Model Transformation at Runtime for Dynamic Adaptation in Distributed Groupware

by

Christopher Wolfe

A thesis submitted to the School of Computing in conformity with the requirements for the degree of Doctor of Philosophy

Queen’s University
Kingston, Ontario, Canada
February, 2011

Copyright © Christopher Wolfe, 2011
Abstract

Adaptive distributed systems have the potential to revolutionize how humans and computers interact. They can enable software to adapt to dynamic human demands, as users change their focuses, goals, locations and devices. However, producing these systems is currently very challenging: developers must solve daunting user interface issues while mired in distributed systems problems.

We present a new class of toolkit, intended to ease the development of adaptive distributed systems. Unlike existing alternatives, we provide a high-level programming model in which developers can easily specify runtime adaptations. Meanwhile, our toolkit automatically generates a fast and tunable implementation. Partial failures in the distributed system are reflected back into the high-level programming model. As a result, developers can remain insulated in their high-level model while building highly-dynamic, high-performance and failure-resistant applications.

Our Fiia.Net toolkit relies on model transformation at runtime to bridge between the programmer’s high-level model and the actual implementation of the distributed system. Our novel model transformation is the first that can practically maintain this transformation, and enables our toolkit to easily support user-driven adaptations, dynamic optimization and self-healing.
Acknowledgments

My greatest thanks go to my doctoral supervisor, Nick Graham, for his tireless support throughout my work. Nick gave me the freedom to pursue my own ideas, while always being there when I needed advice or assistance. He has helped reshape my innate curiosity and reflexive tinkering into valuable research tools.

The question that inspired this thesis grew out of Greg Phillips’ Ph.D work on Fia – then called the Workspace Model. Discussions with Greg, as well as his writings and Python prototype, leave my work standing solidly upon his shoulders.

Many improvements to my work were driven by the experiences of other members of the EQUIS lab, including Banani Roy, David Smith, Eric Qiu and Rob Fletcher. From outside the lab, Christopher Lansing volunteered to help with my performance evaluation. Their feedback, patience with my early prototypes and help in testing is endlessly appreciated.

My examining committee of Philippe Palanque, Tom Dean, Ahmed Hassan and Mohammad Zulkernine provided many perceptive questions and comments. With their help, the final version of this dissertation is much improved.

The final enthusiasm I needed to complete this thesis came about with the advice, and references, of Jim Cordy and Carl Gutwin. Throughout my work I have had the
benefit of encouragement from other members of the School of Computing, and the valuable support of my family.

I began my graduate studies with the goal of producing a quick thesis, and then diving into industry. Along the way that goal was subsumed by sheer enjoyment of research, and the conversations and relationships that it breeds. My only regrets today are for projects not undertaken, papers not written, and collaborations not pursued. Yet.
Statement of Originality

I, Christopher Wolfe, certify that all results in this thesis are original, unless otherwise noted. Specifically, those results due to other authors which have appeared in the literature have been cited as necessary. Earlier versions of some parts of the work reported in this thesis have previously appeared as [136][137][138].

Several researchers have made notable contributions to this work:

- W. Greg Phillips developed the Fiia approach, meta-models and refinement rules [104]. As part of that work, he produced a proof-of-concept Fiia toolkit in Python. That toolkit struggled with the weaknesses of existing model transformation approaches, so was much more limited than Fiia.Net.
- Banani Roy designed and implemented the furniture layout demo described in Chapter 2, and contributed to the development of the related scenario [137].
- J. David Smith designed and implemented the Raptor game prototyping tool [118][119] used for part of our evaluation in Chapter 9.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>i</td>
</tr>
<tr>
<td>Acknowledgments</td>
<td>ii</td>
</tr>
<tr>
<td>Statement of Originality</td>
<td>iv</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>ix</td>
</tr>
<tr>
<td>Chapter 1. Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Motivation</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Problem</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Problem Statement</td>
<td>10</td>
</tr>
<tr>
<td>1.4 Contributions</td>
<td>12</td>
</tr>
<tr>
<td>1.5 Organization of the Thesis</td>
<td>14</td>
</tr>
<tr>
<td>Chapter 2. Context: Adaptive Groupware with Fiia</td>
<td>16</td>
</tr>
<tr>
<td>2.1 A Furniture Layout Application</td>
<td>18</td>
</tr>
<tr>
<td>2.2 Scene 1: Clive explores his future office space</td>
<td>20</td>
</tr>
<tr>
<td>2.3 Scene 2: Adding the quotation</td>
<td>23</td>
</tr>
<tr>
<td>2.4 Scene 3: Clive leaves to discuss with Bill</td>
<td>24</td>
</tr>
<tr>
<td>2.5 Implementing the Scenes</td>
<td>25</td>
</tr>
<tr>
<td>2.6 Refining the Models</td>
<td>27</td>
</tr>
<tr>
<td>2.7 Summary</td>
<td>28</td>
</tr>
<tr>
<td>Chapter 3. Distributed Systems and Groupware</td>
<td>30</td>
</tr>
<tr>
<td>3.1 Programming Models</td>
<td>30</td>
</tr>
<tr>
<td>3.2 Adaptation</td>
<td>36</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
</tr>
<tr>
<td>------------------</td>
<td>------</td>
</tr>
<tr>
<td>A.2 Match Rule</td>
<td>192</td>
</tr>
<tr>
<td>A.3 Apply Rule</td>
<td>193</td>
</tr>
<tr>
<td>A.4 Transform</td>
<td>194</td>
</tr>
<tr>
<td>A.5 Conceptual Update</td>
<td>195</td>
</tr>
<tr>
<td>A.6 Distribution Update</td>
<td>197</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Conceptual and distribution levels of a distributed systems toolkit</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>One refinement in a tree of possibilities</td>
<td>9</td>
</tr>
<tr>
<td>1.3</td>
<td>Commuting of refinements over adaptations</td>
<td>9</td>
</tr>
<tr>
<td>2.1</td>
<td>Scenes from the example scenario</td>
<td>18</td>
</tr>
<tr>
<td>2.2</td>
<td>Overview of Fiia conceptual notation</td>
<td>19</td>
</tr>
<tr>
<td>2.3</td>
<td>Scene 1: Clive and Sally collaboration on the furniture layout</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Scene 2: Clive views a quotation for his new furniture</td>
<td>23</td>
</tr>
<tr>
<td>2.5</td>
<td>Scene 3: Clive goes to discuss the quotation with Bill</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Overview of the Fiia distribution notation</td>
<td>25</td>
</tr>
<tr>
<td>2.7</td>
<td>Two possible refinements of Scene 1</td>
<td>25</td>
</tr>
<tr>
<td>2.8</td>
<td>Fiia local call rule</td>
<td>27</td>
</tr>
<tr>
<td>2.9</td>
<td>Fiia remote call rule</td>
<td>27</td>
</tr>
<tr>
<td>3.1</td>
<td>GroupKit [110]</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>TeamRooms [111]</td>
<td>33</td>
</tr>
<tr>
<td>3.3</td>
<td>Some structural variants from Patterson’s taxonomy [101]</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Two major families of graph transformation</td>
<td>49</td>
</tr>
<tr>
<td>4.2</td>
<td>Example conceptual model with calls passing between two components</td>
<td>50</td>
</tr>
<tr>
<td>4.3</td>
<td>Example rules for refining components and call connectors</td>
<td>50</td>
</tr>
<tr>
<td>4.4</td>
<td>Example of a rewriting transformation</td>
<td>51</td>
</tr>
<tr>
<td>4.5</td>
<td>Example of a relational transformation</td>
<td>51</td>
</tr>
<tr>
<td>4.6</td>
<td>Example conceptual model with events passing between two components</td>
<td>52</td>
</tr>
<tr>
<td>4.7</td>
<td>Example rule for refining subscription connectors</td>
<td>52</td>
</tr>
<tr>
<td>4.8</td>
<td>Example of a rewriting transformation</td>
<td>52</td>
</tr>
<tr>
<td>4.9</td>
<td>Incorrect relational transformation with intermediate elements</td>
<td>53</td>
</tr>
<tr>
<td>4.10</td>
<td>Corrected subscription rule for relational transformation</td>
<td>53</td>
</tr>
<tr>
<td>4.11</td>
<td>Three forms of update in model-to-model transformations</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.12.</td>
<td>Example of a conceptual update</td>
<td>56</td>
</tr>
<tr>
<td>Figure 4.13.</td>
<td>Techniques for updated in forward-only transformations</td>
<td>57</td>
</tr>
<tr>
<td>Figure 4.14.</td>
<td>Conceptual update using a Triple Graph Grammar</td>
<td>60</td>
</tr>
<tr>
<td>Figure 4.15.</td>
<td>How existing model transformation techniques satisfy the requirements of model transformation at runtime</td>
<td>63</td>
</tr>
<tr>
<td>Figure 5.1.</td>
<td>Graph for part of scenario conceptual graph</td>
<td>68</td>
</tr>
<tr>
<td>Figure 5.2.</td>
<td>Changing the example graph</td>
<td>69</td>
</tr>
<tr>
<td>Figure 5.3.</td>
<td>Example graph pattern</td>
<td>71</td>
</tr>
<tr>
<td>Figure 5.4.</td>
<td>Example graph</td>
<td>71</td>
</tr>
<tr>
<td>Figure 5.5.</td>
<td>A graph transformation rule in visual and textual notations</td>
<td>74</td>
</tr>
<tr>
<td>Figure 5.6.</td>
<td>Attempt to match a rule in a graph</td>
<td>75</td>
</tr>
<tr>
<td>Figure 5.7.</td>
<td>Apply a rule to a graph</td>
<td>76</td>
</tr>
<tr>
<td>Figure 5.8.</td>
<td>An example step and the resulting graph</td>
<td>77</td>
</tr>
<tr>
<td>Figure 5.9.</td>
<td>Transform a graph</td>
<td>80</td>
</tr>
<tr>
<td>Figure 5.10.</td>
<td>Apply an existing rule to a graph</td>
<td>81</td>
</tr>
<tr>
<td>Figure 5.11.</td>
<td>Trace and dependency graph of a simple transformation</td>
<td>82</td>
</tr>
<tr>
<td>Figure 6.1.</td>
<td>Graph relationships in refinement</td>
<td>86</td>
</tr>
<tr>
<td>Figure 6.2.</td>
<td>Commuting over an addition to the conceptual graph</td>
<td>88</td>
</tr>
<tr>
<td>Figure 6.3.</td>
<td>Example of an addition to the conceptual graph</td>
<td>89</td>
</tr>
<tr>
<td>Figure 6.4.</td>
<td>Update the refinement after an addition to the conceptual graph</td>
<td>90</td>
</tr>
<tr>
<td>Figure 6.5.</td>
<td>Commuting over a removal from the conceptual graph</td>
<td>91</td>
</tr>
<tr>
<td>Figure 6.6.</td>
<td>Example of a removal from the conceptual graph</td>
<td>92</td>
</tr>
<tr>
<td>Figure 6.7.</td>
<td>Update the refinement after a removal from the conceptual graph</td>
<td>94</td>
</tr>
<tr>
<td>Figure 6.8.</td>
<td>Update the refinement after a change to the conceptual graph</td>
<td>96</td>
</tr>
<tr>
<td>Figure 6.9.</td>
<td>Commuting over a removal from the distribution graph</td>
<td>98</td>
</tr>
<tr>
<td>Figure 6.10.</td>
<td>Example of a removal from the distribution graph</td>
<td>99</td>
</tr>
<tr>
<td>Figure 6.11.</td>
<td>Update a refinement after a removal from the distribution graph</td>
<td>101</td>
</tr>
<tr>
<td>Figure 7.1.</td>
<td>Update a step as part of resolving a change in the conceptual graph</td>
<td>106</td>
</tr>
<tr>
<td>Figure 7.2.</td>
<td>Update the refinement following a change in the conceptual graph</td>
<td>107</td>
</tr>
<tr>
<td>Figure 7.3.</td>
<td>Update a step as part of resolving a removal in the distribution graph</td>
<td>108</td>
</tr>
<tr>
<td>Figure 7.4.</td>
<td>Update the refinement following a removal in the distribution graph</td>
<td>109</td>
</tr>
<tr>
<td>Figure 7.5.</td>
<td>A graph transformation rule with a forbidden precondition</td>
<td>111</td>
</tr>
<tr>
<td>Figure 7.6.</td>
<td>Attempt to match a rule with forbidden preconditions in a graph</td>
<td>112</td>
</tr>
<tr>
<td>Figure 7.7.</td>
<td>Update the refinement after a change to the conceptual graph</td>
<td>113</td>
</tr>
<tr>
<td>Figure 7.8.</td>
<td>A graph transformation rule with repeated sections</td>
<td>116</td>
</tr>
<tr>
<td>Figure 7.9.</td>
<td>Apply a rule with repeated sections to a graph</td>
<td>117</td>
</tr>
<tr>
<td>Figure 7.10.</td>
<td>Apply a trace with repeated sections to a graph</td>
<td>118</td>
</tr>
<tr>
<td>Figure 7.11.</td>
<td>Update a step as part of resolving a change in the conceptual graph</td>
<td>120</td>
</tr>
</tbody>
</table>
Figure 7.12. Update a step as part of resolving a removal in the distribution graph. ................................. 121
Figure 7.13. Update a step as part of resolving a removal in the distribution graph. ................................. 123
Figure 8.1. Runtime architecture of the Fiia.Net toolkit. .................................................. 127
Figure 8.2. Data flow in Fiia.Net during adaptation ............................................................. 128
Figure 8.3. Source of the furniture layout Quotation class .................................................. 130
Figure 8.4. Quotation and viewer created in a single process ............................................. 131
Figure 8.5. Quotation and viewer created in a Fiia.Net application ................................. 131
Figure 8.6. Definition of the Anchor refinement rule ...................................................... 134
Figure 8.7. Definition of the Instantiate refinement rule .................................................. 135
Figure 9.1. The Raptor game prototyping tool in action [119] ............................................. 150
Figure 9.2. Conceptual model of the Raptor game prototyping tool ................................ 150
Figure 9.3. Time required to apply small conceptual additions ........................................ 151
Figure 9.4. Time required to apply small conceptual and distribution deletions .................. 152
Figure 9.5. Case study Fiia.Net runtime performance ..................................................... 156
Chapter 1

Introduction

In this thesis we present a new technique for implementing adaptive distributed systems. Our approach spares developers the traditional compromise between a high-level, abstract programming model, and precise control of runtime changes. As a result, developers can work exclusively within the high-level view, even while building systems that support changing user demands, varied computing devices, dynamic mobile environments and partial failures.

