2003

Extending CSP to disambiguate Linda predicate operations

Kyle Burke
Colby College

Follow this and additional works at: http://digitalcommons.colby.edu/honorstheses

Part of the Computer Sciences Commons

Colby College theses are protected by copyright. They may be viewed or downloaded from this site for the purposes of research and scholarship. Reproduction or distribution for commercial purposes is prohibited without written permission of the author.

Recommended Citation
http://digitalcommons.colby.edu/honorstheses/41

This Honors Thesis (Open Access) is brought to you for free and open access by the Student Research at Digital Commons @ Colby. It has been accepted for inclusion in Honors Theses by an authorized administrator of Digital Commons @ Colby. For more information, please contact enhodes@colby.edu.
Extending CSP to Disambiguate
Linda Predicate Operations

Kyle W. Burke
Honors Independent Study

Advisor: Marc L. Smith
Department of Computer Science
Colby College

Spring 2003
Contents

1 Introduction 1

2 Background of Tuple Space and CSP 5
   2.1 Explaining Tuple Space 5
      2.1.1 The Linda Primitives 7
      2.1.2 The Meaning of a Failed Predicate 7
   2.2 Explaining CSP 8
      2.2.1 Processes and Events 8
      2.2.2 Example: Modelling a Stapler in CSP 9
      2.2.3 Combining Processes 10
      2.2.4 Process Traces 11

3 A CSP Model of Tuple Space 13
   3.1 Setting up a Process-Oriented Tuple Space 13
   3.2 The Linda Primitives 15
   3.3 Using this Model to Locate the Ambiguity 20
      3.3.1 Reason for such a Simplification 21
      3.3.2 Attacking the Interleaving 22
      3.3.3 Possible Results 24
4 Parallel Events: The Door to Intermediate-Trace Land 26

4.1 Merge Operator 26
4.1.1 Rules 26
4.1.2 Empty Event 27

4.2 The pieces() Function 27
4.2.1 Precursory Definition 27
4.2.2 Final Definition 28
4.2.3 A Useful Property 28

4.3 The Ropes() Function 29
4.3.1 Ropes() Defined under Perfect Observation 29
4.3.2 Ropes() Defined using Imperfect Observation (First Try) 30
4.3.3 Ropes() (Final Definition) 31

4.4 Bags (a.k.a. Multisets) 31
4.4.1 Motivation 31
4.4.2 Bag Definitions 32
4.4.3 Leaving Bags Behind 32

4.5 How we will use Parallel Events 33

5 Exploring Intermediate-Trace Land 34

5.1 Intermediate Traces 34
5.1.1 Motivated by Ropes and Views 35
5.1.2 A More Meaningful Example 35

5.2 The Intertwine Operator 36
5.2.1 Motivated by Merging 36
5.2.2 Definition without Parallel Events 37
Abstract

The Tuple Space communication environment is plagued by the apparent ambiguity present in the predicate operations of Linda, the programming extension used to implement Tuple Space. It has been shown that by using a method of reasoning which describes only sequentialized traces of events, it is unclear what a failed predicate operation actually means.

Using an operational semantics model of Tuple Space, an analysis of the predicate operations using a description of events occurring simultaneously has disambiguated the meanings of the failure cases. Here an algebraic model is provided, using the Communicating Sequential Process (CSP) process algebra as a base. CSP has been extended already to describe parallel events, but a further extension is created here to allow a concrete system for reasoning about the failure cases important to us. The reasonings performed with these theoretical elements yield similar results to those of the operational semantics model. Under this analysis the meanings of the failures become clear, disambiguating the Linda predicate operations.
Chapter 1

Introduction

The paradigm of parallel and distributed computing carries many interesting problems that do not exist in sequential programming. One of these problems occurs within Tuple Space, a generative computing environment. This problem exists in the form of an ambiguity, where the result of an operation has an unclear meaning. The intent here is to disambiguate this operation, specifically by using an algebraic model to describe and reason about the problem.

Tuple Space is an environment model for communicating processes, which allows information to be sent and received independent of location, time, and platform. Six simple primitive operations are used in Tuple Space to place, remove, copy, and match packets of information, known as tuples, in and from itself. Tuples fall into one of two categories. They are either active, meaning Linda processes residing within them are performing computations, or they are passive, meaning that they can be sought after for matching by the processes of the active tuples.

The six primitive operations, which comprise the language Linda, de-
scribe all communication between the tuples, but do not restrict the computation each tuple can carry out individually. Two of these Linda primitives, however, are ambiguous in their meaning. The two operations, rdp and inp, both search for a passive tuple that matches some template, and either return an indication that they didn’t find it, or return a positive response, along with the matching tuple itself (inp also removes the tuple from Tuple Space, whereas rdp simply copies it out).

The ambiguity lies in the negative response. Presumably, the process calling the operation wants to know whether a matching tuple exists. The predicate operations don’t make the claim that no tuple exists if they can’t find a matching tuple. They only assert that, while searching, they did not encounter such a tuple. Since Tuple Space is ever-changing, we cannot assume that the search is so deterministic as to state that no matching tuples reside in Tuple Space. So, a failed predicate is ambiguous because it can mean more than one thing.

To investigate this ambiguity, we decided to apply the process algebra, Communicating Sequential Processes (CSP) to the Linda primitives and tuple space. CSP is a mathematically-defined system for describing processes and using the traces of their actions to reason about them. Since Tuple Space is a parallel environment, we needed to construct an extension of CSP to directly represent parallel events.

Part of this extension has already been created by Adrian Lawrence, whose work [5] will be explained and used heavily throughout the coming chapters. The rest of the extension we will develop takes into account not only parallel events, but a variety of ways to reason about them, including incorporating aspects of Smith, et. al.’s View-Centric Reasoning [3]. Views of process traces helped us grasp the different situations within tuple space.
that cause the ambiguities. In fact, much of the work presented in Smith’s paper is followed here, except that our theory will be based on the CSP algebra instead of operational semantics definitions. Also, it is our hope that the variety of available discussion on our extension will help in other areas of reasoning for parallel process computations.

Using these tools, we were able to reveal the true meaning of a failed Linda predicate. The perceived ambiguity that existed beforehand dissolves when reasoned about with the extension.

The next chapter of this work is devoted to explaining a background for the areas that will be used. In here there is an explanation of Tuple Space, complete with the Linda operations that are used within. Following that, there is a tutorial section on CSP, describing much of the notation and concepts that will be necessary in understanding this work.

Directly after the tutorial, that knowledge will be used to define Linda and Tuple Space in terms of CSP processes and events. This dense third chapter provides not only the definitions created, but also an explanation of how these different results were created. Using the semantic definitions of Linda and Tuple Space, this implementation should be pleasing to the reader. This chapter continues by attacking the ambiguity with the processes recently defined. It is recognized, at this point, that parallel events will be necessary to continue the evaluation.

In the fourth chapter, a notation to describe parallel events is introduced. This notation is expanded upon, and many properties of the parallel events are devised and explained. Imperfect observation and view-centric reasoning terms are introduced here as well, and aspects from these areas are woven into the notation as well.

Next, the fifth chapter develops an environment which we can use to
reason about the overwhelmingly large set of possible traces that results from even a simple interleaving of processes. Since this problem escalates in the face of parallel events, this new notation will aid our analysis greatly. Operators across traces are added, and more View-Centric Reasoning terms are implemented, as they relate to this new environment.

Finally we will return to assaulting the ambiguity. This time, with the notation described in chapters four and five, the ambiguity is clarified with a small amount of analysis. The meanings of the predicate failures become clear.

As the reader makes this venture through realms of tuples and processes, notations and descriptions, we hope that it becomes possible for the theory in this work to be applied towards other similar problems. Hopefully this CSP extension will be utilized later to solve another issue in modelling concurrent computation. If the reader does not find a use for this, then we at least hope that the coming journey is enjoyable.
Chapter 2

Background of Tuple Space and CSP

To make sense of the coming tutorial, the reader must first be understand some of the details of Tuple Space operations and be able to read and comprehend much of the notation associated with the CSP algebra [2]. With the information presented in this chapter, the reader should be able to navigate subject-specific details of the coming chapters.

