.4.1 Graph

The basic $Graph$ interface is presented in pseudo code in Listing 5.6. Take note that the properties of states and transitions are accessed through the main graph object. This leaves open the possibility for them to be stored in centralized data structures, while leaving the types $Transition$ and $State$ as simple handles. Of course, this approach does not prohibit keeping actual data in the objects, either.

5.4.2 Annotations

To make the processing of annotations efficient, also all information on what annotations each graph has must be available at compilation time. That means we must store them in the type of the graph, more precisely the type of the $Annotated$ it implements.

See Listing 5.7 for the pseudo code. We use instances of $Annoter$ to access annotations. A specific annotation is identified by a type that implements the type
interface Graph
{
    // Get the initial state of the graph
    instance State initial();
    // Get the transitions starting from a state
    // Must run in $O(m)$ where $m$ is the number of transitions returned.
    instance [Transition] transitions(State);
    // Get the head state of a transition
    instance State head(Transition);
}

Listing 5.6: The fundamental graph interface

interface Annotation. We will never have instances of Annotation or its implementers. They exist only to contain compile time metadata about each annotation. There is a simple implementation of this static interface in Appendix A.1 and Appendix A.2 in D and C++, respectively.

This metadata includes the type of the values of this annotation and whether this annotation is used on states, transitions or graphs. The annotation also provides a way to check if the annotation allows combining two annotated elements. The annotation types may also contain other metadata.

5.4.3 Wiring

We decided to make blocks accept and produce Graphables. The Graphable interface is presented in pseudo code in Listing 5.8. The type of the result graph needs to be known statically. Hence we make it a type parameter.

It is unlikely that blocks will need to be manipulated transparently, but it may be useful in some cases. Therefore we define a simple static interface for them also. It is presented in Listing 5.9.

5.4.4 Composition

The composer needs to make all the annotations of the components available for later blocks to interpret. Since the annotations are in the types of the graphs, this leads to fairly complex type acrobatics in the composer.

The pseudo code is in Listing 5.10. The composer produces an IntermediateGraph. First, it has annotations that describe the composition information: TransitionLabel, StateComponentStates, TransitionRule, GraphRules, GraphComponents. Then, it has an indexed annotation for each annotation that appears in any annotation at least once.

An indexed annotation has a value Type of a tuple that contains the value of the annotation in each component that has it. An indexed annotation can additionally
Listing 5.7: The fundamental interface for getting annotations.

Listing 5.8: The Graphable interface

Listing 5.9: The Block interface. Since the role of blocks is to do anything we want done, they are expected to have a lot more going on than the minimum required by this interface.
5. Implementation Ideas

tell which components provided the values in the tuples.

An example of an intermediate composition can be seen in Figure 5.2. The subfigures a and b present the components. They are composed by the synchronization rules in subfigure c to get the intermediate graph in subfigure d. Subfigure e shows the type of this intermediate graph.

5.4.5 Augmentations

We made the composition augmentations responsible for interpreting the compound annotations associated with them and placing any resulting annotations in the result.

We use the cut state augmentation as an example of how this is visible in the types. Its type is presented in Listing 5.11.

5.4.6 A Colorful Example

To further illustrate these ideas, we will now explore a small example. Take a situation where three component graphs describe separate colors and we want the result to contain luminosity, aka the average of colors. Listing 5.12 presents the annotations needed for the example. Each of the original three graphs is annotated with StateColor and the augmentation shall combine them into a StateLuma annotation.

The augmentation itself is shown in Listing 5.13. It is a Block that takes in an intermediate graph that is annotated with the compounded color information.

It returns a custom graph that forwards all calls except GetAnnoter to the original graph. The customized GetAnnoter also forwards most queries to the original, but no longer allows getting the color compound annotation and instead provides the StateLuma annotation with the StateLumaAnnoter.