This unique combination of programming abstraction and runtime adaptation is provided by a general-purpose model transformation engine. Formal model transformation rules specify how the transformation can refine a high-level programming model to many possible implementations. One of the possible implementations is chosen automatically, or developers can step in to control aspects of the implementation. Unlike typical model-driven engineering systems, we maintain the models and transformation at runtime. This provides a dynamic bridge between the programmer’s view of the system and the instantaneous distributed implementation. The resulting
CHAPTER 1. INTRODUCTION

toolkit supports complex and arbitrary changes at runtime – adaptation – without abandoning the developer to low-level programming.

The core of our contribution is a model transformation algorithm that efficiently maintains this model transformation at runtime. We go beyond existing techniques for incremental update to support the complex implementation decisions required by adaptive distributed systems. Our novel algorithm is the first that makes this general transformation practical for controlling live distributed systems at runtime.

Our work is demonstrated through multi-user distributed systems, often called groupware. Common examples include communication applications like Skype and multi-player games like World of Warcraft. These systems can adapt to some changes at runtime: Skype allows users to join and leave a conversation, and World of Warcraft switches users between physical servers when they move to different areas of the virtual world. These simple adaptations are laboriously hand-coded, requiring expert knowledge of how the distributed system is implemented.

The modern world of mobile users on widely varying devices, from PCs to smartphones, motivates groupware permitting far more flexible adaptations. Such adaptive groupware would allow users to easily change what they are doing, how they are interacting, and what devices each is using. For example, a researcher working on a conference paper could use a collaborative text editor on an office PC, sketch diagrams on a smart whiteboard during a meeting, and then mark up revisions using a smartphone on the train home. However, existing approaches to developing groupware limit its ability to adapt to these scenarios, or abandon developers to extensive low-level programming.
CHAPTER 1. INTRODUCTION

We apply the existing Fiia meta-models and transformation rules to bridge between a high-level programming model and an adaptive distributed system. The resulting toolkit has been used to develop several interesting examples of adaptive groupware, including a game prototyping tool and a multi-platform furniture layout demo. These experiences suggest that model transformation at runtime can offer high-level expressiveness without sacrificing precise adaptation and good performance. This combination is crucial to easing the development of adaptive groupware, and other adaptive distributed systems.

In the remainder of this chapter, we present an introduction to adaptive distributed systems, and show how using model transformation at runtime can simplify their development. From this, we specify the problems addressed by this thesis and outline our contributions.

1.1 Motivation

Recent history has seen exciting evolution in the design of multi-user software systems – from video-conferencing, to collaborative editing, social networks, and games. One emerging trend is toward systems that remain useful as users move between different interactions, different locations, and different devices. Far from traditional desktop-centric applications, a user today might begin writing an e-mail on his work PC, continue it on his phone while riding the train, and finally send it using a netbook from his patio. This sort of adaptation is key to bringing software into a world of increasingly mobile users, unplanned interactions, and massively heterogeneous devices. Many researchers have built adaptive groupware systems focusing on particular applications; for example:
Mohoc [106] lets health-care workers coordinate activities while their locations and levels of network connectivity vary.

Archeo [1] helps organize archaeological field work, by letting users move between their office PC and a PDA at the dig site.

Software Design Board [140] allows software developers to move between synchronous and asynchronous collaboration, colocated and distributed settings, and PCs and electronic whiteboards.

Developers of such systems face challenging technical problems. All three examples provide different views to users who are interacting with data in different ways, and must keep the information consistent. For example a Software Design Board user can keep one region of a UML diagram private, share another region for live collaborative editing, and detach a third region to edit locally and merge later. Mohoc and Archeo use wireless communication from mobile devices, so are further designed to tolerate network failures and respect power consumption. This new flexibility integrates software into the users’ existing workflows and environments, but makes it much harder to develop both the user interface and underlying distributed system. In each of these examples, a significant part of the research contribution is resolving distributed systems problems to support a productive user experience.

This interplay between application-specific design and distributed systems problems is not unique to adaptive groupware. In 1989, Bal et al. surveyed nearly one hundred programming languages for distributed systems [9]. These languages expose many different abstractions of parallelism, communication and failure handling, and use many different distributed implementations. Bal et al. argue that the choice of
CHAPTER 1. INTRODUCTION

tool should be dictated by how well its models support the application, and how well its implementation will perform on available hardware. This has held true through the modern explosion of toolkits for programming groupware [135].

The key idea of these approaches is that the developer’s view of the system is abstracted and simplified from the complexity of the implementation. This leads to the separation shown in Figure 1.1: the developer works in a conceptual programming model, while the distributed system is actually implemented as concrete objects and network links. The distributed systems toolkit – tool, language, and/or library – maps from the developer’s instructions to the implementation. Because developers are insulated from this implementation, many different techniques – from caching and replication, to multicast and compression – can be used without complicating application development.

Applying a similar separation to adaptive distributed systems requires an important addition: expression of runtime change. Traditional groupware toolkits support limited runtime change: typically in the form of users joining and leaving collaborations, and occasionally for handling partial failure in the form of network
CHAPTER 1. INTRODUCTION

disconnections and node crashes. Dynamic software architectures \cite{18, 19} suggest a way to express this adaptation, by programming changes in the adaptive groupware system. Existing approaches to this problem are reviewed in Chapter 3.

An effect of this runtime change is that choosing a distributed implementation is very difficult. The exact configuration of the system will depend on the users, what they are doing, their hardware, and the network environment. This variability makes it hard to predict how well different implementations will fare. Indeed, the costs of both modeling and manually implementing alternatives explode with the complexity of the system \cite{61}. Insulating application code from implementation decisions lowers the cost of such experiments, because the actual distribution can be changed without rewriting the application. Ideally the distribution would even be flexible at runtime, to account for unforeseen conditions (e.g. \cite{77}).

These issues suggest five requirements for an adaptive groupware toolkit:

**Programming Model:** Application developers should be able to work within a high-level programming model appropriate for groupware. This will insulate them from details of the distributed implementation, like consistency maintenance algorithms, threading, and network connections.

**Adaptation:** Application designers and developers should plan and express runtime changes in the high-level programming model. This will allow them to control the distributed system, while still abstracting details of its implementation and exactly how the reconfiguration is performed. For example, simple high-level changes might require the implementation to establish new network connections and migrate data between physical devices.
CHAPTER 1. INTRODUCTION

**Failure Handling:** Distributed applications occasionally suffer from partial failures, such as a network link dropping or a node becoming unavailable. Application developers should be able to monitor and respond to such partial failure without having to leave their high-level programming model. This will help them understand the user-visible effects of failure and program application-specific recovery, without being exposed to the underlying distributed system.

**Performance:** The runtime performance of applications developed using the toolkit should be similar to traditional tools that do not encourage adaptation. This avoids the common criticism that higher levels of abstraction introduce unacceptable performance penalties.

**Optimization:** Developers should be able to build distributed applications without concern for how they are implemented, and optimize them later. The toolkit should automatically choose an implementation that it expects to perform well, and may adjust it at runtime. Developers should be able to override these choices, which will let them easily evaluate different implementations.

In Chapter 3 we review existing tools for developing groupware and distributed systems, and examine how their capabilities compare to these requirements.

### 1.2 Problem

These requirements for an adaptive groupware toolkit position it between the application developer’s high-level programming model and the actual distributed system. Implementing this bridge is far from trivial: it must cross a wide semantic gap, support runtime change from both ends and let developers override the intermediate
CHAPTER 1. INTRODUCTION

decisions. Because of this complexity, an ad-hoc toolkit is infeasible, and we turn to more formal techniques.

Model-based techniques have shown great potential in distributed systems. High-level conceptual models \cite{59, 21, 72, 82} can describe a system’s structure, while abstracting low-level issues like data sharing and caching policies, concurrency control algorithms, and network protocols. Meanwhile, distribution models can help developers reason about architectural trade-offs \cite{62} or configure an implementation \cite{92}

These two levels of model provide a framework for adaptive groupware toolkits. We propose a toolkit maintaining two models at runtime: one containing a snapshot of the system’s abstract state, and one with a snapshot of the actual implementation. The relationship between the models is defined by a refinement model transformation, using formal rules to express how a conceptual model can be implemented. Runtime changes in the models are expressed as precise edits, capturing the various forms of adaptation required in the toolkit.

Of the two models, the high-level conceptual model is exposed to applications via reflection, events, and an editing API. Applications call into the API to specify runtime adaptations, and use events and reflection to gauge the effects of partial failure.

The low-level distribution model describes the underlying distributed system. It is updated automatically by the model transformation engine following conceptual adaptations. Partial failures are detected by components with the distributed system. We limit ourselves to failures easily represented by deletions from the distribution model, such as dropped network connections and crashed nodes. The model transformation engine automatically propagates those deletions up through
the refinement to the conceptual model and the developer’s API. This ensures that, in spite of partial failures, the developer’s high-level programming model continues to reflect the true configuration of the system.

Figure 1.2 defines the precise relationship between the conceptual and distribution models. A conceptual model $C_1$ corresponds to a distribution model $D_1$ through a refinement consisting of a sequence of transformation steps (such as $s_1$, $s_2$). Many different sequences of steps are possible, often leading to different distribution models (shown as unlabeled steps and models). Figure 1.3 concatenates the steps into a single transformation $T_1$. The transformation steps are automatically generated and maintained by a model transformation engine.

At runtime, either model may be modified, as shown in Figure 1.3. When the application needs to adapt, such as letting the user move between devices, it generates a modification $\Delta_C$ to $C_1$. After applying this modification, the resulting conceptual model $C_2$ may not correspond with $D_1$. The model transformation engine must find a new transformation $T_2$ and distribution model $D_2$ which are compatible with $C_2$. 
CHAPTER 1. INTRODUCTION

A similar process occurs when the distributed system detects a failure: it applies a modification $\Delta_D$ to $D_1$, and the engine must find $T_2$ and a new conceptual model $C_2$.

This approach of using runtime models and transformation can provide the properties required by distributed systems such as adaptive groupware. However, it poses challenges that are not answered by existing model transformation engines. These engines typically operate at design- or development-time to refine a generic architecture into a static implementation. This offers interesting abstractions and possibilities for optimization. However, because they do not maintain the models and transformation at runtime, they offer little support for adaptation and failure handling.

1.3 Problem Statement

Existing toolkits for distributed systems force developers to choose between a high-level programming model and runtime adaptation. This is particularly troublesome in adaptive groupware, where toolkits should support easy user-driven changes, failure handling and online optimization.

Using model transformation at runtime to implement an adaptive groupware toolkit poses significant challenges. From the earlier toolkit requirements and related work, we identify five requirements for the model transformation engine:

**Expressive Transformation Rules:** The model transformation must be able to express the complex decisions typical in implementing groupware systems. This supports strong abstraction, and lets developers optimize the system by overriding individual decisions.
CHAPTER 1. INTRODUCTION

Conceptual Adaptations: The transformation engine must support arbitrary changes in the conceptual model, and propagate them through to the distribution model. This allows developers to control the distributed system from their high-level view, while the transformation engine automatically maintains the implementation.

Distribution Adaptations: The transformation engine must support changes in the distribution model to indicate partial failure. The transformation will restore both conceptual and distribution models to a consistent state. This allows developers to observe and respond to failures entirely within their high-level view.

Fast Updates: The transformation engine must apply adaptations quickly, so that they can perform user-requested changes without jarring pauses. Scaling the distributed system toward hundreds of users and frequent adaptations will place a significant burden on the engine.

Stable Updates: Applying an adaptation should have as little impact on the models as possible. This means that updates should be stable; in other words, that the transformation engine will only modify parts of the model directly impacted by the adaptation.

Because the distribution model is coupled to a live distributed system, large changes in it can easily cause noticeable delays as data is copied or network connections reconfigured. Likewise, the conceptual model contains the application developer’s view of the system. The impact of partial failures should be
CHAPTER 1. INTRODUCTION

local so that application developers can diagnose problems and provide user-friendly errors and recovery.

In [Chapter 4] we review existing tools and techniques for model transformation, and examine how their capabilities compare to these requirements.

1.4 Contributions

The core contribution of this thesis is an algorithm that performs model transformation at runtime for adaptive distributed systems. The algorithm and its supporting theory extend the state of the art, and our prototype implementation provides valuable experience. In particular:

**Novel algorithm:** We specify and demonstrate a novel algorithm for maintaining a model transformation during live changes. This algorithm is the first which efficiently supports the expressive rules, adaptations, and fast, stable update motivated by adaptive groupware toolkits.

**Graph rewriting theory:** We present a formal definition of graph rewriting that includes the contents of the trace. This theory is used to explain our novel algorithm and demonstrate its correctness. Because this theory builds from set theory, it maps easily to common data structures and efficient implementations.

**Prototype toolkit:** We have constructed the *Fiia.Net* toolkit to serve as a demonstration of this work. It has been used as the basis for several interesting examples of adaptive groupware (e.g. [51, 118, 137]). Our informal experiences with the toolkit provide valuable guidance in this emerging field of software engineering.
CHAPTER 1. INTRODUCTION

**Evaluation:** We evaluate the performance and distribution stability of the Fiia.Net transformation engine, with and without the novel algorithms described in this thesis. Because our algorithms build from common graph rewriting concepts, they should provide similar benefits to pre-existing engines.

We do not attempt to replicate, or replace, the decades of previous research in groupware and distributed systems. Our objective is, instead, to offer a bridge between conceptual-level and distribution-level development.

We envision the design of adaptive groupware being performed using an abstract graphical notation [105]. This notation would describe usage scenarios as multiple scenes, each an instantaneous snapshot of the application’s conceptual structure. The scenes and the runtime transitions between them would then be programmed in an API based on the conceptual model. Model transformation at runtime, and our trace-based update algorithm, provide the means by which this conceptual model is coupled to the live distributed system.

High-level groupware features, like awareness mechanisms and session management would be built atop our toolkit. Here, they can take advantage of easy runtime adaptation, flexible implementation and high-level failure handling. DISCO [67] is an example of the kind of work that will benefit from this approach: it provides a framework to ease the user interface effects of temporary failures, so is based in a conceptual view of runtime adaptation.