2.1 Explaining Tuple Space

Beginning with the details of Tuple Space, we will describe the nature of tuples (both of the active and passive varieties), introduce the four basic Linda primitives, then explain the predicate versions of two of those primitives. These Linda primitive operations are simply the communication procedures that are specific to the architecture of Tuple Space. Extending any computational language to include the Linda predicates makes it possible
to implement Tuple Space with that language.

In Tuple Space, a tuple is represented by a list of fields, delimited by commas, within parentheses. In passive tuples, each field contains a variable, preceded by its variable type. Thus, \((\text{int } 5, \text{string } "\text{monkey}" , \text{float } 10\text{e}5)\) is an example of a passive tuple. Active tuples differ only slightly, in that their fields may contain Linda processes instead of variables. So, if \(H(x)\) and \(G(x)\) are Linda processes, then the tuple \((G(x), \text{string } "\text{monkey}" , \text{float } 10\text{e}5, H(x))\) would be active. Active tuples become passive as soon as their active processes finish computing.

Templates are another object used in Tuple Space. Templates essentially contain the same structure as tuples, except that it is not necessary for their fields to be completed. Some of these fields, instead of containing an instantiated variable, could contain simply a type definition. Since technically tuples only exist as objects suspended in Tuple Space, active processes must use templates as parameters to perform their own computations. Passive tuples are available for matching with these templates, which means that active processes can search for certain tuples based on the types and values within their fields. A template is said to match a passive tuple if every completed field in that template has the same value as the corresponding field of the tuple, and every type definition in the incomplete fields is of the same type as the values in the tuple's corresponding fields. Thus, the templates \((\text{int}, \text{string } "\text{monkey}" , \text{float})\) and \((\text{int } 5, \text{string}, \text{float } 10\text{e}5)\) both match the passive tuple given above. Here tuples will be written with a letter, such as \(t\), whereas templates will be expressed with a letter followed by an apostrophe, such as \(t'\).
2.1.1 The Linda Primitives

The most simple Linda primitives to grasp are \texttt{out()} and \texttt{eval()}. The call of \texttt{out(t')} simply places the passive tuple representing the completed template \texttt{t'} into tuple space. \texttt{eval(t')} does the same thing, except that it drops an active tuple, containing one or more new Linda processes.

The operations \texttt{rd} and \texttt{in} take incomplete templates as parameters, searching Tuple Space for a tuple to match a their given template. When they find a matching tuple, they return that tuple (\texttt{in} also removes that tuple from Tuple Space). While searching, both of these operations block, forcing the active tuple that called them to wait until matching tuples are found.

Sometimes it is undesirable for a Linda process to block for an undetermined amount of time. For this purpose, the predicate operations \texttt{rdp} and \texttt{inp} were created. These primitives search through Tuple Space, but not necessarily until they find a matching tuple. Instead, they could indicate a failed search, meaning that at some point after having not yet found a match to their template, they decided to give up.

2.1.2 The Meaning of a Failed Predicate

When a failure occurs from one of these predicate operations, the result is not strong enough to determine that no tuple matching the given template exists in Tuple Space. This unsuccessful response means either, "There is a matching tuple, but I did not see it," or "there is no matching tuple," but there is no way to decide which of these is correct.

One way to remedy the ambiguity behind a failure-response of \texttt{inp} is to redefine the meaning behind this predicate. A version of \texttt{inp} proposed by
Jacob and Wood [4], exists which is defined so that the process continues to search Tuple Space until it either finds a matching tuple, or the system becomes deadlocked. In the case of deadlock, the tuple will return a failure, and the active tuple which called the predicate operation will continue, effectively breaking the deadlock.

Although this solution is interesting, it does not represent a good definition of inp for systems that wouldn’t expect to see deadlock often, and for systems that don’t want inp to block for an extensive amount of time. A detailed implementation of both predicates will be given later that should appeal to the reader and reflect an intended meaning of “look for a matching tuple, and let me know if you see it.”

2.2 Explaining CSP

To better understand the definitions of Tuple Space operations presented later, as well as the discussions of extensions of CSP, the reader will need to have some background with reading and writing CSP process and trace notation.

2.2.1 Processes and Events

Despite the use of lower-case letters for the Linda primitives, CSP processes are generally denoted with capital letters, and often use subscripts. CSP processes are defined by explaining which events the process will participate in, sometimes influenced by outside events. The progression from one event to another is denoted by an arrow. Thus $a \rightarrow b$ means that first we have the event $a$, followed by the event $b$.

This notation is somewhat meaningless, however, unless also followed by
a process. $a \rightarrow b \rightarrow STOP$ represents a process which participates in the event $a$, followed by $b$, and then halts. We could define a process $P$ as this process, thus,

\[
\begin{align*}
\text{process} \\
P &= a \rightarrow b \rightarrow STOP
\end{align*}
\]

2.2.2 Example: Modelling a Stapler in CSP

To describe the process of a stapler, we could specify the following,

\[
\begin{align*}
\text{process} \\
\text{STAPLER} &= push \rightarrow staple \rightarrow release \rightarrow STAPLER
\end{align*}
\]

This process utilizes recursion, which is a vital part of the majority of CSP definitions, including those we will see in the next chapter. If we wanted to restrict the number of staples in the stapler, we could do so using the following definitions:

\[
\begin{align*}
\text{process} \\
\text{STAPLER} &= \text{STAPLER}_{100} \\
\forall n > 0 : \text{STAPLER}_n &= push \rightarrow staple \rightarrow release \rightarrow \text{STAPLER}_{n-1} \\
\text{STAPLER}_0 &= push \rightarrow release \rightarrow \text{STAPLER}_0
\end{align*}
\]

We can give a process the chance to do different things when different events occur. This is denoted by separating the two paths with a vertical
bar. Currently, our stapler is not overly useful, but if we allow it to be reloaded, then it could be used more than just a hundred times. With a revising of \( \text{STAPLER}_0 \), we develop:

\[
\begin{align*}
\text{process} \\
\text{STAPLER} &= \text{STAPLER}_{100} \\
\forall n > 0: \text{STAPLER}_n &= \text{push} \rightarrow \text{staple} \rightarrow \text{release} \rightarrow \text{STAPLER}_{n-1} \\
\text{STAPLER}_0 &= (\text{push} \rightarrow \text{release} \rightarrow \text{STAPLER}_0) \mid (\text{reload} \rightarrow \text{STAPLER})
\end{align*}
\]

This can, of course, be revised so that the stapler can be reloaded at any point. The above explanation is sufficient to show some of the notation used later.

### 2.2.3 Combining Processes

Two processes can be appended if one completes successfully by making use of the \( \text{SKIP} \) process. Thus, if we define a new \( P \) so that:

\[
\begin{align*}
\text{NEWP} \\
\text{process} \\
\text{NEWP} &= a \rightarrow b \rightarrow \text{SKIP}
\end{align*}
\]

This states that the process \( \text{NEWP} \) completes successfully after participating in the event \( a \) followed by \( b \). Now, we can append this to \( P \) with the notation, \( \text{NEWP}; P \), which is equivalent to the process, \( a \rightarrow b \rightarrow a \rightarrow b \rightarrow \text{STOP} \).

Another way to combine processes is to interleave them. This means that both processes will run, and engage in their events in parallel. For
any two processes, if there are events that occur in one process, but not the other, then only that process needs to participate in that event in order to continue. If both processes participate in the same event, however, then both processes need to accept that event in order to continue computation.

The interleaving of two processes is denoted by a triad of vertical bars between them, such as: $A \parallel B$.

Thus, if we define:

\[\begin{array}{c}
\text{process} \\
\text{STUDENT} = \text{insert.paper} \rightarrow \text{push} \rightarrow \text{release} \rightarrow \\
\text{remove.paper} \rightarrow \text{STOP}
\end{array}\]

We can interleave a student and our stapler to describe the student stapling a paper. So, $\text{STUDENT} \parallel \text{STAPLER}$ (using our very first stapler definition for simplicity) is equal to: $\text{insert.paper} \rightarrow \text{push} \rightarrow \text{staple} \rightarrow \text{release} \rightarrow (\text{remove.paper} \rightarrow \text{STOP} \parallel \text{STAPLER})$. Note that it isn't necessarily true that the student removes the paper before the stapler becomes active again.

### 2.2.4 Process Traces

To reason about processes, we can look at possible traces of events that occur in them up to any given point. A trace of the events that occur from the process $P$ is simply: $(a, b)$, although the traces $(a)$ and $(\emptyset)$ are acceptable also, although they only describe a piece of the computation of $P$.