Some things done by hand in the example are so common that helpers for the would be useful. The whole pattern of replacing an annotation with another could be packaged into a reusable template.
/// The type interface for an intermediate composer.
/// Composition requires that each graph type
/// has the TransitionLabel annotation

type interface IntermediateComposer<GraphTypes:[<Graph>]>{

typedef IntermediateGraph : AnnotatedGraph<
    Annotations = TransitionLabel [],
    StateComponentStates [],
    TransitionRule [],
    GraphRules [],
    GraphComponents [] // Per graph
    For each unique annotation A
    in Select<Annotations, GraphTypes>
    include IndexedAnnotation<A>
>;

/// An annotation that holds the states the components are in
struct CompoundState: static Annotation{
    typedef Type = Tuple<Select<State, GraphTypes>>;
    Targets Target = State;
}

/// Knows which component each value for the annotation comes from
struct IndexedAnnotation<A>: CompoundAnnotation<A, Count>
    where Count = CountGraphsWithAnnotation<GraphTypes,A>{

    /// The index in the original list of components (GraphTypes),
    /// of the component that provides the index:th value
    /// in the tuple.
    uint SourceGraphIndexOfIndex<uint index>;
}

IntermediateGraph compose(GraphTypes...);
}

struct CompoundAnnotation<A: Annotation, Count: uint>: static Annotation{
    typedef Type = Tuple<A::Type, A::Type, ..., A::Type>; // Count times
    Targets Target = A::Target;
}

Listing 5.10: The intermediate composer
5. Implementation Ideas

(a) A state machine with variables and colors. Having variables but not expressions doesn’t make sense, but is allowed.

(b) A state machine with variables and expressions that manipulate and check them.

(c) The synchronization rules $R$, for composing $L_{\text{color}}$ and $L_{\text{vars}}$.

(d) The raw result with intermediate information available. In addition to what is pictured, the graph itself provides access to the component graphs and the full set of synchronization rules.

```plaintext
typedef L_color_type = AnnotatedGraph<StateColor, StateVariable>;
typedef L_vars_type = AnnotatedGraph<StateVariable, TransitionExpression>;
typedef Composer = IntermediateComposer<L_color_type, L_vars_type>;
Composer::IntermediateGraph == AnnotatedGraph<
    Composer::StateCompoundState,
    GraphRules<L_color_type, L_vars_type>,
    GraphComponents<L_color_type, L_vars_type>,
    TransitionRule<L_color_type, L_vars_type>,
    Composer::IndexedAnnotation<StateColor>, // Count = 1
    Composer::IndexedAnnotation<StateVariable>, // Count = 2
    Composer::IndexedAnnotation<TransitionExpression> // Count = 1
>
```

(e) The annotations generated to the type of the intermediate graph.

Figure 5.2: An example of parallel composition with annotations.
Listing 5.11: Pseudo code for the type of the CutStateAugmentation. To express the type Result more formally one needs to use techniques we won’t go into.

Listing 5.12: Colorful example annotations.
class ColorAugmentation<G: Graph, Annotated<Annotations>> :
Block<G>
where CompoundAnnotation<StateColor,3> in Annotations{

typedef NewAnnotations = StateLuma []
    Remove<CompoundAnnotation<StateColor,3>, Annotations>;

class AnnotationReplacerGraph:
static Graph, Annotated<NewAnnotations> {

private:
    G original;

public:

AnnotationReplacer(G original):original(original){}

State Initial(){
    return original.Initial();
}

[Transition] Transitions(State state){
    return original.Transitions(state);
}

State Head(Transition transition){
    return original.Head(transition);
}

Annoter<A> GetAnnoter<A>(){
    static if(A == CompoundAnnotation<StateColor,3>){
        error, no longer have it
    } static else if(A == StateLuma){
        return StateLumaAnnoter(original);
    } else {
        return original.GetAnnoter<A>();
    }
}
}

class StateLumaAnnoter{
private:
    Annoter<CompoundAnnotation<StateColor,3>> compound;

public:
    StateLumaAnnoter(G original):
    compound(original.GetAnnoter<CompoundAnnotation<StateColor,3>>()){}