Meanwhile, we also seek to reuse the extensive previous research focused on implementation techniques. These include architectures used in distributed systems [62], protocols for data consistency and concurrency, compression and network protocol, and the many approaches to low-level failure detection and tolerance. Components
implementing many of these techniques can simply be plugged into the distributed infrastructure underlying our toolkit. Others will contribute to the future design of better meta-models and refinement rules.

This thesis provides a model transformation system, which is the first capable of supporting model transformation at runtime for adaptive groupware and our Fiia.Net toolkit. We lay the foundation for what, we hope, is the exciting future of adaptive distributed systems.

1.5 Organization of the Thesis

We first explain the context of our work through a concrete example of adaptive groupware, in Chapter 2. This scenario serves to ground many of our requirements for an adaptive groupware toolkit, and is used as a running example through the rest of the thesis. We then review existing toolkits for distributed systems in Chapter 3 and examine which of our requirements they can satisfy. The limitations of these approaches show the need for a more powerful bridge from programming model to implementation, as could be provided by model transformation at runtime. In Chapter 4 we turn to the state of art in model transformations to implement this bridge. These transformation systems fall short of our requirements, but serve as a foundation for our contributions.

In Chapter 5 we outline the structure of our approach to adaptive distributed systems using model transformation at runtime, and present its formal underpinnings. We then present the core of our novel algorithm in Chapter 6 and show how it can support model transformation at runtime. Chapter 7 presents several extensions to this core algorithm. This extended algorithm is used to power our Fiia.Net toolkit,
CHAPTER 1. INTRODUCTION

which is itself described in Chapter 8. In Chapter 9 we collect our experiences and examine parts of our scenario to demonstrate our support for adaptive groupware. Finally, Chapter 10 reviews our contributions and outlines interesting topics for future research.
Chapter 2

Context: Adaptive Groupware

with Fiiia

In Chapter 1 we have outlined the promises and challenges of adaptive distributed systems. To ease the development of such systems, we propose an adaptive distributed systems toolkit using model transformation at runtime to bridge between a high-level programming model and the actual implementation of the system. The core of this thesis presents our novel model transformation system, with the capabilities demanded by this toolkit.

Earlier, we discussed three interesting examples of adaptive groupware from the literature: Archeo [1], Mohoc [106] and Software Design Board [140]. Each of these projects was focused on the user’s experience, but had to solve significant distributed systems issues to produce a usable implementation. Reaching our goal would allow future developers to ignore distributed systems issues, while providing even more flexibility to their users.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

In this chapter we present the context of our work, using a concrete scenario that combines many ideas from previous work in adaptive groupware. This scenario describes some of the complexities encountered by adaptive groupware developers, further motivating our toolkit and approach. In addition, it introduces the Fiia meta-models and rule-based refinement.

Fiia is a combination of a design notation and a precise model-driven implementation process, designed by Phillips and Graham [105] for adaptive groupware (originally published as the Workspace Model). It defines a conceptual meta-model, distribution meta-model and refinement rules for a model transformation between them. While Fiia proposes the use of model transformation at runtime, it leaves open the question of how such a transformation can be practically implemented. This thesis is the answer to that question.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Figure 2.1: Scenes from the example scenario

2.1 A Furniture Layout Application

Our example scenario is outlined in Figure 2.1. “Clive” is an office worker who is planning to purchase furniture for a new office space. He contacts “Sally”, a sales representative at a local furniture store, who helps him consider items and layout alternatives. Finally, Clive takes a formal quotation for the items to his supervisor “Bill” for purchase approval. The application used in the scenario has been implemented and demonstrated using our Fiia.Net tool [137]. This furniture layout scenario will be used in examples throughout this thesis and as the context for some parts of our evaluation.

Our scenario is described as three scenes. Each scene captures a portion of the entire scenario, analogous to scenes in a play. Actions by the users cause transitions between the scenes, which involve changes to the human interactions, the user interfaces and the underlying distributed system. Fiia proposes that designers use similar scenes and transitions to design adaptive groupware, described in its conceptual model. This model is a precise visual representation of the
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Figure 2.2: Overview of Fiia conceptual notation

developer’s programming model, and can serve as a design document analogous to UML component diagrams.

We describe each scene informally and present the corresponding Fiia conceptual model. These precise conceptual models help designers specify how the system should behave, and give developers a language in which to communicate with each other and our Fiia.Net toolkit. Figure 2.2 shows a summary of Fiia’s conceptual notation; the elements will be discussed with examples later in our scenario.

We will return to formalize many of these concepts in later chapters: Chapter 5 details how our models are represented and transformed, and Chapter 8 describes how an application is actually programmed in Fiia.Net.
2.2 Scene 1: Clive explores his future office space

In our first scene, Clive is preparing to move to a new office and is interested in choosing furnishings for it. He opens the floor plan in a first-person viewer on his PC. This application displays a 3D view of his future office space, with realistic renderings of furniture, walls, and lighting. Clive uses his keyboard and mouse to move through the virtual space.

Clive uses an option in the viewer to start collaborating with Sally, a sales representative at a local furniture store. Sally accesses the same floor plan using a dedicated furniture layout application, running on a large multi-touch surface. Her application shows a top-down view of Clive’s office space, and a palette of available furniture. Sally can drag items from the palette into the floor plan, and use gestures to orient them to fit. Clive sees these changes in real-time from his first-person view,
so can consider both the aesthetics and functionality of the layout. Both of these interfaces are shown in Figure 2.3a.

The conceptual model in Figure 2.3b describes the two separate settings (dashed boxes) – Sally’s Store and Clive’s Office – and the software components, physical devices, and people in each. The floor plan components are data stores, indicated by the \( \Theta \) symbol. Two or more stores may be synchronized by attaching them with a synchronization connector (\( \rightarrow \)). This indicates that the two stores should be kept observationally equivalent; that is, that queries to either should return the same information. The toolkit is free to implement this synchronization in many ways, including merging the components into a single centralized object. Other component types are either actors (\( \mathcal{O} \)), reactors (\( \mathcal{R} \)), or adapters (\( \mathcal{A} \)). Actors and reactors typically perform computations and provide user interfaces. Adapters, meanwhile, describe devices which bridge between the virtual and the physical world; for example, keyboards, mice and displays. Normal communication between components is over call (\( \rightarrow \)) or subscription (\( \Rightarrow \)) connectors, which, respectively, indicate blocking method calls and asynchronous event streams. Like synchronization connectors these may be implemented in many ways: as local pointers, network calls, queued packets or multicast messages. The conceptual model allows a designer or a developer to express the functionality of a system at a high level, without concern for how it will actually be implemented.

This scene illustrates two people sharing data, but using very different interfaces. This is important when, as here, users have varying goals, devices and levels of expertise:
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Users: Clive is a novice user of the floor planning software, mainly interested in getting desks to put computers on. Sally, meanwhile, has been trained to use the software effectively. She will try to make the resulting office safe and easy to navigate, as well as attractive.

Views: Clive is examining a small part of his floor plan in a detailed first-person rendering, while Sally is using a top-down overview that mixes 2D images and 3D models.

Devices: Sally is using a digital multi-touch table – a horizontal touch-sensitive display, with which she interacts using gestures – while Clive is using a traditional PC with mouse and keyboard.

Creating user interfaces for these varying configurations is challenging. The adaptive groupware toolkit must allow designers and developers to focus on that problem, and insulate them from the details of the underlying distributed system. At the same time, that insulation can not prevent developers from building adaptation into their applications.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Figure 2.4: Scene 2: Clive views a quotation for his new furniture on his PC, then migrates it to his PDA to avoid obscuring his first-person view.

2.3 Scene 2: Adding the quotation

Clive inquires about the price of the furniture. Sally uses a menu option to create a quotation which is shown on Clive’s display. As she modifies the furniture in the floor plan, the quotation updates automatically. Clive prefers not to have part of his first-person view obscured by the quotation, so moves the viewer from his PC display to his PDA.

These changes emphasize the importance of flexible adaptation and integrating varied components. The quotation contains information which was not available to Clive’s initial floor plan viewer, but now needs to be displayed in a convenient form. The adaptive groupware toolkit should let developers design and specify such changes within the high-level conceptual programming model.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Figure 2.5: Scene 3: Clive disconnects from the collaboration with Sally, and carries the quotation on his PDA to discuss with Bill.

2.4 Scene 3: Clive leaves to discuss with Bill

Sally and Clive reach agreement on the floor plan and quotation. Clive thanks her, and Sally signs off to help another customer. Clive carries his PDA to Bill’s office where they discuss final approval of the quotation.

This scene lasts only a few minutes, but demonstrates several types of user-driven adaptation: Clive changes location, from his office to Bill’s, carrying his PDA; Sally leaves the collaboration and Bill joins; and the originally distributed collaboration is replaced with two people looking at the same PDA.

Traditional applications provide no support for these adaptations, and adding them to existing distributed systems is a challenging prospect. We hope to ease the development of such adaptive distributed systems, by providing programmers with both a high-level programming model and extensive support for runtime adaptation.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

2.5 Implementing the Scenes

In addition to the conceptual meta-model, Fiiia provides a low-level distribution meta-model and rules for refining a conceptual model to a distribution model. These rules are non-deterministic; that is, the process of refining a conceptual model will reach states at which more than one rule can be used. The choice of rule will lead to different distribution models, and hence different implementations of the distributed system. These implementation possibilities often have very different performance characteristics, and effects on other quality attributes.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

For example, Figure 2.7 shows two possible refinements of the conceptual model in Figure 2.3b where Clive and Sally are collaborating. These distribution models specify exactly what physical node each component executes on, and how they communicate.

In the first possibility (Figure 2.7a), the floor plan has been replicated to both computers. The concurrency and consistency maintenance (CCCM) components communicate over the network to keep both floor plans in the same state. When each floor plan is modified, it reports the change to its local editor or viewer via a broadcaster.

The second possibility (Figure 2.7b), instead, centralizes to a single floor plan on Sally’s Computer. Sally’s top-down editor makes calls to the local floor plan normally, via its CCCM. Clive’s first-person viewer, however, most make calls to the floor plan over the network. To improve performance, this refinement has added a cache to that network connection. Events when the floor plan changes are, likewise, sent over the network to Clive’s first-person viewer.

These two implementations have very different performance and failure characteristics. The replicated case requires that Clive’s Computer store and update a complete copy of the floor plan, so might occupy more of its memory than desired. The centralized case gives Clive’s Computer with a lighter-weight cache. However, if the network connection is lost the replicated case could let Clive continue to explore, while the caching alternative would be missing some information.
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

Figure 2.8: Fiia local call rule

Figure 2.9: Fiia remote call rule

2.6 Refining the Models

In addition to the conceptual and distribution meta-models, Fiia provides a set of rules which define how a conceptual model can be refined to a distribution model. Each rule describes a pattern (left) that, if found in the model, can be replaced with a more-refined result (right). The original definition of Fiia [104] contains more information about this rule syntax.

As an example, consider two of the possible implementations of call connectors. Figure 2.8 shows that a call connector between two components (shaded rounded rectangles) on the same node can be implemented as a local pointer. Figure 2.9 meanwhile, shows that a call connector spanning nodes can be implemented using a network transmitter and receiver. Two additional rules allow call connectors spanning nodes to be implemented with caches or via an intermediate proxy node.

These call connectors may be present in the conceptual model, or created by other rules. For example, in Figure 2.7b, one rule merged the two synchronized floor plan components. That rule also replaced the call connector from Clive’s first-person viewer to its local floor plan with one to the central floor plan. This new call connector was
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

then, finally, implemented using the caching approach. These multiple layers of rules are complex, but provide much of the power and flexibility of the Fiia refinement.

Later in this thesis we will show more of the Fiia refinement rules, once we have established a more complete definition of their behavior. Phillips’ Doctoral thesis \[104\] describes the full 34 rules.

2.7 Summary

This scenario exhibits many of the challenges of adaptive groupware. We show participants with greatly different roles and devices sharing information and interacting. They select drastically different user interfaces, with capabilities that best match their requirements. In spite of this, they are aware of the actions of other participants. We show how the natural progression of their collaborations requires significant runtime changes. This involves change in tools (as the quotation is added), devices (as the quotation is moved from PC to PDA). Finally, we show change in participants and location, using a mobile device in a colocated setting.

The adaptations through this scenario have an obvious impact on the user interface. They also require drastic changes in the distributed system upon which the groupware runs. When Clive first connects with Sally, his application needs to establish shared data containing the floor plan. When he disconnects, the once-shared data needs to remain available to his PC and PDA. All of the transitions need to be fast and smooth, without any loss of data, to avoid confusing the participants.

As this scenario has shown, even simple multi-user scenarios encounter many different types of adaptation. To support these scenarios, a toolkit must let developers specify runtime adaptation within a high-level programming model. In the next
CHAPTER 2. CONTEXT: ADAPTIVE GROUPWARE WITH FIIA

In this chapter we examine how existing distributed systems and groupware toolkits have approached this problem.
Chapter 3

Distributed Systems and Groupware

Our primary motivation in this work is easing the development of adaptive distributed systems, particular adaptive groupware. Earlier, in Section 1.1, we gathered five requirements for a toolkit that accomplishes this goal: a high-level programming model, with support for runtime adaptation and failure handling, that offers good performance and easy optimization. In this chapter, we survey the state of the art in toolkits for constructing distributed systems. We focus on how they address the challenges of adaptive distributed systems.

3.1 Programming Models

Much of the history of computer science involves the development of abstract programming models in order to simplify software development. From programming languages themselves to garbage collection and networking libraries, myriad tools
endeavor to spare developers from unnecessary details. This is particularly visible in distributed systems: as computer systems moved from central mainframes to networks of workstations, the number of programming languages for building distributed systems exploded.

By 1988, Bal et al. [9] had gathered nearly one hundred examples of distributed programming languages. These languages simplify distributed system development by offering programmers abstractions of parallel execution, remote communication and partial failures. Even though their goals are similar, many of the languages use drastically different programming models [9, 29]. For example, Orca [10] represents remote communication via sharing of data objects, while Emerald [15] uses asynchronous messages, and Linda [42, 112] exposes a single shared tuple database. Modern languages, libraries and tools, have continued this approach of abstracting distributed issues [135].