Traces involving our stapler are slightly more complex, and would look something like: $(\text{push}, \text{staple}, \text{release}, \text{push}, ..., \text{release})$. Traces are a big fo-
**Notation** | **Meaning**
--- | ---
\(a \rightarrow P \mid b \rightarrow Q\) | \(a\) then \(P\) choice \(b\) then \(Q\)
\(P || Q\) | \(P\) interleaved with \(Q\)
\(P \sqcap Q\) | \(P\) or \(Q\) (non-deterministic)
\(P; Q\) | \(P\) successfully followed by \(Q\)
\(P \triangleleft b \triangleright Q\) | \(P\) if \(b\), else \(Q\)
\(b!e\) | channel \(b\) outputs value \(e\)
\(b?e\) | input of \(z\) from channel \(b\)
\(#s\) | length of trace \(s\)
\(s_0\) | head of trace \(s\)
\(s'\) | tail of trace \(s\)
\(s\) in \(t\) | trace \(s\) is within trace \(t\)
\(P/s\) | process \(P\) after engaging in the events of trace \(s\)

Table 2.1: Necessary CSP Notation

As of discussion in our expansion of CSP and their meaning will be further explained.

Other CSP notation exists in this work, and we have provided table 2.1 to be used as a quick reference. This table affords very fast but short explanations, and should the explanations there not be clear, the reader should feel free to reference the notation index of *Communicating Sequential Processes*. 
Chapter 3

A CSP Model of Tuple Space

To investigate examples of failures in Linda predicate operations using event traces, we will now model the six Linda primitives using the CSP process algebra. Once this model is complete, we can generate some of the instances where the failures occur. By examining these cases, we hope to find any ambiguities that may call for an extension of the CSP notation.

We want to assume that the sets of active and passive tuples are unbounded, but have an unbounded size. Thus, we define two sets, active and passive tuples. We will define the sets of active and passive tuples as index sets for our active and passive tuples, respectively. Next, we simplify an index set for our active and passive tuples, respectively, leaving the CSP process algebra.

3.1 Setting up a Process-Oriented Tuple Space

We define the actions of the passive tuples using event traces. However, we proceed with definitions of these operations without first defining a tuple space architecture in CSP. We cannot proceed with definitions of these operations, as well as the predicate operations rdp and inp, without finding a means to represent the Tuple Space architecture in CSP.

The Linda primitives we shall define are the rd, in, out, and eval, operations. The rd operation takes an index value and returns the value of the tuple at that index. The in operation takes an index value and returns the value of the tuple at that index. The out operation takes an index value and returns the value of the tuple at that index. The eval operation takes an index value and returns the value of the tuple at that index.
Note that the \( PTUPLE \) process (3.1) does not exactly represent a passive tuple. Instead it represents a holder for tuples, which will have the form \((t)\) throughout this explanation. Here \( TS \) represents Tuple Space, as an entity, and thus \( TSp \) and \( pTS \) represent channels from Tuple Space to \( PTUPLE_p \) and vice versa, respectively.

The purpose of our \( PTUPLE \) is to wait until it is given a tuple, then sit with that tuple on it's output channel with \( TS \), and once it has given that tuple up, prepare to accept another such tuple. The problem with this current definition, is that multiple tuples can pile up on the \( TSp \) channel before \( PTUPLE_p \) is ready to process them. In order to remedy this problem, we will use a sentinel tuple that will sit on \( pTS \) while \( PTUPLE_p \) is awaiting input. Then we must remember that if this sentinel is not present when another process attempts to place a tuple in the container \( PTUPLE_p \), that process will have to find another place to store that tuple.

Naming our sentinel tuple ("ready"), we now add an initializing process for each \( PTUPLE \) (3.2).

Now, we must simply be careful to preserve the property of passive tuple
processes offering the 'ready' tuple when not holding another tuple. When defining the Linda primitives, we will be certain to incorporate this property.

Also, if we wish to define a process to represent the set of all passive tuples, we can manage that easily, by taking the concurrent composition of all passive tuple processes indexed by the set \( \text{Passive} \) (3.3).

\[
\text{PASSIVE} = \|_{p \in \text{Passive}} \text{PTUPLE}_p
\]

Figure 3.3: \text{PASSIVE}

### 3.2 The Linda Primitives

Having created a basis for our passive tuples, we will begin to interact with them by defining some of the Linda primitives (yes, are neglecting the definition of the active tuples for a while). Our first definition will be the process \( \text{OUT} \) (3.4), which a Linda process uses to place a given tuple into Tuple...
Notice that the process only goes ahead with placing the tuple in the pth PTUPLE if our sentinel is sitting on the output channel. Otherwise OUT proceeds to call itself again, using the non-deterministic choice operator (\(\sqcap\)) to try again with another passive tuple. This definition, however, does not describe from where the tuple \((t)\) has come. Tuple Space must have received \((t)\) from an active tuple. Since we want the Linda primitive out to be equivalent to the process OUT, we need to fix our definition to create an OUT that describes which active tuple contains the process that called this operation (3.5).

One down, five to go!

Next we will try our hand at defining the Linda in primitive (3.6). This operation searches Tuple Space for a tuple matching a given template until it finds one. Once it does, it removes that tuple from Tuple Space, and returns it to the tuple containing the searching process. Our symbol for a template, similar to a tuple, shall be: \((t')\), which may satisfy the predicate,
The event \( TSa(t) \) corresponds to the sending of tuple \((t)\) to active tuple \(a\), where it will be handed back to the process that called the search.

Our \( INP \ (3.7)\) should act just as the Linda operation does. Thus, its actions will look remarkably like \( IN \), except that the returned signal will
return an indicator of whether the search failed (noted by the character ‘0’) or succeeded (‘1’), as well as either the original template, or a matching tuple. Thus definition uses a bit of set notation, and we will assume that $S$ is a set, which is also a subset of $\text{Passive}$.

\[
\text{process}
\]

\[
\forall a \in \text{Active} : \text{INP}_a(t') = \text{INP}_a(t').\text{Passive}
\]

\[
\forall a \in \text{Active} : \text{INP}_a(t').S = \cap_{p \in S} pTS?(t) \rightarrow
\]

\[
(TSp!(\text{"ready"}) \rightarrow TSA!1.(t) \rightarrow \text{SKIP})
\]

\[
\not \exists t \text{ matches } t'
\]

\[
(TSp!(t) \rightarrow
\]

\[
(TSa!0.(t') \rightarrow \text{SKIP})
\]

\[
\not \exists S = \{p\} \not \\
\]

\[
(\text{INP}_a(t').(S - \{p\}))
\]

---

**Figure 3.7: INP**

In a similar fashion we will define both the Linda \text{rd} and \text{rdp} operations (3.8, 3.9). They are similar to their \text{in} counterparts, except that they don’t remove the tuple from Tuple Space once they make a match. This actually makes the definitions slightly more simple than \text{IN} and \text{INP}.

And then, before you know it, we’ve defined five of the six Linda primitives! One remains: the \text{eval} operation, which is the operation that allows Linda processes to spawn new processes. Since this basically places an active tuple into Tuple Space, we should treat this primitive in a similar manner.
to the out operation we defined earlier. In the same manner, however, we should define the wrapper process that will be assigned to dealing with each active tuple. Remembering that Active is already an index for our set of tuples, ATUPLE should not be too difficult to manage (3.10).

This COMPUTE process is simply the concurrent composition of processes running within the fields of tuple \( t \). Once COMPUTE finishes (meaning that all the individual Linda processes within have completed their computations), the active tuple simply creates a new passive tuple using the OUT process, and the wrapper process prepares to accept another active tuple. Notice that the same 'ready' sentinel is used.

Of course, we want to define an ACTIVE process (3.12), just as we have with PASSIVE.

The eval operation should now not be difficult to model. We more or less want to do the same thing as with OUT, since the ATUPLE process should take care of the rest of the computations (3.13).

This EVAL has an identical structure to OUT, which possibly forces the
need for the special case in \textit{ATUPLE}. This case occurs when the \textit{ATUPLE} process is handed a 'ready' tuple, and instead of computing it, just turns around and offers the tuple back on its output channel.

While this special case may not be necessary, as the computing and then 'outing' of the 'ready' tuple should be negligible, it has been included for the moment for simplicity. There is every chance, that as this notation evolves, this special case will simply be removed.