Get(State state){
    Tuple<double, double, double> val = compound.Get(state);
    return (val<0> + val<1> + val<2>) / 3;
}

typedef Result = AnnotationReplacerGraph;
Result Execute(G graph){
    return Result(graph);
}

Listing 5.13: An example augmentation.
6. CONCLUSIONS

We have presented an architecture for a (semi)automated verification system for LTS-like formalisms with additional annotations. Our background section should provide a useful introduction to the formalism even if the architecture is not needed.

We tried to keep our ideas abstract enough to be implemented in any programming language. However, as may be discerned from the presentation, it was made with fairly rich static typing in mind. The design is made to be capable of taking advantage of the possibilities of correctness checks afforded by such a language.

The design is also suited to being implemented with static polymorphism, which would alleviate the costs of its high level of abstraction. It should, however be taken into account that the template metaprogramming facilities of C++, for example, are notoriously difficult to use. Therefore the implementer should think carefully before fully embracing this approach. The correctness part of the advantages can be attained using languages with generics instead of templates such as C# and Scala.

It may however be wise to study the possibility of using a language like D. It is a system programming language and therefore it has performance comparable to C++ and the power of its template programming system rivals that of C++’s, while having fewer pitfalls and less historical baggage.

If one is truly inclined to embark on adventures in advanced typing, it would be unwise to ignore the advances in the field achieved in functional programming languages, such as Haskell. The design presented in this thesis was not constructed with such an environment in mind, but many of the principles should be highly applicable.

There exist many systems striving to enable automated verification, using various formalisms and approaches with different strengths and weaknesses. By emphasizing the CFFD equivalence we place our system in the same family as the older ARA and TVT systems, which were references to our design.

We improve on both of these in flexibility and efficiency thanks to our in-process blocks and generic annotation infrastructure. An implementation of our architecture would be a powerful tool in exploring this corner of the verification field and thereby another step towards a more safely concurrent tomorrow.
BIBLIOGRAPHY


module main;
import std.stdio;

/// The Annotation target enumeration
enum Targets { State, Transition, Graph}

/// A compile time function for
/// checking if a type is a valid annotation.
/// Returns the answer in member Ok
/// and if there is an error, it
/// is in member error.
template IsAnnotation(A){
    // We don't know what operations exist
    // so we just check that a default implementation
    // exists for unknown operations by using
    // a fake operation Dummy.
    alias Dummy = int;