Groupware toolkits add more specialized abstractions to their programming models, focused on dealing with multiple users and varying devices. Some of the most common features manage sessions as users join and depart [44, 129], and make users aware of what others are doing [102, 110]. These special-purpose features are built upon lower-level distributed systems primitives, which are also provided by the groupware toolkit. These primitives offer a simple distribution model: higher-level than raw socket programming, but still exposing many implementation details.

When designers and developers use distributed systems toolkits, they work within a programming model that combines distributed systems abstractions with these special-purpose features. Developing within the abstract programming model is intended to make it easier to produce applications. However, limitations of the toolkit
can force developers to abandon this convenient programming model and delve into the internals of the toolkit and distributed system:

**Is my application a good match for this programming model?**

Problems which are easy to solve in one programming model may be difficult in another [5]. Because of its emphasis on distributed people and devices, groupware has different requirements than traditional distributed systems [110] and single-user applications [131].

**How will my application really be implemented?**

Many programming models are very similar to their underlying implementation; for example, remote procedure calls implemented using TCP/IP. Tools providing these views are easy to develop, but limit the level of abstraction and flexibility of the implementation [135]. At the other extreme, when developers can not predict how a more abstract programming model will be implemented, they may be left wary of the tool [5].

**How will the resulting system behave?**

An architecture or tool provides little value if the resulting application is not actually usable. The quality of a groupware system depends on both user interaction [60] and the underlying distributed system [62].

These issues lead to a tension between highly abstract programming models and simple, understandable toolkits. On one hand, developers are more productive working within an abstract programming model suited to their task. On the other, highly abstract toolkits are hard to create or modify, and can make it difficult to understand the behavior of the resulting system [132].
At one extreme, distributed systems can be implemented using plain TCP and UDP sockets. This avoids the need to select and learn a toolkit, and provides developers with direct access to the distribution and communicated data. However, designing, developing, and testing an application at such a low level is substantially more difficult than building atop an existing toolkit.

At the other extreme, developers can choose a toolkit that presents the entire distributed system as a uniform shared-memory space. This makes developing a distributed application as easy as a traditional single-process program. Unfortunately, developers will need to step outside this simple abstraction to deal with partial failures, or to understand the performance implications of their code.

To reduce these problems, the programming models of distributed systems toolkits usually offer multiple levels of abstraction. GroupKit [110] follows a typical pattern by offering four communication styles among distributed applications:

**Shared Data:** GroupKit provides a shared dictionary object. Applications can add, remove, and replace data in the environment using string keys. Changes are automatically propagated to other applications, which can monitor the
CHAPTER 3. DISTRIBUTED SYSTEMS AND GROUPWARE

environment for changes. This high-level structure makes communication among the applications transparent, as well as storage of keys and values. The shared dictionary was implemented by maintaining a complete copy of the environment in each application, and using plug-replaceable protocols to keep the replicas consistent.

Events: Applications can also subscribe to and publish events. These events are automatically communicated between applications, but, unlike the shared environment, are not stored. This gives the applications more control, but discards the pre-existing consistency protocols.

Remote procedure call: Applications may make blocking procedure calls to a single target. This requires one application to know the name and arguments of a procedure in another application, and careful design to avoid latency and deadlocks.

Messages: Finally, applications may send non-blocking procedure calls to one, all, or all other applications. This communication style is a close match to the underlying network communication: messages over TCP/IP.

Each of the four styles provides different compromises between abstraction and control. GroupKit uses a simple model of communicating TCL processes, so these communication styles also impose a structure on applications. This was extended in TeamRooms [111], which splits each application into cooperating widgets. A similar design appears in many other toolkits, including Fiia, to give developers a choice of different communication abstractions between components. The compromise in this approach is that application developers must understand all of the communication
styles to take advantage of their different features. In addition, the toolkit developers need to implement and optimize the wide variety of different options.

The alternative – a single communication style – often appears in toolkits designed to transparently distribute an existing programming language. Here, toolkit developers face the difficult task of making the single communication style work well across many different uses. J-Orchestra \cite{86} presents communication between Java objects as blocking method calls. Behind the programmer’s simple model, the toolkit uses many tricks to minimize the difference between remote and local calls. For example, it copies modified parameter objects back to the caller \cite{128}, and will rewrite threading and locking methods to support distribution \cite{127}.

Communication styles are the most obvious example of differing levels of abstraction. However, similar choices are needed in designing many other aspects of a distributed systems toolkit. These issues include distributing threads \cite{127}, resource management \cite{79}, code mobility \cite{88,45}, group awareness \cite{40,64}, and access control \cite{41,126}. Across all of them, toolkit designers are searching for abstractions which ease development without constraining or confusing developers.

This past development of distributed systems toolkits shows that raising the level of abstraction can serve to ease application development. However, highly abstract programming models are difficult to implement, and pose a risk of poor generality, flexibility, maintainability, and performance. With these issues in mind, we next examine the particular challenges of adaptive distributed systems: changing the system at runtime, handling partial failures, and optimizing the implementation.
3.2 Adaptation

Distributed systems, and particularly groupware, have always encountered changing environments. In some of the earliest toolkits, these changes were merely to expand a virtual mainframe when more physical systems were added [5]. As the cost of computer hardware decreased, the scope broadened to include networked workstations and more diverse servers [124]. Users did not need to consider whether the distributed system had changed: they could simply issue a command, and then thank the administrator for it running faster than previously.

The modern ubiquity of mobile users and devices has expanded both the importance of such flexibility, and the variety of adaptation needed. An important difference is that these new environments are unmanaged: no system administrator is available – or trusted – to reconfigure the smartphones of everyone walking through a park. This leaves the distributed systems responsible for automatically discovering their environments, and adapting to improve their users’ experiences. The added complexity drives most mobile software to avoid even the adaptations of traditional distributed systems, in favor of simple architectures with centralized servers [76].

In spite of the complexity of adapting distributed systems at runtime, the potential benefits have attracted substantial research effort. These works span many fields of computer science and software engineering: from model analysis and transformation to software architectures and communication protocol design. Indeed, research in adaptive software is by no means limited to distributed systems or groupware [18, 98, 125]. Within this huge body of literature, four areas are of particular interest for adaptive groupware:
CHAPTER 3. DISTRIBUTED SYSTEMS AND GROUPWARE

**Discovery:** Automatically finding interesting properties of the environment, such as available devices [46], network and processor limits [108], and users who happen to be nearby [73].

**Conceptual Adaptation:** Adjusting the overall system [17] and user interfaces [94, 23] to take advantage of unplanned environments and spontaneous interactions.

**Failure Handling:** Avoiding and recovering from partial failures; e.g. replicating checkpoints of long-running jobs [134], and deriving repair procedures from software architecture [54].

**Optimization:** Adjusting the system to behave well with the available devices and networks; e.g. using a low-power interface on a smartphone when the battery is limited [26], or generating network configurations tuned for different environments [14].

These diverse systems satisfy many of our individual requirements for an adaptive groupware toolkit, as stated in Section 1.1. However, in doing so, they have moved away from the abstract programming models found in many groupware toolkits. Their concrete programming models simplify the toolkit and grants more control to the application developer, at the cost of more complex applications. In our vision for adaptive groupware toolkits, all of these powerful adaptations could be implemented at a low level – as found in the literature – but requested via a high-level programming model.

Several of these areas have already been examined in association with Fiia. Qiu [108] has proposed a discovery system and platform built within a Fiia-based
CHAPTER 3. DISTRIBUTED SYSTEMS AND GROUPWARE

toolkit. Phillips \[104\] describes briefly how Fiia could support conceptual adaptations, failure handling and optimization. However, both of these works fall short of a practical system for synchronizing the programmer’s model with the low-level details.

For a more extensive review of discovery techniques, we defer to Edwards \[45\] and Qiu \[108\]. Failure handling and optimization are special cases of adaptation which deserve further treatment – we will return to them in the next section. We now continue to review approaches for supporting conceptual adaptation.

### 3.3 Conceptual Adaptation

Most toolkits for distributed systems and groupware provide incidental support for conceptual adaptation: the toolkits allow developers to create and delete objects, but offer no features for coordinating larger-scale adaptations. As a result, developers are either forced to build complex coordinate systems, or forego many adaptations entirely. The flexibility motivated by adaptive groupware requires substantially more support.

Several approaches based in user-interface plasticity offer to ease conceptual adaptations. *Plasticity*, in this context, is the ability of a user interface to adapt to a variety of devices, environments and uses \[22\]. Traditional work in plasticity has focused on user interfaces. For example, widgets that behave differently on different devices \[74\], or generating interfaces tailored to a device \[120\]. These tools allow an application to work to different devices, but do not ease moving data between the devices or connecting them together. These obvious needs have been integrated into the modern sense of plasticity \[24\], and prompted many experimental solutions \[91\] \[100\].
Extending plasticity from single devices to flexibly cooperating networks requires that different systems support mutual discovery and integration. Edwards describes this as *serendipitous interoperability* [46], motivated by scenarios very similar to our own. Edwards argues that three properties can allow applications to recombine at runtime, and offer this new interoperability:

**Well-Known Interfaces:** If components of the distributed system are to discover each other and communicate, they must support some common interfaces. Edwards proposes a small set of simple interfaces for communication, discovery, inspection, and control.

**Dynamic Extensibility:** For flexibility and efficiency, components need the ability to extend the well-known interfaces at runtime. Edwards suggests mobile code as a means to add specialized functionality to the generic interfaces; for example, a network video stream could provide a custom decoder to make its compression transparent to the receiver.

**User Arbitration:** To avoid limiting interoperation, users need to understand the behavior of entities, and choose when they are used. Edwards uses the example of a PDA detecting a generic “printer” device, which the user would interpret to mean data sent to the device would be printed.

Edwards et al. have built the Speakeasy toolkit based on these properties, and used it to implement several applications [48, 47]. An important extension is the combination of dynamic extensibility with user interfaces [94]. This offers users control of devices without sacrificing the simple interfaces and genericity; like whether the “printer” device is using portrait or landscape layout.
Speakeasy’s properties offer an interesting contrast with our requirements for an adaptive groupware toolkit. Adaptive groupware and serendipitous interoperability have significant overlap, as evidenced by the similar examples. However, our scope incorporates more general adaptations, such as changing what is shared between participants in a collaboration. Speakeasy, meanwhile, focuses on the lower-level interactions between components, and how their communications are implemented.

These approaches offer varying levels of support for conceptual adaptations. Mobile devices have motivated many different methods for migrating user interfaces, but most offer only rigid implementations. A notable exception is Speakeasy, which was designed explicitly to let dynamically-discovered components interoperate. It imposes a simple peer-to-peer structure, but uses mobile code to customize the implementation. Our motivation goes beyond these local adaptations to express arbitrary changes in the programmer’s conceptual model.

### 3.4 Failure Handling

Handling partial failures was one of the original motivations for distributed systems. By spreading a computation across loosely-coupled devices they gained easy fault tolerance [9]. This has developed into a tremendous breadth of research, focused on reliable distributed systems. These systems can continue functioning through unexpected failures, like nodes crashing or network failure partitioning the system (e.g. [81], [87], [114]).

The modern proliferation of lightweight mobile devices has made groupware ubiquitous, but also exposes it to highly unstable environments, including wireless networks. Mobile wireless networks are prone to both short outages – for example,
as a user drives through a tunnel or between cell towers – and longer disconnections. Letting users continue to collaborate through failures requires a careful balance between transparent handling and meaningful notifications [67, 68]. However these failures are initially detected at a low level, and are tightly tied to the particular implementation.

The most common approach to failure handling is fault tolerance. This is based on designing the distributed system so that common or expected forms of partial failure – typically network failures and node crashes – can be easily handled by the application. For example, a collaborative text editor could choose to store a single copy of the shared document on the server, or replicate the document to every connected client. In the latter case, the system is tolerant to network failures: clients can continue editing their local copies even if the server is inaccessible. Such fault tolerance is relatively simple to use from an application, but relies on the distributed systems toolkit to actively perform repairs.

Argus [6] is a classic example of a toolkit that provides fault tolerance. The Argus programming language is intended for distributed computation, using a model of parallel tasks with shared memory. Developers explicitly mark variables as stable, and then indicate when the runtime system should save a checkpoint. If a task crashes, it is restarted with the data from its most recent checkpoint.

Hall et al. [68] look explicitly at the problem of fault tolerance in groupware, and conclude that user intentions and roles have a dramatic effect on fault tolerance. They use this basis to propose a groupware toolkit with two communication styles, tailored to the needs they observed.
CHAPTER 3. DISTRIBUTED SYSTEMS AND GROUPWARE

More transparent approaches to fault tolerance have also been widely used. For example, both CORBA [81] and MPI [134] have been extended to provide fault tolerance via replicas. These replicas are automatically created on various nodes, and used to conceal failures in the distributed system.

A different approach to failure handling appears when applications are aware of failure, and take active steps to resolve it. These applications are referred to as self-healing. An improvement to the document editor example above could automatically reconnect to the server and merge any pending changes. This simple self-healing is typical of many groupware applications, and supported by some toolkits [67].

More complex self-healing allows the distributed system to work around errors. For example, if the document server crashes, one of the clients could become the new server. Several groups have proposed self-healing using model-based techniques [36, 55], however their solutions rely on the developer to write explicit repair rules. These rules are specified at a low level, violate our requirement for handling failure in the high-level programming model. However, these approaches and other initiatives in self-adaptive systems emphasize the value of formal models for understanding and resolving failures.

Our motivation in adaptive groupware requires both fault tolerance and self-healing. To maintain abstraction, application developers must be able to understand and respond to failures within their high-level programming model. However, the underlying functionality needed to detect failures and provide fault tolerance must be linked to the actual implementation. This separation again requires a bridge between the implementation and the programming model which can propagate changes at runtime.
3.5 Optimization

Poor performance in groupware applications will severely detract from the user experience, whether from network delay \[65\], processing time or other quality attributes \[113\]. This is particularly the case with mobile devices, as applications must allow for limited processing power, memory, and battery life. The abstract programming models motivated by adaptive groupware pose two major risks to performance: developers may build poor systems because they can not predict the effects of design choices, or the underlying implementation may fall short. In either case, developers need the ability to experiment with the system, and incrementally optimize it.