To put the final touches on this explanation, we offer a final, simple process description (3.14), reflective of figure 3.11.

3.3 Using this Model to Locate the Ambiguity

To look at the ambiguity with prior CSP notation, we will present the most simple of cases, an interleaving of four processes:
process

\[ \forall a \in \text{Active} : \text{ATUPLE}_a = aTS!("ready") \rightarrow \text{ATUPLE}_a \]

\[ \forall a \in \text{Active} : \text{ATUPLE}_a = TSa?(t) \rightarrow \]

\[ (\text{COMPUTE}_a(t) : \text{OUT}_a(t) : aTS!("ready") \rightarrow \text{ATUPLE}_a) \]

\[ \uparrow (t) \neq ("ready") \uparrow \]

\[ (aTS!("ready") \rightarrow \text{ATUPLE}_a) \]

Figure 3.10: ATUPLEI, ATUPLE

\[
\text{OUT2}(t) \parallel \text{INP}_a(t').\{p\} \parallel (pTS!("ready") \rightarrow \text{PTUPLE}_p) \parallel (TSa?m \rightarrow \text{MOREATUPLE}_a) \]

where \( t \) matches \( t' \).

3.3.1 Reason for such a Simplification

We have simplified things greatly, by forcing the \( \text{INP} \) process to look at the passive tuple \( p \). We will impose the same restriction on \( \text{OUT} \). In any other example, the interleavings of these processes is practically irrelevant. Also note that the passive tuple \( p \) is offering ("ready") and will be able to accept \( t \) if it gets the chance. The last piece of the interleaving is simply the waiting of a process within the active tuple \( a \) for the result of the \( \text{INP} \), and then the continuation of that process. We will forget about this last part until this interleaved process returns either a success or failure to \( a \), since it is simply awaiting this information.

Since we are looking to investigate the ambiguity, we will consider only the case where \( \text{INP} \) returns the failure. Also, to resolve the interleaving, we
need to start by looking at the first events in each of these process definitions.

3.3.2 Attacking the Interleaving

The first event that will result from the OUT process is $pTS?(r)$ (remember, we are forcing the process to act on $p$), and the first from INP is $pTS?(q)$. So we can rewrite our interleaving (leaving the fourth part out, since it won't be important until the very end) as:

$$(pTS?(r) \rightarrow OUT2(t)/(pTS?(r))) || (pTS?(q) \rightarrow INP_{p}(t').{p}/(pTS?(q))) || (pTS!("ready") \rightarrow PTUPLE_{p}).$$

**Basically,** there are two queries competing for ("ready") which is residing on the out channel for $PTUPLE_{p}$. Since we are only concerned about the failing case, we will need the INP query to occur next. Thus by the
process

\[ ACTIVE = \big\|_{a \in \text{Active}} \text{ATUPLE}_a \big\]  

Figure 3.12: ACTIVE

process

\[ \forall a \in \text{Active} : EVAL_a(t) = aTS!(t) \rightarrow EVAL2(t) \]

\[ EVAL2(t) = \bigwedge_{a \in \text{Active}} pTS?(r) \rightarrow \]

\[ (TS{p}!(t) \rightarrow \text{SKIP}) \]

\[ r = \text{"ready"} \]

\[ (TS{r}!(r) \rightarrow EVAL2(t)) \]

Figure 3.13: EVAL, EVAL2

associativity of \( \| \), we can rearrange our expression to the following:

\[ ((pTS?(q) \rightarrow \text{INP}_o(t').\{p\}/(pTS?(q))) \| (pTS!(\text{"ready"}) \rightarrow \text{PTUPLE}_p)) \]

\[ \| (pTS?(r) \rightarrow \text{OUT}2(t)/(pTS?(r))). \]

Which is equal to:

\[ (pTS!(\text{"ready"}) \rightarrow (\text{INP}_o(t').\{p\}/(pTS?(q)) \| \text{PTUPLE}_p)) \| (pTS?(r) \rightarrow \text{OUT}2(t)/(pTS?(r))). \]

Then, since the query for the OUT has to wait, the event \( pTS!(\text{"ready"}) \) will be chosen, and, since \( (\text{"ready"}) \) does not match the template \( t' \), we can proceed to the failure case from the INP process. With similar steps as
process

\[ TS = ACTIVE \parallel PASSIVE \]

Figure 3.14: TS

these, we will continue to evaluate this process interleaving.

\[
pTS.("ready") \rightarrow ((TS!("ready") \rightarrow TSa!0.(t') \rightarrow SKIP) \parallel (TS?.(s) \rightarrow pTS!(s) \rightarrow PTUPLE_p)) \parallel (pTS?(r) \rightarrow OUT2(t)/(pTS?(r)))
\]

\[ = pTS.("ready") \rightarrow TSp.("ready") \rightarrow (TSa!0.(t') \rightarrow SKIP \parallel pTS!"ready") \rightarrow PTUPLE_p \parallel pTS?(r) \rightarrow OUT2(t)/(pTS?(r)) \]

\[ = pTS.("ready") \rightarrow TSp.("ready") \rightarrow pTS.("ready") \rightarrow (TSa!0.(t') \rightarrow SKIP \parallel TSp?(s) \rightarrow pTS!(s) \rightarrow PTUPLE_p \parallel TSp!(t) \rightarrow SKIP) \]

It is at this point that we want to bring our ATUPLE piece back into the interleaving, when then looks like:

\[
pTS.("ready") \rightarrow TSp.("ready") \rightarrow pTS.("ready") \rightarrow (TSa!0.(t') \rightarrow SKIP \parallel TSp?(s) \rightarrow pTS!(s) \rightarrow PTUPLE_p \parallel TSp!(t) \rightarrow SKIP \parallel TSp?m \rightarrow MOREATUPLE_a) \]

3.3.3 Possible Results

Now there are two possibilities that could result. Either a communication across TSa could occur, relaying the failure, or the tuple t could be placed into the passive tuple p. Or, in the case that we are most interested in, the two events could happen at the same time.
This is a possibility, since Tuple Space should be implemented as a parallel environment. Currently, CSP has a method for dealing with this sort of occurrence. When recording a trace of the events, if two or more events occur simultaneously, the observer simply writes them down in either order.

3.4 A Glimpse of the Ambiguity

Suppose we are given the following trace, derived mostly from the work done above:

\[(pTS.("ready"), TSp.("ready"), pTS.("ready"), TSp.(t), TSa.0.(t')).\]

Here, the ordering of these last two events causes a bit of a problem. It could be that they occurred in this order, and that the failed return doesn’t quite make sense, because the tuple \(t\) was waiting in a passive tuple container at that point. If, however, they occurred simultaneously, then the failure return makes somewhat more sense, because although the tuple is being accepted by the container, it is not quite available for matching at that point.

If there were a way to describe, in an event trace, parallel events in traditional CSP, then this would not be an issue, because then we could clearly discern the order of events. In the following chapters, new CSP tools will be described which will allow us to remove this level of ambiguity.
Chapter 4

Parallel Events: The Door to Intermediate-Trace Land

In this chapter we will begin to define a helpful extension to CSP. Here we introduce the notion of parallel events, and will generate a solid set of theory to make this notion concrete.

4.1 Merge Operator

Our first step in including parallel events into our theory will be to look at the notation that we will use. The merge operator, introduced by Lawrence [5], can be used on a pair of events (and as we shall see, also with more than two), to indicate that they occur in parallel, as one event.

4.1.1 Rules

Laws that apply for this operator are as follows (the _ denotes an empty event, which represents the lack of observing an event):
\[ a \circ_\_ = a \text{ (Identity)} \]
\[ a \circ b = b \circ a \text{ (Commutative)} \]
\[ (a \circ b) \circ c = a \circ (b \circ c) \text{ (Associative)} \]

This last property removes the need for parentheses in these situations.

We can just as easily describe a “large” parallel event consisting of several “event-pieces” as:
\[ a_1 \circ a_2 \circ ... \circ a_n \]

It is worthwhile to note that \( a \circ a \neq a \), because \( a \circ a \) represents two simultaneous occurrences of event \( a \). This is clearly different from just a single occurrence of \( a \).