    // Check that A defines some type in member Type
    static if(!is(A.Type)){
        immutable enum string error =
            "Annotation must have a data type in member Type";
    }
    // Check that A.Target specifies the target of the annotation
    else static if(!is(typeof(A.Target)==Targets)){
        immutable enum string error =
            "Annotation must have a Target.";
    }
    // Check that A has a template function Combinable,
    // which when instantiated is a static function
    else static if(!__traits(isStaticFunction,
        A.Combinable!(Dummy))){
        immutable enum string error =
            "A must have a static function template Combinable.";
    }
    // Use dummy values to create an expression whose type
    // is the return type of A.Combinable
    // to check that it returns a boolean.
    else static if{
        !is( typeof(A.Combinable!(Dummy) (*cast(A.Type*)null, *
            *cast(A.Type*)null))
            ==
            bool
        )}
A. Appendix

```
)}

};

immutable enum string error = "A.Combinable(A.Type, A.Type) must return bool. Returns:
~
typeof(A.Combinable!(Dummy)({*cast(A.Type*)null,
    *cast(A.Type*)null}).stringof;

// A is a valid annotation if there were no errors.
enum bool Ok = !is(typeof(error));

class StateCutState{
    alias Type = bool;
    Targets Target = Targets.State;
    static bool Combinable(Operation)(bool one, bool other){
        return one == other;
    }
}

class NotValid{
}

static assert(IsAnnotation!(StateCutState).Ok,
    IsAnnotation!(StateCutState).error);
static assert(!IsAnnotation!(NotValid).Ok);
static assert(IsAnnotation!(NotValid).error ==
    "Annotation must have a data type in member Type");

void main(string[] args)
{
    writeln("Hello");
}
```
A.2 Example Static Interface in C++

```cpp
#include <iostream>
#include <type_traits>

using std::enable_if;
using std::is_same;
using std::declval;

enum class Targets{
    State, Transition, Graph
};

/// One is different from two in size
typedef char one;
/// Two is different from one in size
typedef struct{char d[2];} two;

/// Compile time function for checking that an annotation has a typedef Type
template<typename A>
class HasType{
    /// Valid only if C::Type is a type
    template<typename C>
    static one test(typename C::Type*);
    /// Low overload priority
    /// => chosen only if nothing else is valid
    /// Is itself always valid.
    template<typename C>
    static two test(...);

    public:
    /// Uses the first test if valid,
    /// we can inspect the return type to
    /// see if it was <= if C::Type is a type
    const static bool value =
        sizeof(test<A>(nullptr)) == sizeof(one);
};

/// Compile time function to check that
/// A has a static member Target
/// and that it's type is Targets
template<typename A>
class HasTarget{
    /// A type that is definitely not Targets
    typedef struct{ Targets d[2]; } NotTargets;

    /// Return type is C::Target's, if it exists
    template<typename C>
    static decltype(C::Target) get(C);

    /// Return type is definitely not
    /// Targets
    template<typename C>
    static NotTargets get(...);

    /// Returns type on if arguments is Targets
};
```
static one test(Targets);
    /// Return two if argument is not Targets
static two test(...);

public:
    /// Get something that is of same type
    /// as C::Target if it is present
    /// and definitely not Targets if
    /// it is not present.
    /// Then give it to test,
    /// which returns one if and only if
    /// C::Target existed and was of type Targets
    const static bool value =
        sizeof(test(get<A>(declval<A>()))) == sizeof(one);
    
    /// Checks that A has a static template
    /// function Combinable with the signature
    /// (A::Type, A::Type) -> bool
    template<typename A>
    class HasCombinable{
        // No real operator known.
        // We check the presence of
        // a default with a Dummy
typedef int Dummy;
        // Definitely not a boolean.
typedef struct{bool d[2];} NotBool;

        // Valid if C::Combinable<Dummy> takes
        // two A::Types as arguments.
        // Return type is the return type of it.
template<typename C>
        static decltype(C::template Combinable<Dummy>                  
            (declval<typename A::Type>(),
             declval<typename A::Type>())) get(C);

        // Valid always, return type is not bool.
template<typename C>
        static NotBool get(...);

        // Check if argument is bool.
        static one test(bool);
        static two test(...);

    public:
        // Use get to get the type
        // of Combinable and
        // use test to make sure it is bool.
        const static bool value =
            sizeof(test(get<A>(declval<A>()))) == sizeof(one);
    
    /// Check if A is a valid Annotation
    template<typename A>
    struct IsAnnotation{
        const static bool value =
            HasType<A>::value &&
HasTarget<A>::value &&
HasCombinable<A>::value;
}

struct StateCutState{
    typedef bool Type;
    const static Targets Target = Targets::State;
    template<typename Operation>
    static bool Combinable(Type a, Type b){
        return a == b;
    }
};

struct NotValid{
    typedef int Type;
    const static bool Target = true;
};

static_assert(IsAnnotation<StateCutState>::value,
    "StateCutState is an Annotation");
static_assert(HasType<NotValid>::value,
    "NotValid has a Type");
static_assert(!IsAnnotation<NotValid>::value,
    "NotValid is not an annotation");

int main(int argc, char** argv){
    std::cout << "Hello " << std::endl;
}