These optimizations fall into two broad categories: changing the structure of the distributed system, and tuning the infrastructure implementing that structure. The first encompasses architectural decisions \[62, 103\], like whether data is stored on one node or replicated to many \[31, 77\]. The second captures many lower-level decisions \[95, 130, 2\] like network protocols, thread management, and how different replicas are kept consistent. In common distributed systems, all of these decisions are hard-coded by application developers or toolkit developers.

Distributed systems and groupware researchers are no strangers to either category of optimization. Many toolkits for parallel computation perform structural optimizations. Typical examples can automatically allocate work across nodes \[7, 27\] or place replicas to improve speed \[8, 107\]. Within groupware, Patterson’s Taxonomy \[101\] describes variants of the same system with different combinations of centralized and replicated components (Figure 3.3). Different infrastructure choices have likewise been widely researched, such as the numerous network protocols \[66, 89, 133\],
techniques for maintaining consistency between replicas \[43, 122, 32\], and approaches for handling concurrency \[93, 139\].

These numerous techniques offer developers myriad possible implementations of a distributed system. Choosing a combination of techniques that will behave well in a groupware applications is difficult, particularly as users may place unexpected demands on the system \[61\]. Moving to adaptive groupware and mobile devices expands this problem: not only is a good design hard to create, but a design which is good in one environment may be poor in another. Alleviating this problems requires not only easy experimentation with different implementations, but also the ability to change between them at runtime \[25\].

Unfortunately this runtime optimization is not merely a matter of switching one aspect of the implementation. For example, consider the move from a single centralized server with multiple clients (such as Figure 3.3b) to a completely peer-to-peer architecture (Figure 3.3b). Getting the best results from this change might also
CHAPTER 3. DISTRIBUTED SYSTEMS AND GROUPWARE

require new concurrency control algorithms, replacing caches with replicas, adding consistency maintenance support or changing network protocols.

Several promising techniques for choosing and applying optimizations have been based on *dynamic variability*. These techniques developed out of *variability modeling*, which was traditionally used to produce different versions of a software product. The complex inter-dependencies between design, features, and code drove practitioners to formal model-based methods [99]. Analogous techniques applied to the implementation of a distributed system help both developers and the toolkit perform optimizations.

The most notable example of this dynamic variability is Genie [14, 13], which generates multiple possible implementations from developer-supplied policies [92], and can switch between them at runtime [34]. These reconfigurations can be driven automatically based on formal goals [28], analogous to quality-of-service requirements in computer networks [57]. Similar systems have been proposed to adapt collaborations in groupware and to automatically choose user interaction techniques [90].

Dynamic variability offers a powerful means to generate and switch between different implementations, with different performance characteristics. This capability is crucial for adaptive groupware. Unfortunately, these systems do not provide the high-level abstractions or conceptual adaptations also required by adaptive groupware. While they fall short of our conceptual model, they are still valuable examples of adapting and choosing between different distribution models.
3.6 Summary

In this chapter we surveyed a wide variety of toolkits for constructing distributed systems, each focused on a different problem shared with adaptive groupware. Examples like GroupKit and J-Orchestra provide high levels of abstraction and simple programming. Speakeasy exposes more implementation details, but adds discovery and serendipitous interoperability. Genie steps even farther toward the implementation, with few high-level abstractions but extensive support for runtime optimization.

These varied toolkits solve the individual requirements of adaptive groupware, but merging their capabilities is far from a trivial task. Our toolkit will need a highly abstract programming model, which can still express conceptual adaptations. It must keep the system consistent with the programmer’s conceptual model during failures, and provide useful notifications. Finally it must offer good performance, and permit extensive optimization by application developers.

Our diverse requirements leap from the high-level programming model to the underlying implementation, with runtime changes propagating from both levels. This complexity requires a strong separation of concerns, to avoid a toolkit exponentially more complex then the state of the art. As used in both model-driven engineering and dynamic variability, we turn to formal models and model transformation. The next chapter examines how model transformation can be used to satisfy the requirements of a toolkit for adaptive groupware.
Chapter 4

Model Transformation

Models of software have long been used to simplify design and development, as popularized by UML. They can describe anything from how users interact with software, to the high-level architecture of a large system, to individual objects and references in an application [52]. With the use of models comes the need to express changes to models, and describe relationships between models [117].

Model transformations provide a formal way to edit models, describe changes, and relate different models. Transformations are often applied to link pairs of models that describe the same software architecture at different levels of abstraction. This approach was popularized by the OMG MDA [96] initiative; refining from a platform-independent model to a platform-specific model. In common practice, such model-driven engineering involves refining a high-level design model into code templates that developers complete. As designers and developers modify the design and source code, they can easily become inconsistent. Some model transformations can automatically propagate such incremental changes through the refinement. These
CHAPTER 4. MODEL TRANSFORMATION

traditional techniques maintain design documents against code, so all transformations are finalized before the application is deployed.

The popularity of model-driven engineering does not imply that model transformation is restricted to software architectures. Indeed, models and transformation have been applied in many other areas, including: interpreting mathematical notation in scanned images [17], building editors for new diagrammatic notations [58], integrating different engineering specifications of a chemical process [12] and optimizing programs during compilation [4].

Many model transformations represent their models as graphs, and use graph transformations to perform the actual manipulation [35]. This approach separates the semantics of the models and transformations from general-purpose data structures and algorithms used to implement them. Developers can then use existing libraries for model transformation, rather than manually writing code to search and edit data [16].
CHAPTER 4. MODEL TRANSFORMATION

4.1 Rewriting and Relational Transformations

Graph transformation has been extensively researched, both as a theoretical problem and as part of applications like model transformation. This previous work falls into two broad families, based on how they define the relationship between conceptual and distribution models:

Rewriting transformations (Figure 4.1a) defines a conversion from conceptual to distribution as a sequence of steps. Each step performs a specific modification to the model. Performing all of the steps in turn converts the conceptual model into the distribution model. Many transformation approaches fall within this family, including AGG [123], Fujaba [50], PROGRES [115], and GREAT [30].

Relational transformations (Figure 4.1b) instead define constraints between the conceptual and distribution models. Each constraint directly relates part of the conceptual model to part of the distribution model, without any intermediate models. Common examples of these approaches include QVT Relational [97, 80], and tools using Triple Graph Grammars [38], like ATOM³ [39].
CHAPTER 4. MODEL TRANSFORMATION

The choice of rewriting or relational rules greatly changes the capabilities of a tool. In general, rewriting rules for a given transformation are more compact and express complex transformations in incremental steps. Relational rules, meanwhile, are easier to verify and provide good support for updating existing transformations. Both capabilities are required for our application of model transformation at runtime to adaptive distributed systems, so the details of these transformation families are important.

As an example for both families, consider the model in Figure 4.2. This model simply shows that component A can make blocking calls to component B over connector C. We define two rules, sketched in Figure 4.3: the first marks a component as implemented (the grey triangle), and the second implements a call connector (\(\rightarrow\)) as a local pointer (\(\rightarrow\)).

In a rewriting transformation, rules are applied to the model sequentially. We arbitrarily choose to first apply the component rule to B, yielding the model shown in Figure 4.4a. This intermediate model contains a mix of elements from the conceptual model and elements that will appear in the distribution model. Applying the call
CHAPTER 4. MODEL TRANSFORMATION

connector rule to C, followed by the component rule to A, yields the distribution model in Figure 4.4b.

Relational transformations, meanwhile, apply rules to the model in parallel. We again arbitrarily choose to first apply the component rule to B. This leads to the conceptual model being related to a incomplete distribution model, as shown in Figure 4.5a. Unlike the intermediate model in the rewriting transformation, this incomplete distribution model does not contain any untransformed parts of the conceptual model. Applying the call connector rule to C and the component rule to A yields the conceptual model related to a complete distribution model, shown in Figure 4.5b.

Thus far, the two transformations have used the same rules in identical orders, and produced exactly the same distribution model. However the distinction between
CHAPTER 4. MODEL TRANSFORMATION

Figure 4.6: Example conceptual model with events passing between two components

Figure 4.7: Example rule for refining subscription connectors

(a) One intermediate model  (b) Distribution model

Figure 4.8: Example of a rewriting transformation

Intermediate model and incomplete distribution model becomes important as the transformation grows more complex.

Figure 4.6 shows two components connected by a subscription connector ( الجنسية), which transports asynchronous events. Obviously this connector can not be implemented as a simple pointer, because it must somehow buffer and dispatch events. This added functionality is provided by a broadcaster (الذكاء). As shown in Figure 4.7, the subscription connector can be implemented by a blocking call into a broadcaster, which will later make a call to the conceptual destination.

In a rewriting transformation, this rule can simply be used along with those shown in Figure 4.3. As earlier, we apply the component rule to B and A, and then transform the connector. This yields the intermediate model in Figure 4.8a with a broadcaster and two call connectors. These two call connectors are intermediate elements, because they exist only in intermediate models, rather than being part of the conceptual or
CHAPTER 4. MODEL TRANSFORMATION

Figure 4.9: Incorrect relational transformation with intermediate elements

Figure 4.10: Corrected subscription rule for relational transformation

distribution models. FiiA uses intermediate elements widely in its rules, to provide functional separation and avoid duplication. Applying the call connector rule twice implements the intermediate elements, yielding the distribution model in Figure 4.8b.

With the relational transformation, rules must map directly between the conceptual and distribution models. As a result, intermediate elements can not be used. Using the same subscription connector rule would yield the incorrect transformation shown in Figure 4.9; here the final distribution model still contains unimplemented call connectors.

Avoiding this problem in a purely relational transformation requires modifying the rules to ensure that they produce only elements for the distribution model. This correction is easy in our simple example (Figure 4.10), because call connectors can only be implemented using local pointers. Adding these rules would leave three possibilities for each of the two call connectors in the subscription connector rule, requiring nine near-identical relational rules. This exponential explosion of rules slows the transformation and makes it hard to understand or debug.

Other alternatives used in the literature begin to blend rewriting behaviors with the relational transformation. Some, such as triple graph grammars, allow rules to
create linking elements between conceptual and distribution. The linking elements can be matched by later rules, and serve some of the purposes of intermediate elements in rewriting. Other alternatives rely on the rule author to structure them into multiple layers, or define an explicit data flow between rules. All of these alternatives add substantial complexity to the transformation engine and rules, and undermine the ease of updating transformations.

The compromises of existing rewriting and relational transformations are unacceptable for our work. Supporting an adaptive distributed systems toolkit requires both very expressive transformations and fast, stable updates. Rewriting approaches are attractive because they can capture nested decisions – like the multiple implementations of an event stream – using intermediate elements. However, the update capabilities of relational approaches are equally necessary. In the next section we examine how existing model transformation techniques perform updates, and the expressiveness limitations these impose.

4.2 Transformation Update

Using model transformation at runtime to support adaptive distributed systems requires flexible, rapid, and stable updates. User- and application-driven adaptations may cause arbitrary modifications to the conceptual model. Meanwhile, partial failures will be reported in the distribution model. Both kinds of change must be propagated through the transformation quickly, and with minimal effects on the models. These bidirectional updates are extremely challenging to support along with the expressive transformations required for adaptive distributed system implementations.
CHAPTER 4. MODEL TRANSFORMATION

![Diagram of three forms of update in model-to-model transformations](image)

(a) Forward-only
(b) Bidirectional
(c) Parallel

Figure 4.11: Three forms of update in model-to-model transformations

Maintaining transformations while models change is an active area of research, and has attracted a wide variety of techniques [3]. Current approaches fall into three broad categories:

**Forward-only update** (Figure 4.11a) applies the graph transformation to the conceptual model and generates the distribution model. This approach allows the most expressive transformation, but is slow to update and only a few variants support distribution changes.

**Bidirectional update** (Figure 4.11b) uses a reversible graph transformation to support changes in both conceptual and distribution models. Conceptual changes are handled via a forward update, while changes in the distribution trigger the reverse. This approach typically requires relational rules, though a few restricted rewriting techniques can be inverted.

**Parallel update** (Figure 4.11c) derives equivalent changes in both models. This approach can be very efficient, and avoids unnecessary changes in either model. However, it requires rules to deal with each possible change, which must typically be written by hand.
As a simple example, consider a conceptual adaptation used to create an event connector between two components, shown in Figure 4.12. This must produce the transformation shown earlier from Figure 4.6.

Through the remainder of this section we show update techniques from the different categories, and how they perform this example update. Finally we compare the techniques with our requirements for model transformation at runtime in an adaptive groupware toolkit.
4.3 Forward-only Update

Current transformation techniques demonstrate two approaches to updating based on forward-only transformations. Most transform the new conceptual model using a rewriting approach (Figure 4.13a, \( C_2 \) to \( H \)), the compare the old (\( D_1 \)) and temporary (\( H \)) distribution models to derive a distribution change (from \( D_1 \) to \( D_2 \)). A few instead analyze the pre-existing transformation (Figure 4.13b, \( T_1 \) to \( T_2 \)), and use this as a basis of the distribution-level update.

Metke [75] provides a good example of the first approach, based on the Tefkat [84, 83] rewriting engine. When a conceptual change occurs, the transformation engine derives a new distribution and compares it with the previous one. Applied to Figure 4.12, this would refine \( C_2 \) and compare the result with \( D_1 \). This comparison finds that two local connectors and a broadcaster appeared, so performs the same addition to get the final distribution \( D_2 \). Rather than simply replacing the old distribution, Metke’s engine will update only parts created by the old transformation. This is useful in Metke’s intended application of updating code templates, as any developer modifications will remain untouched. However it is not applicable to our
CHAPTER 4. MODEL TRANSFORMATION

use in adaptive distributed systems, and can not propagate the distribution changes back to the conceptual model.

Hearnden at al. [69] present an interesting technique for incrementally updating a model transformation. Their formulation maintains a tree of all possible derivations in a logic language, which has similar semantics to a rewriting transformation. The advantage of their approach is that the tree does not need to be reconstructed for each conceptual change. Instead, branches are incrementally expanded or pruned based on the addition and removal of conceptual elements. Unfortunately, the structure of their tree depends heavily on the order in which the logic engine performs derivations: if a variable is resolved near the root of the tree, a change to it will force the entire tree to be reconstructed. In our example, the first rule applied happens to be implementing $B$. Using Hearnden’s approach, a conceptual adaptation deleting $B$ would trigger a complete new forward transformation, which loses the speed and stability advantages of incremental update.