### 4.1.2 Empty Event

The empty event, \( \_ \), also requires further explanation. As stated above, it represents the lack of observing an event. It could be that no event occurred, or that an event occurred which was simply “missed” by the observer. In either case, its meaning stands that it represents the lack of observation of an event. The latter possibility, however, further suggests that we do not have observers who are perfect at recording events into a trace. This possibility of imperfect observation is extremely useful when modelling real systems, and this idea will be expounded upon as we continue.

### 4.2 The \( \text{pieces()} \) Function

#### 4.2.1 Precursory Definition

If we wish to describe a method for accessing the different singular events within a parallel event, we may want a function to generate the set of all
possible "sub-events" for a given event. To make sure that this function will be compliant with our old notation, we should define it under the scope of non-parallel events first:

\[ \text{pieces}(a) = \{a, \_\} \text{ where } a \text{ is an atomic event.} \]

The inclusion of the empty event exists because, by our first property, the empty event can be considered a piece of every event. Thus we must consider, even in the realm of non-parallel events, that any event occurs in parallel with the empty event. It is at this point, then, that we should remove the distinction between an "event" and a "parallel event". Our next definition (allowing parallel events) will incorporate this consideration as well:

\[ \text{pieces}(a \circ b) = \{\_, a \circ b\} \cup \text{pieces}(a) \cup \text{pieces}(b) \]

4.2.2 Final Definition

The following definition is probably most preferable, however.

Let \( c = c_0 \circ c_1 \circ \cdots \circ c_n \), such that \( c_i \) is an atomic event, \( 0 \leq i \leq n \).

Then \( \text{pieces}(c) = \{c_{j_0} \circ c_{j_1} \circ \cdots \circ c_{j_m} \mid \forall (j_0, j_1, \ldots, j_m) \subset [0, n] \exists j_k < j_{k+1} \forall k \in [0, m - 1]\} \).

4.2.3 A Useful Property

This last definition is pretty solid. With it, we want to make the following claim:

\[ a \in \text{pieces}(b) \land b \in \text{pieces}(c) \Rightarrow a \in \text{pieces}(c). \]

We can prove this without too much effort. Since \( b \in \text{pieces}(c) \), there exist some \( c_{j_0}, c_{j_1}, \ldots, c_{j_m} \), such that \( b = c_{j_0} \circ c_{j_1} \circ \cdots \circ c_{j_m} \). Now, since \( a \in \text{pieces}(b) \), we know that there exist some \( c_{k_0}, c_{k_1}, \ldots, c_{k_q} \) such that \( q \leq
$m \land \forall i \in [0, q] : \text{there exists } i' \in [0, m] \text{ such that } k_j = j_{i'} \land i \leq i' \land a = c_{k_0} \circ c_{k_1} \circ \cdots \circ c_{k_q}$. But, clearly, this $c_{k_0} \circ c_{k_1} \circ \cdots \circ c_{k_q}$ is an element of $\text{pieces}(c)$. Thus $a \in \text{pieces}(c)$, and our claim has been proven.

### 4.3 The $\text{Ropes}()$ Function

Continuing, as Smith describes, it is often necessary to create an ordering across the atomic subevents of a given parallel event [3]. Thus, given an event $a$, we wish to define a function that will generate all possible orderings. Each ordering is referred to as a Randomly Ordered Parallel Event (ROPE), which gives us a quick name for a new function: $\text{Ropes}$. This function will generate a set of all possible orderings of the event’s atomic pieces, each in the form of a trace.

#### 4.3.1 $\text{Ropes}()$ Defined under Perfect Observation

This definition for $\text{ropes}$ may not be obvious at first, but an explanation follows directly afterwards that should be suitable:

$$\text{Ropes}(a) = \{ (b_0, b_1, \ldots, b_m) \mid b_0 \circ b_1 \circ \cdots \circ b_m \in \text{pieces}(a) \land b_k \text{ is atomic } \forall k \in [0, m] \}.$$  

The necessary element to understanding this definition fully is that there are two levels of elements (traces) of $\text{Ropes}$ that are getting generated here. The first is explicit, by way of the generation of events by the $\text{pieces}$ function. The second is somewhat less obvious, and revolves around the fact that $\circ$ is a commutative operator. Thus, if $a \circ b \in \text{pieces}(c)$, then it must also be true that $b \circ a \in \text{pieces}(c)$. Thus, every permutation of any trace found within the set resulting from $\text{Ropes}$ will also be an element of that set.

This factor, implicit by the set definition we are using, could be more
obvious if we pretend that we have created a function, \textit{permutations}, which returns a set containing all the permutations of a given event, in terms of its atomic sub-events joined by the \textit{o} operator. Now, we can see that every element of the resulting set is equal to the given event. For example, if \( a, b \) and \( c \) are atomic events, then \( \text{permutations}(a \circ b \circ c) \) would return the set \( \{a \circ b \circ c, a \circ c \circ b, b \circ a \circ c, b \circ c \circ a, c \circ a \circ b, c \circ b \circ a\} \), and each element of this resulting set is equal to the initial \( a \circ b \circ c \). Thus, saying that something is an element of \( \text{permutations}(a) \) for a given \( a \) is equivalent to saying that it is equal to \( a \).

We may already be able to see our definition forming. Perhaps we want to use \textit{permutations}, defining \textit{Ropes} as: \( \text{Ropes}(a) = \{(b_0, b_1, ..., b_m) \mid b_0 \circ b_1 \circ \cdots \circ b_m \in \text{permutations}(a) \land b_k \text{ is atomic } \forall k \in [0, m]\} \).

But, from above, to say that something is one of the permutations of an event is to say that it is equivalent to that event. Thus, we can just use equivalence of the two events instead and sidestep any formal definition of the \textit{permutations} function completely, as shown in our initial \textit{Ropes()} definition.

\subsection*{4.3.2 \textit{Ropes()} Defined using Imperfect Observation (First Try)}

We must, however, take into account the possibility of imperfect observation, as recognized by Smith [3]. Thus, we cannot just use:

\( \text{Ropes}(a) = \{(b_0, b_1, ..., b_m) \mid b_0 \circ b_1 \circ \cdots \circ b_m = a \land b_k \text{ is atomic } \forall k \in [0, m]\} \),

but instead we must utilize the \textit{pieces} function on our input for \textit{Ropes}, giving us:

\( \text{Ropes}(a) = \{(b_0, b_1, ..., b_m) \mid b_0 \circ b_1 \circ \cdots \circ b_m \in \text{pieces}(a) \land b_k \text{ is atomic} \)
$\forall k \in [0, m]$.

Unfortunately, even this definition will not suffice, mostly because it summons forth an infinite number of sets, no matter what the properties of the input are. This occurs because of another of our properties of $\circ$, namely that any event occurring with the empty event is still equal to itself. Thus, if \langle a, b \rangle is generated by $\text{Ropes}$, then not only is \langle b, a \rangle as well, but also \langle -, a, b \rangle, \langle a, - b \rangle, \langle a, b, - \rangle, \langle a, - , - b \rangle, and so forth, which alone is an infinite number of traces.

4.3.3 $\text{Ropes}(\cdot)$ (Final Definition)

The best solution to this problem is probably to simply remove the empty event from having a place in any of these traces, which gives us this final definition:

$$\text{Ropes}(a) = \{(b_0, b_1, \ldots, b_m) | b_0 \circ b_1 \circ \cdots \circ b_m \in \text{pieces}(a) \land b_k \text{ is atomic } \forall k \in [0, m] \land b_i \neq - \forall i \in [0, n]\}. $$

4.4 Bags (a.k.a. Multisets)

4.4.1 Motivation

As a substitute to much of the notation we have developed in this chapter, we could be using a multiset (or bag) notation to model parallel events. By drawing out the connection in this section, we hope to give the reader an alternative method to reason about the structure of parallel events. The difference between sets and multisets is that sets cannot contain repetitions of any element, while multisets can consist of repeated elements. Thus, while $\{a, a, b\}$ is not a valid set, it is a valid multiset (the reader may have noticed
that the set we described generated by \textit{permutations} isn't really a valid set either, although it could instead be described as a multiset). Like sets, multisets are unordered, so we can naturally use them to describe parallel events. In essence, since an event can contain two or more instances of a given subevent, it is a bag of its subevents.