Model transformation techniques based on forward update are very interesting for our purposes, because they support the full power of rewriting transformations. However they have two significant weaknesses: First, none can propagate distribution changes back to the conceptual model, as required to report failures in our application. Second, these techniques suffer from stability problems. If the refinement is non-deterministic – that is if $C_1$ is refined twice, it might produce two different distributions – the transformation may unexpectedly switch distributions. Moreover, even if the refinement is deterministic, small changes in the conceptual model may cause unnecessarily large changes in the distribution.
CHAPTER 4. MODEL TRANSFORMATION

As an example of these unnecessary distribution changes, consider a rule which assigns components to physical nodes in the distributed system. If components are assigned randomly, the refinement will be non-deterministic, and any change will require many components to be copied. If, instead, components are assigned to the node with the closest identifier (as is common in distributed hash tables), the placement is deterministic. However, adding a new node may still require a number of components to be moved due to this policy. Because these changes require copying data over a network, and will interrupt user tasks, we want to ensure that distribution changes are kept to a minimum.

These weaknesses render existing forward update techniques inappropriate for our application to adaptive distributed systems. However Hearnden at al. show that modifying an existing transformation offers the possibility of improved stability.

4.4 Bidirectional Update

The most common approach to performing rapid and stable updates is to restrict the transformation such that bidirectional update is possible [121]. These transformations usually follow the relational style, so rules must map directly from the conceptual to distribution models. They typically also require that the transformation be injective; that is, each conceptual model corresponds to one unique distribution model, and vice versa.

Triple Graph Grammars (TGGs) [116] are a common basis for incremental bidirectional transformations (e.g. [56] [78] [12]). TGGs add several extensions to relational transformations, but still allow updates to be applied using a search for compatible constraints [49]. Returning to our example of a conceptual update in
CHAPTER 4. MODEL TRANSFORMATION

Figure 4.14: Conceptual update using a Triple Graph Grammar

Figure 4.12, the TGG will begin with links attached to A and B (Figure 4.14). However, the new event connector is not linked, so has not yet been refined. The only compatible rule (Figure 4.10) is used to create a new link and the corresponding broadcaster and local connectors.

Automatic bidirectional update is typically restricted to such relational transformations. QVT [97] provides two user languages, which are typical approaches in the literature: QVT Relational is similar to TGGs [63], and supports bidirectional update but, as a relational transformation, forbids intermediate elements. QVT Operational, meanwhile is an imperative language that allows more complex rewriting transformations, but requires hand-coded reverse rules to support bidirectionality. Given the complex rules required to implement adaptive distributed systems, manually writing reverse rules is infeasible.
CHAPTER 4. MODEL TRANSFORMATION

The Atlas Transformation Language (ATL) is a notable exception. It is an imperative rewriting technique, which has been extended by Xiong et al. [141] to support arbitrary changes at the conceptual level, and removals at the distribution level. This behavior is provided by automatically inverting the ATL program specifying the transformation. Unfortunately, the transformation must be deterministic and injective. Moreover, as in forward update, small changes in the conceptual model may produce large distribution changes.

Hettel et al. [71] present an interesting approach to bidirectional update of forward-only transformations, based on logic programming. Their engine performs reverse updates by searching for a set of changes in the conceptual model that could explain the observed distribution changes. While their process is subject to further optimizations, Hettel observes [70] that small changes require "seconds to several minutes".

Bidirectional update of relational transformations, like TGGs, provides fast and stable update. However, these techniques place significant restrictions on the supported transformations, which make them unsuitable for specifying adaptive distributed system implementations. The rare techniques which provide bidirectional update based on other transformations have their own limitations: they either place severe restrictions on the transformation, or are far too slow to control a live distributed system.

4.5 Parallel Update

In databases, the view-update problem is a classical application of parallel update. A view can be thought of as the result of a database query, often renaming columns
and combining information from different tables. Such views are a convenient way for applications with differing needs to access the same data in different forms. However, for the view to be modified, changes must be propagated back to the physical tables. This process is neither simple nor always possible, so became known as the view-update problem [11]. Dayal and Bernstein [37] presented a solution using parallel update, which converts view updates into updates on the underlying tables. Extensions of this approach continue to simplify the generated table updates [85]. The relationship between a view and its underlying tables is much simpler than typical model transformations.

To perform parallel update, rules are defined to execute upon specific changes in the models. For example, to support the earlier conceptual update example (Figure 4.12), a rule author would need to specify that adding a subscription connector at the conceptual level adds two local connectors and a broadcaster at the distribution level. The corresponding removal would require another rule. This differs from relational rules, which establish automatically-managed constraints between the models.

Réth, Varró and Varró [109] present an approach to parallel update based on high-level patterns. When patterns appear or disappear in the conceptual model, rules are triggered which revise the distribution model. These rules need to be written manually, but the automatic pattern matching allows them to be simpler than traditional event-based approaches.

Parallel update is, in theory, faster and more flexible than bidirectional update of relational transformations. This is because arbitrary events (or patterns in Réth et al.) can trigger rule-specified changes. However, where there is not a clear
CHAPTER 4. MODEL TRANSFORMATION

<table>
<thead>
<tr>
<th>Name</th>
<th>Expressive Rules</th>
<th>Conceptual Adaptations</th>
<th>Distribution Adaptations</th>
<th>Fast Updates</th>
<th>Stable Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Metke [75]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Hearnden [69]</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>TGG [78]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>QVT [63]</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>ATL+Xiong [141]</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>○</td>
</tr>
<tr>
<td>Hettel [71]</td>
<td>○</td>
<td>○</td>
<td>●</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>Ráth [109]</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>●</td>
</tr>
</tbody>
</table>

Each requirement is:
- ○ satisfied;
- ● partially satisfied; or
- ○ not satisfied

Figure 4.15: How existing model transformation techniques satisfy the requirements of model transformation at runtime

correspondence between the conceptual and distribution models, these rules are extremely complex to define and ensure correct. Refining a distributed system to an arbitrary implementation leaves no such clear correspondence.

4.6 Summary

Existing techniques for model transformation cover a tremendous range of approaches, and draw from many areas of computer science. This variety provides many different blends of features, outlined in Figure 4.15. Unfortunately, none of this wide body of work satisfies the requirements of model transformation at runtime for adaptive distributed systems.

Resolving these limitations requires an improvement to the state of the art, which is the core contribution of our work. Over the next chapters we introduce and specify our extension of a rewriting transformation to fast, stable, and bidirectional updates.
Chapter 5

Framework for Graph Rewriting and Tracing

In introducing our work we outlined the need for, and promise of, adaptive distributed systems. These systems have the potential to let software adapt to dynamic human demands, as users change both what they are doing and how they are doing it. Making such flexible applications practical to develop requires a new breed of distributed systems toolkits. These toolkits must offer high-level programming models and natural support for runtime adaptation, together with excellent performance and failure handling. While the toolkits reviewed in Chapter 3 offer valuable subsets of these properties, their combination has previously eluded the state of the art. By applying model transformation at runtime we are able to demonstrate these capabilities, both in theory and with a prototype toolkit for adaptive groupware.

However, our use of model transformation imposes a challenging combination of requirements. The model transformation techniques in Chapter 4 present a choice between simple transformations and time-consuming processing. To bridge
between a distributed systems programming model and optimized implementation, the transformation must be extremely powerful and flexible. At the same time, adaptations to the running distributed system are performed by updating the models and transformation. These updates must be applied very quickly, on the order of hundreds of milliseconds, while minimizing user-interrupting changes to the distributed system. Providing this combination of power and speed, beyond the capabilities of earlier model transformation techniques, is the core contribution of this thesis.

We approach model transformation algorithms from the well-understood and well-studied area of graph rewriting. The rewriting techniques in Chapter 4 have demonstrated the power and flexibility we need, but fall short in their update speed and stability. However, the runtime updates required by adaptive distributed systems are typically small and local, caused by specific user requests (as we saw in Chapter 2), optimizations, or partial failure. By exploiting this property, we can ensure that our updates are faster and more stable than completely re-transforming the conceptual model. Unlike previous rewriting approaches, our algorithm revises an existing transformation. Because our algorithm reuses most of the transformation – the large parts unaffected by the local update – we can avoid unnecessary new rewriting, and the associated speed and stability problems. The result of our approach is that, even in our prototype implementation, updates to a large transformation are accelerated by a factor of ten (see Chapter 9 for our detailed methodology and results).

Revising an existing transformation is significantly more difficult than building a new one. To do so, our algorithm must examine and manipulate the transformation in a way that guarantees its result is correct. The intuition behind this correctness is simple: the graphs produced by our algorithm must be a possible result of running
the transformation in the traditional fashion. The gains of our algorithm come from reducing expensive searches and non-determinism, in favor of “what we did last time”.

To define our algorithm and argue its correctness, we need a formalism which can express both general graph rewriting and revision of an existing transformation. While many different formalisms are demonstrated in the literature, they focus on expressiveness and safety rather than later manipulations. As a result, they are relatively complex, and would be challenging to extend for our purposes. Instead we turn the one of the earliest, and simplest, approaches for representing graphs: as sets of edges. Starting with set theory, we construct a formalism for graphs and graph rewriting which supports both highly-expressive rules and later revision of an existing transformation. Our formalism represents the existing transformation as a detailed trace of the transformation, so we describe our algorithm as trace-based update.

The \textit{Fiia} models contain much more complex features than our simple set-theoretic graphs, including types, nesting and various symbols. To represent \textit{Fiia} models and refinement rules, these must be translate to a corresponding graph. This conversion offers two important advantages: first, it spares our graph formalism and algorithms from the modeling features used by \textit{Fiia}; and second, it ensures that our algorithm can be used with meta-models and transformation rules other than \textit{Fiia}.

Through this chapter we build from a set theoretic representation of graphs toward our graph rewriting formalism. This formalism later serves as the foundation for our core trace-based update algorithm in \textit{Chapter 6} and several extensions in \textit{Chapter 7}. We introduce our formalism in several stages: \textit{Section 5.1} introduces our core graph definition based on sets of edges, \textit{Section 5.2} defines a pattern language and the process of matching subgraphs, \textit{Section 5.3} defines a language for rewriting rules
CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING

over graphs, Section 5.4 combines multiple rewriting rules to transform a graph and, finally, Section 5.5 explains how this forward transformation is recorded for later modification.

5.1 Graph Structure

We define a graph as a set of directed edges with labels. Each edge is represented by a triple of source vertex, label, and target vertex, written as \( \langle \text{source}, \text{label}, \text{target} \rangle \). Vertices and edge labels are simple identifiers, so two triples are equal if they have the same source, label and target. A useful result of this graph definition is that the vertices of the graph are implicit. That is, a vertex \( v \) is part of graph \( G \) if and only if it appears as the source or target of some edge.

For example, consider the graph for part of our scenario from Chapter 2:

\[
G = \{ \langle \text{floor\_plan}, \text{is}, \text{Store} \rangle, \langle \text{first\_person\_viewer}, \text{is}, \text{Actor} \rangle, \langle \text{connector}, \text{is}, \text{Call} \rangle, \\
\langle \text{connector}, \text{source}, \text{first\_person\_viewer} \rangle, \langle \text{connector}, \text{target}, \text{floor\_plan} \rangle \}
\]

This graph contains five edges, which include six vertices \{ floor\_plan, first\_person\_viewer, connector, Store, Call, Actor \} and three edge labels \{ is, source, target \}. Operations on the graph are performed using classical set manipulations. For example, \( (G - \{ \langle \text{connector}, \text{is}, \text{Call} \rangle \}) \) would produce a modified graph with that single edge removed. Likewise we will use union (\( \cup \)), intersection (\( \cap \)), and a set-builder notation to manipulate graphs. The latter will be written following \{ \( f(x) \mid x \in G, p(x) \} \) to create a set of \( f(x) \) for all edges \( x \) in \( G \) where \( p(x) \) is true.
While we represent and process all graphs as sets, displaying them as such obscures the content and structure. Where possible, we will instead draw graphs in the common “boxes and arrows” form, as in Figure 5.1a. In these diagrams the labels on vertices and edges correspond to the identifiers in our set, with slight changes to punctuation. As the graphs grow larger, this visual form become unwieldy. For these cases we will instead use a simplified form of the set notation, shown in Section 5.1b. This form is identical to the earlier set notation, but lacking the excessive punctuation.

Throughout these graph examples we will use different fonts to indicate the source of an identifier. Like the different notations, these fonts are for human comprehension, and are not considered by our formalism or algorithm. Identifiers based on our scenario in Chapter 2 will be presented in an sans-serif font. Identifiers which are defined by Fiia will be written in the normal font. Finally, variables, which we introduce in later sections of this chapter, will be written in italics.
Representing graphs as sets of edges makes manipulations much simpler than in theories with explicit vertices, types and attributes. Because the graph contains nothing but edges, any edit can be expressed as a set of removed edges ($\Delta^-$) and a set of added edges ($\Delta^+$). For example, let us replace the first_person_viewer with a quotation store:

$$\Delta^- = \{ \langle \text{first_person_viewer}, \text{is}, \text{Actor} \rangle, \langle \text{connector}, \text{source}, \text{first_person_viewer} \rangle \},$$

$$\Delta^+ = \{ \langle \text{quotation}, \text{is}, \text{Store} \rangle \}$$

To delete the first_person_viewer vertex, this edit removes both associated edges, leaving Figure 5.2a. It then adds an edge involving the new quotation vertex, which produces Figure 5.2b. Because the vertices have no internal state, there is no additional cleanup for the first_person_viewer, and no extra data needed to create quotation. When building graphs to represent the full Fiia conceptual model, numerous additional edges and vertices are used.

The formal definition of these edits uses set operations to manipulate the graphs. As outlined earlier, two edges are considered equal when their source, label and
CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING

target are, respectively, identical identifiers. As a result, the set operations behave as commonly defined for more complex graph structures. For an arbitrary pair of graphs $G$ and $H$:

$\Delta^- = G - H$ \hspace{1cm} \text{edges in G but not in H}

$\Delta^+ = H - G$ \hspace{1cm} \text{edges in H but not in G}

$\Delta^- \cap \Delta^+ = (G - H) \cap (H - G) = \emptyset$ \hspace{1cm} \text{sets are disjoint}

$H = (G - \Delta^-) \cup \Delta^+$ \hspace{1cm} \text{apply the edit}

We use this simple form of edits both to manipulate graphs and to reason about past manipulations.

5.2 Pattern Matching

To programmatically query and edit a graph, the first challenge is finding the interesting parts of the graph. These searches can be programmed imperatively in a traditional language, for example by iterating through the graph and checking each vertex. However, writing non-trivial searches by hand is difficult and error-prone. To simplify the task of specifying and performing searches, many programming languages and tools provide special-purpose pattern languages [16]. These pattern languages compactly specify a pattern in some target data. Some typical examples are regular expressions used to match text, and XPath with XML or HTML. As this thesis deals with transforming graphs, a pattern language based in our graph formalism is extremely useful.

Let a pattern consist of a graph, called the pattern graph, and a map over the variables. We will represent a pattern $p$ as a structure of $p$.edges, containing the
pattern graph, and $p$.vars, containing the variable map. Figure 5.3 shows a pattern which will match part of a simplified Fiia diagram: a conceptual component ($c$), with its type ($t$) and setting ($s$), and some node ($n$) in the same setting. Each variable (here $s$, $n$, $c$ and $t$) is a vertex in the pattern graph. The variable map associates each vertex with the set of vertices that it could match in the input graph. We use $U$ to indicate the universal set, for variables that could match any vertex. For Figure 5.3, this variable map consists of:

$$\text{vars} = \{ s \mapsto U, n \mapsto U, c \mapsto U, t \mapsto \{\text{Store, Reactor, Actor}\}\}$$

Variables $s$, $n$ and $c$ are unrestricted, so could match any vertex. $t$, however, can match only the exact identifiers Store, Reactor or Actor.

Searching for a pattern in an input graph (such as Figure 5.4) consists of finding possible embeddings. An embedding is an injective mapping from the variables of the pattern graph to vertices in the input graph, consistent with the pattern’s variable
map. Applying the embedding to the pattern graph must yield a subgraph of the input graph. Searching for Figure 5.3 in Figure 5.4 finds two embeddings:

\[
\{ \text{s} \mapsto \text{Clives.Office}, \text{n} \mapsto \text{Clives_PC}, \text{c} \mapsto \text{floor.plan}, \text{t} \mapsto \text{Store} \}, \quad \text{and} \\
\{ \text{s} \mapsto \text{Clives.Office}, \text{n} \mapsto \text{Clives_PC}, \text{c} \mapsto \text{first_person_viewer}, \text{t} \mapsto \text{Actor} \}
\]

In each embedding, the four variables are bound to distinct vertices of the input graph. If we apply these mappings to the pattern in Figure 5.3, each yields a subgraph of Figure 5.4. More precisely, an embedding \( e \) from pattern graph \( p \).edges with variable map \( p \).vars to input graph \( G \) is a mapping such that:

\[
\begin{align*}
\text{dom}(p \text{.vars}) &= \text{dom}(e) & \text{all variables are bound} \\
\forall i \in \text{dom}(p \text{.vars}) \cdot e(i) & \in p \text{.vars}(i) & \text{variable restrictions are respected} \\
\forall (s,l,t) \in p \text{.edges} \cdot \langle e(s), e(l), e(t) \rangle & \in G & \text{mapped pattern appears in graph}
\end{align*}
\]

Many different combinations of algorithms and data structures have been used in the literature to find similar embeddings. Other approaches to updating rewriting transformations typically require that these searches produce repeatable results. We require only that the search find some embedding, if any exist. The choice of a particular embedding among multiple possibilities may be arbitrary, and need not be consistent if the search is repeated. This permits our implementation of pattern matching in Fiia.Net to use fast techniques, such as hash tables, that cause stability problems in other graph transformation techniques.

The pattern language introduced in this section omits many common features, such as optional or repeated sections, forbidden sections and programmed constraints. These features allow complex patterns to be expressed more tersely, and for some similar patterns to be combined. Rather than complicate the pattern language and
CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING

theory, we later provide several of these features by extending our transformation rules and algorithm. These extensions are discussed in Chapter 7, after our basic theory and core trace-based update algorithm are completely defined.

5.3 Transformation Rules

Once part of the graph is selected by a pattern, the software can proceed to manipulate it. As with the graph patterns themselves, these operations can be programmed imperatively but are often easier and safer in a special-purpose transformation language. These transformation languages may be combined with the pattern language, as with TXL [33], or defined separately, like XSLT and XPath. We take the first approach, of integrating the pattern and replacement into a single rule definition. The transformation language used in this work again builds upon our set theoretic graph formalism. A transformation rule defines a pattern of changes to the graph, based on three sets of edges:

- **consumes**: must exist, will be deleted;
- **requires**: must exist, will not be modified; and
- **produces**: must not exist, will be added.

These sets describe both the prerequisites of a rule and its effects. In addition each rule includes a variable mapping, similar to that of a pattern. We will represent each of these components as a field of the rule, respectively, $r$.consumes, $r$.requires, $r$.produces and $r$.vars. Figure 5.5 shows a rule that attaches a component to some node in its setting. The rule removes the “in” edge connecting the component to its
setting, and replaces it with an “anchors” edge from a node to that component. This behavior can be specified equivalently using the visual notation shown in Figure 5.5a or the textual notation shown in Section 5.5b.

In general, a rule \( r \) is said to match if its prerequisite pattern has an embedding in the input graph \( G \). The prerequisite pattern is the union of the consumes and requires edges, together with the variables appearing therein. More precisely, the prerequisite pattern \( p \) of rule \( r \) is:

\[
p.\text{edges} = r.\text{consumes} \cup r.\text{requires} \quad \text{union of consumes and requires edges}
\]

\[
p.\text{vars} = \{ i \mapsto r.\text{vars}(i) \mid i \in \text{dom}(r.\text{vars}), \exists(s,l,t) \in p.\text{edges} \cdot (i = s \lor i = t) \} \quad \text{all variables from the rule, which appear in the pattern}
\]

**Figure 5.5:** A graph transformation rule in visual and textual notations.

\[
r.\text{vars} = \{ s \mapsto \text{U}, n \mapsto \text{U}, c \mapsto \text{U}, t \mapsto \{\text{Store, Reactor, Actor}\} \}
\]
CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING

function matchRule\((G, r, e = \emptyset) \rightarrow (e)\)

inputs
\n\begin{itemize}
\item \(G\) : input graph
\item \(r\) : rule to match
\end{itemize}

Derive the prerequisite pattern of the rule.
\[
p.\text{edges} = r.\text{consumes} \cup r.\text{requires}
\]
\[
p.\text{vars} = \{ \langle i \mapsto r.\text{vars}(i) \mid i \in \text{dom}(r.\text{vars}), \exists (s, l, t) \in p.\text{edges} \cdot (i = s \lor i = t) \rangle \}
\]

Find the pattern’s embedding as described in Section 5.2.
If no embedding is available, matchPattern will return \texttt{undef}.
\[
e = \text{matchPattern}(G, p)
\]
if not defined\(e\) then return \(e = \texttt{undef}\)

Expand the variable mapping to include produced variables.
\[
\text{for } i \text{ in } r.\text{vars}
\]
\[
\text{if not defined}\(e(i)\)
\]
\[
e = e \cup (i \mapsto \text{a new unused identifier})
\]

outputs
\[
(e : \text{resulting embedding, or } \texttt{undef} \text{ if none was found})
\]

Figure 5.6: Attempt to match a rule in a graph.

Embeddings are found normally for this pattern, as described in Section 5.2. If an embedding is available, then the rule matches. However, the embedding of the prerequisite pattern may not contain all of the variables of the rule. This is the case for variables that only appear in the produces edges. These unbound variables are used to create new unique vertices, and are common in the Fiua refinement rules. To complete the match, we must expand the embedding to map these unbound variables to new unused identifiers. This expansion is performed as part of our process for matching rules in Figure 5.6.
Once the rule’s embedding is available, the rule can be applied to modify the graph. This is performed in two parts, using the applyRule function in Figure 5.7: first, we create a step from the rule and embedding; second, we apply the step to the graph. The step captures the dependencies and effects of a rule application, so that we can reason about them later. We represent the step as a structure with three fields: edges which were consumed by the rule, edges which were required by the rule, and edges which were produced by the rule. These fields differ from a rule because they do not contain variables. Instead, the step describes the prerequisites and effects of a
particular rule and embedding. As a result, as we shall see in the next sections, the steps can be preserved for later analysis and used to repeat a previous transformation. In applying the rule, we modify the graph by removing consumed edges and adding produced edges, similar to the graph edits introduced in Section 5.1.

For example, the prerequisite pattern of Figure 5.5 was shown earlier in Figure 5.3. Using its first embedding in the example graph yields the step and resulting graph shown in Figure 5.8.

In effect, each rewriting rule defines a specific kind of change to the graph. The example used above is simplified from a Fiia refinement rule, but captures the concept of identifying the “interesting” part of the graph and making local modifications to it. The next sections show how multiple rules are used together, and how their individual steps are recorded for use in our trace-based update algorithm.

![Generated step](a) Generated step

![Resulting graph](b) Resulting graph

Figure 5.8: An example step and the resulting graph. Resulting graph. The floor plan’s “in” edge is consumed and replaced by an “anchors” edge from Clive’s PC.
5.4 Forward Transformation

The power of a single transformation rule is quite limited. Repeatedly applying multiple rules, however, produces very powerful transformations. This is a result of two key properties: first, each rule can apply multiple times in different regions of the graph; second, the output of one rule application can serve as part of the input for another. This allows rules to cooperate by building overlapping groups and chains of dependent steps. These combinations can express complex operations and hierarchical trees of decisions, like those used in implementing distributed systems.

One of the major variations between graph transformation techniques is how they choose among multiple matching rules and embeddings. The approach we use, and require, is unordered graph rewriting, in which one of the possibilities is chosen arbitrarily. Unordered rewriting is easy to implement, often by repeatedly looping through the rules and applying the first match. This simplicity combines well with our trace, as all dependencies between the rules are recorded in the steps. However, for this simple approach to always produce a valid result, several properties need to be considered when designing the meta-models and transformation rules:

**unordered**: Any matching rule may be applied to any embedding. Many tools use explicit ordering or structuring to make execution more predictable to rule authors. The rules can use intermediate elements to provide similar behavior, by producing constructs which are later consumed.

**non-deterministic**: Multiple executions of the transformation may choose different matching rules and embedding, even if the input graph is unchanged. This
greatly simplifies implementation because many techniques, such as hash tables, and UUIDs, introduce order-dependent or random effects.

**terminating** : Repeatedly applying matching rules must eventually produce a graph in which no rules match, for any finite input graph. This property ensures that unordered rule selection can never produce an infinite loop.

**complete** : When the transformation terminates, it will always result in a distribution graph. This property depends heavily on the definition of the conceptual and distribution meta-models.

Together these requirements ensure that the transformation will eventually terminate in a distribution graph, regardless of the order in which rules and embeddings are found. Phillips demonstrated that Fiia satisfies the same requirements [104], so can be used unchanged by our transformation engine.

Our process for applying a transformation is shown in Figure 5.9. The function produces two outputs: a completely transformed distribution graph and a trace of the transformation. We will discuss the trace in more detail in the next section.

Because the transformation is unordered and non-deterministic, different executions of **transform** with the same conceptual graph could produce different distribution graphs. We will refer to these varying distribution graphs as corresponding to the conceptual graph. Typical implementations of **transform** will impose some determinism on the transformation. For example, Fiia.Net iterates through the rules repeatedly, applying all matches found for each. This variation does not affect what graphs correspond, but merely which of the corresponding distributions can be produced by a particular implementation.
5.5 Transformation Tracing

Applying a rule to a graph, as described earlier, creates a step for each rule application. As multiple rule applications are performed, preserving their steps as a sequence yields the trace of the transformation. Each step records its prerequisites and effects on the graph, so the explicit trace can support many interesting operations. Traditionally, informal traces have been used to help developers debug transformations and occasional as part of rule matches [35]. Integrating steps, and hence the trace, into our graph rewriting formalism makes it easy to reason about different traces and graphs. This foundation is used in the next chapters to argue the correctness of our trace-based update algorithm.

```
function transform(C) → (D, T)

inputs
  C : conceptual graph

G = C
T = empty trace
while any rules match G
  r = an arbitrary one of the matching rules
  e = matchRule(G, r)

  # Apply the rule to the graph
  (G,s) = applyRule(G, r, e)
  T.append(s)

D = G

outputs
  D : distribution graph
  T : trace of steps
```

Figure 5.9: Transform a graph.
**CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING**

```latex
function applyTrace(G, T) \to (G)

inputs
\begin{align*}
C & : \text{starting conceptual graph} \\
T & : \text{existing trace}
\end{align*}

G = C

for \( s \) in \( T \)
\begin{itemize}
\item \# Ensure all prerequisites are available
\item assert \( s.\text{consumed} \subseteq G \)
\item assert \( s.\text{required} \subseteq G \)
\end{itemize}

\begin{align*}
\# \text{Apply the step to the graph} \\
G &= (G - s.\text{consumed}) \cup s.\text{produced}
\end{align*}

D = G

outputs
\begin{align*}
D & : \text{resulting distribution graph}
\end{align*}
```

Figure 5.10: Apply an existing to a graph.

Because a trace generated by a transformation contains the prerequisites and effects of every step, it can be used to re-apply the transformation to the same conceptual graph. This process, shown in Figure 5.10, applies the steps in the same order as they were generated. In addition, it checks that the prerequisites of each step are satisfied. This re-application is very fast compared to a normal transformation, because no repeated pattern matching is necessary. Instead, a simple implementation can apply the trace in \( O(e \log(n)) \) where \( e \) is the total number of edges consumed and produced, and \( n \) is the number of edges in the graph. During our evaluation in Chapter 9, we found that re-applying 8000 steps less than 20ms, as compared to nearly 5 seconds for the normal transformation.
We describe a trace as *valid* for a conceptual graph if it can be applied with all prerequisites satisfied. Two cases of valid traces are particularly common: first, any trace which corresponds to a conceptual graph will be valid for it; second, any prefix of the corresponding trace will also be valid for that conceptual graph. This can be seen in the *transform* function: while the trace $T$ is built incrementally, it is always valid for the input conceptual graph $C$.

A trace is also *complete* for a conceptual graph if applying it yields a distribution graph. That is, no rules match the resulting graph, so the trace could have been produced by completely executing *transform* starting from that conceptual graph.