4.4.2 Bag Definitions

Pretend, for a moment, that we aren't using set notation, but instead multiset notation. Now, modelling parallel events as multisets of their subevents can be used to simplify our definitions of \textit{pieces()} and \textit{Ropes()}. Thus the event \(a \circ b \circ a \circ c\) may be represented with the bag \(\{a, b, a, c\}\) (remember that the order doesn't really matter), and our previous definitions for \textit{pieces()} and \textit{Ropes()} are replaced by the very elegant and concise following definitions:

\[
\text{pieces}(a) = \{b \mid b \subseteq a\}, \text{ and}
\]

\[
\text{Ropes}(a) = \{(b_0, b_1, \ldots, b_n) \mid \{b_0, b_1, \ldots, b_n\} \subseteq a \land b_i \text{ is atomic } \forall i \in [0, n]\}.
\]

4.4.3 Leaving Bags Behind

Bags provide an alternative to our merge symbol for simple multiset notation features, which on one hand allow our assertions to take a more general approach. The problem with this, however, is that we cannot simply pretend that we are not using set notation. Traditional mathematical notation is an important aspect of CSP, and is used widely throughout the Tuple Space definitions we have already generated. Thus, in order to represent and use this notation, we would have to generate a new spread of notation specific to multisets. If we didn't write up a separate batch of notation, then suddenly

32
we don't know what to do when given an expression such as: \( \{a\} \cup \{a\} \).

Thus, although we can represent each parallel event as a bag of its pieces, we will continue using the merging notation presented earlier in this chapter. Hopefully, this abstraction of looking at events as bags is extremely helpful, and the reader should keep this model in mind throughout the following chapters if confusion arises concerning the use of event merging.

4.5 How we will use Parallel Events

The modelling of parallel events in CSP creates a much greater complexity when reasoning about traces of processes. The next chapter will introduce another layer of theory and notation which should aid in simplifying the analysis of complex calculations including parallel events. Just as we have done here with the pieces() function, the next chapter will continue to support imperfect observation, as well as introduce concepts prevalent in the field of View-Centric Reasoning.

With this theory we should be able to pull apart the ambiguity that is caused by reasoning with sequentialized traces.
Chapter 5

Exploring

Intermediate-Trace Land

Having defined notation to describe the phenomenon of events occurring simultaneously, we will now explore another tool to help us investigate possible traces that can occur, given a process expression. We want this level of notation to exist somewhere between process definitions and their many possible resulting traces. Intermediate-Trace Land will deliver a means of generating and reasoning about the sets of traces that result from parallel and concurrent computations. Also, the facets of this new notation must be logically sound, meaning that we can prove the mathematical properties that we will need to rely upon.

5.1 Intermediate Traces

In this system, we will denote intermediate traces in the following manner: \((a, b)\), where \(a\) and \(b\) are events. Although this will be different from a
trace, we will refer to intermediate traces as just traces, unless otherwise specified.

One of the facets of the flexibility of these traces will be to allow other traces to be treated as events, where this was not previously possible. In the following repetitive trace, we have represented a part of it as an inner trace. This trace,

$$\langle \ldots \langle \text{coin, coin, choc} \rangle, \langle \text{coin, coin, choc} \rangle, \ldots \rangle$$

will describe the workings of a vending machine, where a whole iteration of the apparent underlying process, which appears to be something like:

$$VM = \text{coin} \rightarrow \text{coin} \rightarrow \text{choc} \rightarrow VM,$$

is described in one event.

5.1.1 Motivated by Ropes and Views

While it is not apparent at all how these intermediate traces are useful, it will become clear when describing traces which include parallel events. Here, the flattening of the above trace can be applied without changing the meaning of the trace. It also follows, however, that one of these inner traces could be created using the \textit{Ropes} function on a parallel event, as defined earlier. Since ROPEs are critical to creating Views, it already appears that we could use these intermediate traces (without flattening, which would remove the context of the underlying ROPEs) to generate views for a given trace (with or without parallel events) or an intermediate trace. We will go further into generating views later.

5.1.2 A More Meaningful Example

Suppose now, that we take a more simple chocolate machine:
\[ VMC = coin_1 \rightarrow choc \rightarrow VMC \]

and another machine which dispenses only orange juice:
\[ VMO = coin_2 \rightarrow juice \rightarrow VMO. \]

Now, examining \( VMC \parallel VMO \), some portions of possible traces are:

\[
\begin{align*}
(...coin_1, choc, coin_2, juice, ...) \\
(...coin_1, choc \circ coin_2, juice, ...) \\
(...coin_1, coin_2, choc, juice, ...) \\
(...coin_1, coin_2, choc \circ juice, ...) \\
(...coin_1, coin_2, juice, choc, ...)
\end{align*}
\]

5.2 The Intertwine Operator

Of course, more traces exist, but if we wanted to examine the set of all possible traces without enumerating each of them, we may want to introduce an "intertwining" operator (\( \circ \)) over traces to be used like:

\[
(\langle ... coin_1, (choc) \circ (coin_2, juice), ... \rangle)
\]

which is an intermediate trace representing all possible traces that could be derived from interleaving the events of the two traces together.

5.2.1 Motivated by Merging

Although this intertwine operator appears to be far removed from the meaning of the merge (\( \circ \)) operator, it will be used specifically to complement event merging. The important thing to note is that \( \circ \) retains the order of events \( coin_2 \) then \( juice \), which occur in every case of our list of possible traces.

It could be that we want to look at more examples, and they might be described by something like:

\[
(\langle ... (coin_1, choc) \circ (coin_2, juice), ... \rangle)
\]

36
which could represent a customer who wants both chocolate and juice, but will only operate the machines when he knows no one else will.

5.2.2 Definition without Parallel Events

If we restrict our gaze to a system that does not allow parallel events to take place, the definition of ♦ is straightforward. One working definition could be:

\[ a♦b = \{ t \mid a \text{ in } t \land b \text{ in } t \land \#t = \#a + \#b \} \]

where \( a \) and \( b \) are traces.

Here we are just explaining that all resulting traces must contain \( a \) and \( b \), but contain nothing aside from them. Unfortunately this also assumes that \( a \) and \( b \) are mutually exclusive, meaning that they don't contain any events that exist in the other.

Instead of reworking this definition to fit our model, we will create another definition for the ♦ operator which will utilize recursion. Assuming still that we want to keep parallel events out of our traces, we can form another equivalent definition for our operator, as shown below.

\[ a♦() = a \]

\[ a♦b = \{ (a_0)^\neg t \mid t \in a♦b \} \cup \{ (b_0)^\neg t \mid t \in a♦b' \}, \]

where \( a \) and \( b \) are traces.

5.2.3 Definition Assuming Perfect Observation

Now it may already be apparent to the reader how we can expand the second part here to include parallel events. Our result will simply "tack on" another piece to our collective union.

\[ a♦() = a \]

\[ a♦b = \{ (a_0)^\neg t \mid t \in a♦b \} \cup \{ (b_0)^\neg t \mid t \in a♦b' \} \cup \{ (a_0•b_0)^\neg t \mid t \in a♦b' \}, \]
where $a$ and $b$ are traces.

**5.2.4 Final Definition**

In the case where we want to represent imperfect traces (because of the possibility of imperfect observation), we should make our notation compatible. Thus, we will use the *pieces* function in a third definition:

$$a \triangleright b = \{(c)^{-}t \mid t \in a' \triangleright b \land c \in \text{pieces}(a_{0})\} \cup \{(c)^{-}t \mid t \in a \triangleright b' \land c \in \text{pieces}(b_{0})\} \cup \{(c)^{-}t \mid t \in a' \triangleright b' \land c \in \text{pieces}(a_{0} \circ b_{0})\},$$

where $a$ and $b$ are traces.

**5.3 The Intertwine Operator, Properties and Proofs**

**5.3.1 Basic Properties**

Since the *pieces* function has been defined for systems with and without parallel events, we will refer to this final definition unless otherwise specified. Now we want to inspect some properties of the $\triangleright$ operator:

$$a \triangleright b = b \triangleright a \quad \text{(Commutative)}$$

A proof of this is trivial. A proof of the following statement, however, is not quite so simple, and we will present it directly afterwards.

$$ (a \triangleright b) \triangleright c = a \triangleright (b \triangleright c) \quad \text{(Associative)}$$

This notion seems simple. We wish to use the intertwine operator across all three traces, to intertwine them, but leaving their respective orders intact (and allowing imperfect observation, as it may be).
5.3.2 Overloading the Intertwine Operator

Notice that the associativity property assumes that we can use this operator on a trace and a set requires that we overload $\odot$. Thus, we first allow intertwine to be defined over sets of traces:

$$S \odot T = \{ s \odot t \mid s \in S \land t \in T \},$$

where $S$ and $T$ are sets of traces.

This definition clearly also retains the commutative property. Also, however, it makes for a simple definition relating a trace and a set of traces:

$$S \odot t = t \odot S = \{ t \} \odot S = S \odot \{ t \}$$

where $S$ is a set of traces, and $t$ is a trace.

5.3.3 Proof of Associativity

Now, our next objective will be to elaborate on the expression on one side of our equation.

$$(a \odot b) \odot c = (a \odot b) \odot \{ c \}$$

$=$

$\{ s \odot c \mid s \in (a \odot b) \}$

$=$

$\{ ((d)^{-1} s) \odot c \mid s \in (a' \odot b) \land d \in \text{pieces}(a_0) \} \cup$

$\{ ((d)^{-1} s) \odot c \mid s \in (a' \odot b') \land d \in \text{pieces}(b_0) \} \cup$

$\{ ((d)^{-1} s) \odot c \mid s \in (a' \odot b') \land d \in \text{pieces}(a_0 \circ b_0) \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(a_0) \land r \in (a \odot b) \odot c' \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(a_0) \land r \in (a' \odot b) \odot c \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(a_0 \circ a_0) \land r \in (a' \odot b) \odot c' \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(b_0) \land r \in (a' \odot b') \odot c \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(b_0 \circ a_0) \land r \in (a' \odot b') \odot c' \} \cup$

$\{ (e)^{-1} r \mid e \in \text{pieces}(a_0 \circ b_0 \circ a_0) \land r \in (a' \odot b') \odot c' \}$

While this mess may appear to be undecipherable, we can see, by looking at the recursive use of the intertwine operator here, that it might be possible
to structure a sort of inductive proof across the length of the traces. In order to complete this step, however, it would be necessary to assume \((a\oplus b)\oplus c' = a\oplus (b\oplus c')\) as well as the rest of the permutations that appear in our expansion above. Before we lose focus too far, we should start at the beginning of an inductive proof: the base cases.

Base case 1: \( (\langle 0 \rangle \langle 0 \rangle) \langle 0 \rangle \)

\[= \langle 0 \rangle \langle (\langle 0 \rangle \langle 0 \rangle) \rangle \]  
because \(\langle \rangle\) is commutative.

Base case 2: \( (\langle a_0 \rangle \langle 0 \rangle) \langle 0 \rangle \)

\[= \langle a_0 \rangle \langle (\langle 0 \rangle \langle 0 \rangle) \rangle \]  
by a double usage of our base case definition of \(\langle \rangle\).

Base case 3: \( (\langle a_0 \rangle \langle b_0 \rangle) \langle 0 \rangle \)

\[= (\langle a_0 \rangle \langle b_0 \rangle) \]

\[= (\langle a_0 \rangle \langle (b_0) \rangle \langle 0 \rangle \)

Base case 4: \( (\langle a_0 \rangle \langle b_0 \rangle) \langle c_0 \rangle \)

\[= \{ (x, y), (y, z), (x \circ y) \mid x \in \text{pieces}(a_0) \land y \in \text{pieces}(b_0) \} \langle a_0 \rangle \]

\[= \{ (x, y), (x, y \circ z), (y, z), (x, z), (x, y, z), (x \circ y, z), (x, z, y), (x, y \circ z) \}

\[\mid x \in \text{pieces}(a_0) \land y \in \text{pieces}(b_0) \land z \in \text{pieces}(c_0) \} \}

\[= \{ (y, z), (y, z \circ x), (y, z), (x \circ y, z), (x, y, z), (y, z) \}

\[\mid y \in \text{pieces}(b_0) \land z \in \text{pieces}(c_0) \} \langle a_0 \rangle \]

\[= (\langle a_0 \rangle \langle (b_0) \rangle \langle c_0 \rangle) \}

These four cases are sufficient for us to make seven assumptions which will enable us to prove the inductive step. Thus our inductive hypothesis asserts that:

\[ (a' \oplus b') \oplus c' = a' \oplus (b' \oplus c') \]

\[ (a' \oplus b') \oplus c = a' \oplus (b' \oplus c) \]

\[ (a' \oplus b) \oplus c' = a' \oplus (b \oplus c') \]

\[ (a \oplus b') \oplus c' = a \oplus (b' \oplus c') \]

\[ (a' \oplus b) \oplus c = a' \oplus (b \oplus c) \]
$$\text{Looking back on our expansion of } (ab')c \text{, we will now replace all instances which fit into the seven above assumptions with their equivalents, and remove extraneous parentheses:}$$

$$\begin{align*}
&= \{(e)^r | e \in \text{pieces}(a_0) \land r \in a\diamond (b'c') \} \cup \\
& \{(e)^r | e \in \text{pieces}(a_0) \land r \in a'(b\diamond c) \} \cup \\
& \{(e)^r | e \in \text{pieces}(a_0 \circ a_0) \land r \in a'(b'c') \} \cup \\
& \{(e)^r | e \in \text{pieces}(b_0 \circ a_0) \land r \in a'(b\diamond c) \} \cup \\
& \{(e)^r | e \in \text{pieces}(a_0 \circ b_0) \land r \in a'(b'\diamond c) \} \cup \\
& \{(e)^r | e \in \text{pieces}(b_0 \circ a_0 \circ a_0) \land r \in a'(b'\diamond c) \} \cup \\
& \{(e)^r | e \in \text{pieces}(b_0 \circ a_0 \circ a_0) \land r \in a'(b'\diamond c) \}
\end{align*}$$

Rearranging the pieces, and following the reverse of our above steps to generate the mess, we can actually simplify this to:

$$= a\diamond (b\diamond c)$$

Thus we can see that we can use the $\diamond$ across a collection of traces, and not just a pair. (We can rewrite the two sides of the above equation as simply: $a\diamond b\diamond c$.)

### 5.4 Intertwining and View-Centric Reasoning

#### 5.4.1 Alternate Ropes Definition (unproven)

If we investigate our definition of the $\text{Ropes()}$ function, it returns a set of traces, given an event. Since our intertwine operator also returns sets of traces, it seems possible that we could use this to redefine $\text{Ropes()}$.

So, given an event, we wish to produce a trace (or intermediate trace) containing all orderings of the atoms of that event. We can do this by placing
each of those atomic events into their own traces, and then use the operator on the whole set of them. Thus:

\[ Ropes(a) = \langle a_0 \rangle \cdot \langle a_1 \rangle \cdot \ldots \cdot \langle a_n \rangle, \]

where \( a_0 \circ a_1 \circ \ldots \circ a_n \in \text{pieces}(a) \) and \( a_i \) is atomic \( \forall i \in [0, n] \) and \( a_i \neq -\forall i \in [0, n] \).

5.4.2 Moving on to Views

We began to talk about views before, and how we might be able to represent them using intermediate traces and the \( Ropes \) function. A given view can be expressed as a(n intermediate) trace, where each element is, itself, a trace. Each of these inner traces will represent a ROPE from a parallel event. Thus, if we have a trace that looks like:

\[ (a, b \circ c, a \circ b \circ a), \]

then a resulting view from this trace could be:

\[ \langle \langle a \rangle, \langle c, b \rangle, \langle b, a, a \rangle \rangle. \]

We want to adhere to the same pattern we've been following so far, meaning that we need to define a function to generate the views of a given trace.

Let's attempt a definition:

\[ \text{Views}(\langle a_0, a_1, \ldots, a_n \rangle) = \{ \langle b_0, b_1, \ldots, b_n \rangle \mid b_i \in Ropes(a_i) \forall i \in [0, n] \}. \]

This definition should not be too surprising. We merely wish to create intermediate traces which are formed using a ROPE of each event in the given trace.
5.5 Can Intermediate Trace Land Help us Break the Linda Ambiguity?

The next chapter is dedicated to applying the tools we have defined here towards our goal of disambiguating the reasoning of inp and rdp. With the in-depth attention we have given to the different properties of the ◆ operator, there should be little doubt as to the integrity of intertwine. By building off of the problems we inspected at the end of the third chapter, we will continue to attack the Linda ambiguity. This time, our method of reasoning should be powerful enough for us to succeed.
Chapter 6

Back to the Ambiguity

So far, we have developed many tools with which to analyze the Linda predicate ambiguity. Although it might be the case that our tools are far more powerful than what we would need in this investigation, we hope that their potential can be realized in other cases. Still, we will attempt to make use of our new notation as much as possible as we revisit the Linda predicate operations.

6.1 The New Interleaving

Before we finish, however, we need to reevaluate a CSP operation in terms of our new extension. Namely, the \( \parallel \) operator needs to be redefined so that it supports the option of parallel events occurring simultaneously, as Lawrence already has [5]. Thus, where the original definition works as:

\[
(a \to P) \parallel (b \to Q) = (a \to (P \parallel (b \to Q))) \Chair (b \to (a \to P \parallel Q)),
\]

we need our definition to be slightly modified. One option would be to use a different interleaving, which is defined in the same place that provided
the notation for our parallel events [5]. This interleaving, however, forces the events to occur in parallel if such a choice is possible. We want to describe an interleaving that contains no such bias, but still describes the parallel events that can occur. Thus, we will extend the CSP symbol \( \parallel \), so that it will be used in the context of parallel events in the following way:

\[
(a \rightarrow P) \parallel (b \rightarrow Q) = (a \rightarrow (P \parallel (b \rightarrow Q)) \square (b \rightarrow (a \rightarrow P \parallel Q)) \square (a \circ b \rightarrow (P \parallel Q)).
\]

6.2 The Ambiguous Example

Looking back at the trace we had constructed to generate the ambiguity (the reader may wish to review the material covered in chapter three), the first three terms were \( pTS.(\text{"ready"}), TSp.(\text{"ready"}), pTS.(\text{"ready"}) \) and are required to demonstrate our ambiguous case. The remaining two events, \( TSp.(t) \), and \( TSa.0.(t') \) could occur in any order, including together. We can model these possibilities in intermediate trace land using the intertwine operator.

For the purpose of this investigation we will assume perfect observation, since imperfect observation is not required to demonstrate the ambiguity. Thus, we do not have to worry about the use of \( \text{pieces}() \), and we can very simply evaluate operations using intertwine.

6.2.1 Using Intermediate-Trace Land

Thus we will first rewrite our trace as it appears in Intermediate-Trace Land, \( \ll pTS.(\text{"ready"}), TSp.(\text{"ready"}), pTS.(\text{"ready"}) \rr \), but continue it with the permutations of the remaining two events, which we can describe as: \( \ll TSp.(t) \rr \circ \ll TSa.0.(t') \rr \). Our result is sim-
ply a longer intermediate trace, actually representing a set of intermediate traces, which appears to need some sort of flattening: \((pTS.("ready"), TSp.("ready"), pTS.("ready"), \langle TSa.(t) \rangle)\). 

Because of the length of our inner intermediate traces, intertwining them is not difficult. So, very simply, we get the following set of intermediate traces:

\[\{(TSa.(t'), TSp.(t)), (TSa.(t') \circ TSp.(t)), (TSp.(t), TSa.(t'))\}\].

6.2.2 Three Resulting Traces

Thus, as was probably obvious before, the three traces that we can get from this (after a flattening) look like:

\[\langle pTS.("ready"), TSp.("ready"), TSa.(t'), TSp.(t) \rangle,\]
\[\langle pTS.("ready"), TSp.("ready"), pTS.("ready"), TSa.(t') \circ TSp.(t) \rangle,\]
\[and \langle pTS.("ready"), TSp.("ready"), pTS.("ready"), TSp.(t), TSa.(t') \rangle.\]

These three traces correspond to the three different meanings that a failure signal could represent. Either the failure will be communicated before, while or after a matching tuple becomes available. In the cases above, we can see a simple manifestation of these possibilities. Although we have not covered every single possibility, any instance of a failed Linda predicate will have a trace that falls into one of the three categories represented here.

6.3 Parallel Events Make it Possible

Previously, without the help of modelling parallel events, we were unable to correctly distinguish the three different categories. Sequentialized traces only suggest that two different possibilities, the before and after cases, can occur. There simply is no direct way to express the case of simultaneity.
With this developed notation of traces, however, it is clear when the failure is communicated relative to the time the match becomes available. Assuming perfect observation, as we have been doing, there is no chance that a trace will be given which does not specify whether the two events happened simultaneously.

We have shown now that when given an intermediate trace describing the events leading up to an *inpor rdp* failure, we can see the order of events that led to that failure. The actual meaning of any failure can be unambiguously discerned. Thus, because of this extension of CSP we have built, we are able to clarify the predicate ambiguity.
Chapter 7

Results and the Future

The ambiguity we have studied through the previous six chapters has finally given way when viewed with our new notation. However, this is not the only positive result of our story which has wound through intermediate trace land, CSP, and Tuple Space. There are also a variety of tools and techniques presented here, which should be useful to other future research in the area of concurrent computing.

7.1 Success with the Ambiguity

Fighting against this ambiguity, however, has been the pressing concern throughout this explanation, and this it is from the path to this goal that all our work has been produced. Previously, the ambiguity existed because any means to reason about the results of a failure using sequentialized traces could not reveal the exact order of occurrence of the given events. Through the work completed here, as soon as events could be recorded simultaneously in event traces, and methods to reason about this were given, the perceived
ambiguity gave way.

Although this goal has been tackled before by Smith [3] using an operational semantics model of Linda and Tuple Space. He was able to derive this same result using the concept of parallel events.

In using a CSP-based model, it seems clear that it is our extension that has removed the ambiguity. It remains to be seen, however, whether other ambiguities aside from this one could be solved with this extension. Perhaps, if the ambiguities arise from problems with sequentialized traces, then we could use intermediate traces to resolve these problems. If similar problems exist, then this method should be applied to them.

7.2 The CSP Model of Tuple Space

The by-products of this exploration should also be noted. The first of these is the model of Tuple Space which was created in the CSP process algebra. Not only does this provide a means to reason about Tuple Space with CSP, but it also represents a specific implementation of the Linda primitives and the structure of Tuple Space. This implementation, pieced together from the semantic definitions of Tuple Space given by Gelernter [1], is not the only means of defining the system. It could, however, be a guide for future implementations, in the case that our method for creating Tuple Space closely matches specifications that might be desired.

7.3 Utilizing Imperfect Observation

Another by-product of our work is the development of imperfect observation. Although many definitions were designed in the effort to formalize
this concept, the actual final cases in resolving the ambiguity do not use this important facet of reasoning about processes. In fact, it is somewhat disappointing that such studies were not made on the cases investigated here. However, this work can be done without extensive effort, since a great deal of the theory has already been formulated. Such inspections should certainly be made, and probably would already have been deduced if more time had been permissible here.

7.4 Intermediate-Trace Land

In addition, although a good deal of the work has been structured towards defining intermediate traces and the intertwine operator, many closure properties still remain to be finalized. There are a few properties which are simply stated, but should be proved if they are to be fully believed. At the same time, studying these properties should also inspire a further exploration of the uses of these new tools. Even in the realm of View-Centric Reasoning [3], which we began to model, more interrelations must exist. Hopefully, links between this theory and other areas can be made that suggest some further importance for intermediate trace land.

7.5 Expansions on the Merge Operator

The greatest contribution that has come from here is the work on extending Lawrence's event-merging [5]. Here, too, we cannot imagine that all properties have been explored for the merge operator. It is our intent that the work presented will support and further the CSP-based notation which Lawrence has begun. Certainly the extensions to intermediate traces and view-centric
reasoning are a useful application and extension to the theory described as $HCSP$. Interestingly enough, it should be explained that although Lawrence appears to be interested in describing hardware-functions, his system was strong enough to have been used in the high-level realm of Tuple Space.

7.6 Conclusion

In closing, we will make no claims that we have completely resolved all the problems inherent with the Linda predicate operations. We have shown a solid method for reasoning about them which overcomes any ambiguity within the predicates, however, which was our goal from the beginning. Also, we have hopefully explained the use of expressing simultaneous events in parallel, and produced a solid theory with which to utilize this model. Using this model, which does not produce limiting, sequentialized traces, we were able to mimic one solution to the Linda predicate ambiguity problem, using an extension of the algebraic process notation, CSP. This solution and the accompanying theory concretely display the necessary elements of our extension, and the description presented in this work finely details the steps taken to generate these results.
Bibliography