Consider the artificial example trace shown in Figure 5.11a. The conceptual graph $C$ is transformed to $D$ in two steps: $T = s_1; s_2$. This shows the actual ordering of steps, but does not define the relationships between them. Incorporating the consumed, required and produced edges yields the dependency graph in Figure 5.11b. It shows that $s_1$ consumes $e_a$ (solid incoming arrow), requires $e_b$ (dashed incoming arrow) and produces $e_c$ (solid outgoing arrow); Meanwhile, $s_2$ consumes $e_b$ and produces $e_d$. Because $s_1$ requires an edge which is consumed by $s_2$ it must occur first, so only the $s_1; s_2$ order is possible.
CHAPTER 5. FRAMEWORK FOR GRAPH REWRITING AND TRACING

This simple example shows several different combinations of validity and completeness. The existing trace $s_1; s_2$ is both valid and complete for $C$, because it satisfies prerequisites and yields a distribution graph. A trace consisting of only $s_1$ would be valid, because it is a prefix, but not complete. A trace consisting of only $s_2$ would also be valid, because its prerequisites are satisfied. However, this trace might be complete: by consuming $e_b$, $s_2$ prevents the match that created $s_1$; unless another rule matches the product applying $s_2$ to $C$ will yield a distribution graph.

Our functions use several methods to modify a trace $t$: an individual step $s$ is appended to the trace by “$t.append(s)$” (as used in transform), removed with “$t.remove(s)$”, or replaced with $s'$ by “$t.replace(s,s')$”. In addition, entire traces are concatenated via “$t_1 + t_2$”. These manipulations can easily produce traces which are not valid or complete for any conceptual graph. However, as we shall see in the next chapters, their judicious use can modify a trace to be both valid and complete for new conceptual and distribution graphs.

5.6 Summary

In this chapter we have defined a new formalism for unordered graph rewriting. This formalism is particularly useful because it includes a complete trace of the transformation. This trace can be used to quickly re-execute a transformation, or to formally examine and modify transformations. These capabilities are used through the next chapters to support our novel trace-based update algorithm.
Chapter 6

Core Trace-Based Update Algorithm

In the earlier chapters of this thesis we have reviewed the challenges of developing adaptive distributed systems, and proposed a solution using model transformation at runtime. Our approach promises the valuable combination of a high-level programming model with extensive support for runtime adaptation, optimization and failure handling. However, our approach demands a model transformation that can express the complex decisions involved in implementing a distributed system, and rapidly update them at runtime as the models change. Existing model transformation techniques require a compromise between expressiveness and update speed, which is unsatisfactory for adaptive distributed systems. The core contribution of this thesis is our trace-based update algorithm, which satisfies the requirements of model transformation at runtime, and can support our vision for an adaptive distributed systems toolkit.
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

Our approach to adaptive distributed systems describes the system using two complementary runtime models: the conceptual model defines a high-level programmer-centric view; and the distribution model specifies a concrete implementation. The conceptual model is automatically refined into a distribution model, using a model transformation and set of refinement rules. While the models change at runtime to describe adaptations, our trace-based update algorithm keeps them consistent. That is, it ensures that the models are always related by a transformation following the refinement rules. Much of the challenge of this model transformation at runtime is performing this update quickly.

Chapter 5 presented a formalization of graphs and transformation that allow us to reason about the preconditions and effects of an existing transformation. In this chapter, we build from those foundations to introduce and define our novel algorithm for trace-based update. We demonstrate our algorithm applied to adaptive distributed systems, using the Fiia models and refinement rules. However, our algorithm does not depend on this application and can equally be used with to other meta-models and refinements. We will describe several other interesting applications of our algorithm in Chapter 10.

We present our core algorithm in several parts. First, Section 6.1 describes the relationship between conceptual model and distribution model, and the initial generation of a trace. Section 6.2 shows how additions to the conceptual model are propagated through the transformation using the recorded trace. This is then expanded to conceptual removals in Section 6.3 and general conceptual updates in Section 6.4. Finally, we show how distribution removals affect the models and trace in Section 6.5.


6.1 Refinement

The relationship between conceptual and distribution model is specified by the set of refinement rules. These models are represented as graphs, and transformed using the unordered rewriting engine described in Section 5.4. We refer to this transformation as a refinement to distinguish it from other model and graph transformations, such as the updates caused by adaptation.

Our algorithm performs a refinement in two situations. First, it is used to create an initial distribution graph $D$ and trace $T$ from a conceptual graph $C$ (Figure 6.1a). Here the starting graph is purely conceptual elements, and the initial trace is empty. Second, refinement is used by the functions presented later to complete refinement of an intermediate graph (Figure 6.1b). In this case the starting graph $H$ and initial trace $T_{old}$ represent a valid partial refinement. That is, $H$ and $T_{old}$ have values that
could occur if a refinement were paused part-way through. The refinement then “picks up where it left off”, and adds steps $T_{\text{add}}$ to yield a complete refinement $T_{\text{new}}$.

In a typical rewriting model transformation techniques, like those introduced in Chapter 4, conceptual changes are applied by refining the new graph. This is supported by our earlier formalism, but is slower and less stable than required by our motivation in adaptive distributed systems. In Chapter 9 we compare full refinement against our trace-based update algorithm, and find that even our prototype offers a ten-fold speedup. This difference is because normal refinement requires repeated pattern matching on the graphs, which is much slower than the set and list operations used to manipulate a trace. As we can see from the cases above, refining the intermediate graph $H$ avoids re-running all of the pattern matches required to build $T_{\text{old}}$.

The functions introduced through the remainder of this chapter describe how to efficiently apply changes, by updating the existing refinement. This approach avoids the drawbacks of complete re-refinement, but makes the correctness of the transformation less obvious. For each type of update, we will argue that the resulting trace is both valid and complete, so the produced conceptual and distribution graphs correspond.

### 6.2 Conceptual Addition

The first form of update we address is adding edges to the conceptual graph. Such adaptations typically occur when new nodes or components appear in a distributed system, such as when Sally connects to Clive’s floor plan viewer in Section 2.2.

Our approach to fast handling of conceptual additions relies on the trace generated by the previous refinement. As shown in Figure 6.2, this initial trace $T_1$ relates the
current conceptual $C_1$ and distribution $D_1$ graphs. Section 5.5 explained how this trace can be applied to $C_1$ to recreate $D_1$.

Now consider $C_2$, produced by adding edges $\Delta^+_C$ to $C_1$. These added edges will not preclude any pattern matches, so can not prevent rules which matched $C_1$ from matching $C_2$. Therefore, the trace $T_1$ must also be valid for $C_2$. That is, refining $C_2$ could happen to begin with exactly the steps in $T_1$. Rather than refining from the conceptual model, we can simply preserve $T_1$, and apply it to the new conceptual model.

Applying $T_1$ to $C_2$ yields an intermediate graph $H$, which may not be completely refined (due to the added edges). Refining this intermediate graph yields an additional trace $T_H$ and the updated distribution $D_2$. The final trace $T_2$ is the concatenation of $T_1$ and $T_H$. Because we know that $T_1$ is valid for $C_2$ and $T_H$ is valid and complete for $H$, their concatenation must be valid and complete for $C_2$.

For example, consider the dependency graphs in Figure 6.3 show a simple artificial case of conceptual addition. The initial trace $T_1$ (Figure 6.3a) consists of a single step $s_1$, consuming $e_a$ and producing $e_c$. The update to $C_2$ adds a new conceptual edge.
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

Figure 6.3: Example of an addition to the conceptual graph.

$e_b$, shown in Figure 6.3b. Applying $T_1$ to $C_2$ replaces $e_a$ with $e_c$, but leaves the new $e_b$ unchanged. Refining that intermediate graph $H$ then adds a new step $s_2$, which consumes $e_b$ and produces $e_d$. The final trace $T_2 = s_1; s_2$ includes both steps (Figure 6.3c), so yields a distribution graph containing $e_c$ and $e_d$.

The procedure used to update the trace following a conceptual addition is shown in Figure 6.4. Applying the existing trace uses only simple set operations, so executes far more quickly than the equivalent refinement. We then refine the intermediate graph normally. This approach of reusing the trace provides two important effects: First, performance is better than full re-refinement because fewer rule matches and applications are needed. Second, stability is guaranteed, because the existing selection of rules and embeddings is preserved unchanged.

As described in Chapter 5, any change to a graph can be performed by removing one set of edges and adding another. The process described in this section provides the addition half of that operation. In the next section, we describe our procedure for updating after a removal.

89
function \texttt{addConc}(C_1, T_1, \Delta^+_C) \rightarrow (T_2, D_2)

inputs
\begin{itemize}
    \item $C_1$: previous conceptual graph
    \item $T_1$: previous trace
    \item $\Delta^+_C$: conceptual edges added by adaptation
\end{itemize}

# Apply the change to the conceptual graph
$C_2 = C_1 \cup \Delta^+_C$

# Apply the existing trace to the new graph
$G = C_2$
for $s$ in $T_1$
    $G = (G - s\.consumed) \cup s\.produced$

# Refine the intermediate graph to a distribution.
$(G, T_H) = \texttt{transform}(G)$
$T_2 = T_1 + T_H$
$D_2 = G$

outputs
\begin{itemize}
    \item $T_2$: updated trace
    \item $D_2$: updated distribution graph
\end{itemize}

Figure 6.4: Update the refinement after an addition to the conceptual graph.
6.3 Conceptual Removal

The second form of update we address is removing edges from the conceptual graph. These are typically caused by adaptations which delete components and connectors, such as Section 2.4 in our furniture layout scenario, when Clive leaves the collaboration with Sally to confirm with Bill. Conceptual removals can also be caused indirectly, by removals from the distribution graph, as described later in Section 6.5.

As in Section 6.2, our approach reuses the trace generated by the previous refinement. There we were able to show that adding edges to the conceptual model must leave the old trace valid, and so it could be applied unchanged. In this case, however, removing edges from the conceptual model will often make the old trace invalid. Rather than reusing the old trace unchanged, we must now modify it to also be valid for the new conceptual model.

Section 5.5 provides a definition of validity based on the prerequisites of every step being satisfied. Removing edges from the conceptual model will often violate the prerequisites of some of the steps in the trace, leaving the entire trace invalid for the new conceptual model. This follows from the basis of a valid trace as a possible
refinement of the conceptual model: if prerequisites are missing, some of the steps could not have occurred during a normal refinement. However, if we prune these impossible steps, the trace can be made valid for the new conceptual model.

Figure 6.5 shows the flow of this approach. First, the existing trace $T_1$ is pruned to remove all impossible steps, based on the actual conceptual removal $\Delta \tilde{C}$. The pruned trace $T_1^-$ is now valid for $C_2$, because any impossible steps have been removed. However, $T_1^-$ is not necessarily complete, so the intermediate graph $H$ it yields is refined. The final trace is then the concatenation of $T_1^-$ and $T_H$.

The artificial case in Figure 6.6 shows the removal of one edge ($e_b$) from a simple conceptual graph. In the original trace, $s_2$ consumes $e_b$. With $e_b$ removed, $s_2$ is impossible, so must be discarded. Discarding $s_2$ removes $e_d$, which is required by $s_3$. Therefore, $s_3$ is also impossible, and must itself be discarded. This, in turn, removes $e_e$. The pruned trace of $s_1$ is valid for $C_2$, but may not be complete. Refining the intermediate graph $H$ adds a new step $s_4$ which consumes $e_c$. 

Figure 6.6: Example of a removal from the conceptual graph.
The key part of this procedure is identifying and removing impossible parts of the trace. To keep the final refinement fast, and preserve stability, this pruning should be kept to a minimum. The delConc function (Figure 6.7) does this by maintaining a $\Delta^-$ set of the edges that are now unavailable – either due to the update or because of an impossible step. If a step consumes or requires one of these unavailable edges, that step is pruned and its produced edges added to the $\Delta^-$ set. As a result, no steps remain in $T_2$ which are not possible starting with $C_2$, so $T_2$ is valid for $C_2$. The final refinement then yields a complete and valid trace $T_2$ and corresponding distribution graph $D_2$.

The processing time required to apply a conceptual removal depends on both the scope of the change and the refinement rules. In our application to adaptive distributed systems, and particularly in groupware, most conceptual changes are small and local. Many are as simple as that mentioned in our scenario as Clive moves the quotation viewer from his PC to PDA (Section 2.3), which involves removing only 6 of 96 edges. Larger removals are rarer, and tend to occur when users are changing their interactions, such as when Clive disconnects from Sally (Section 2.4), taking 42 of the 96 edges. In both cases the intermediate graph ($H$) is almost completely refined, so the computation is dominated by the pruning. The prune consists of set operations between the graph $G$ and small-constant-sized sets, it will scale at $O(m \log(n))$, where $m$ is the number of steps in the trace and $n$ is the number of edges in the graph.

Fiaa was deliberately designed such that most refinement rules are small and local. However, in addition to our common cases, it is worth considering the effects of larger and more global rules. Imagine a degenerate set of refinement rules, in which each
function \textbf{delConc}(C_1, T_1, \Delta_C^-) \rightarrow (T_2, D_2)

inputs
\begin{itemize}
    \item $C_1$: previous conceptual graph
    \item $T_1$: previous trace
    \item $\Delta_C^-$: conceptual edges removed by adaptation
\end{itemize}

# Apply the change to the conceptual graph
\begin{align*}
G &= C_1 - \Delta_C^- \\
T^- &= T_1 \\
\Delta^- &= \Delta_C^-
\end{align*}

# Prune the trace to be valid for the new graph
for $s$ in $T_1$
\begin{itemize}
    \item if $\Delta^- \cap (s\.consumed \cap s\.required) \neq \emptyset$
        \begin{itemize}
            \item This step depends on edges that are no longer available.
            \item $T^-\.remove(s)$
        \end{itemize}
        \begin{itemize}
            \item Mark edges produced by this step as unavailable.
            \item $\Delta^- = \Delta^- \cup s\.produced$
        \end{itemize}
    \item else
        \begin{itemize}
            \item This step is still possible, so apply it.
            \item $G = (G - s\.consumed) \cup s\.produced$
        \end{itemize}
\end{itemize}

# Refine the intermediate graph to a distribution.
$(G, T_H) = \textbf{transform}(G)$
$T_2 = T^- + T_H$
$D_2 = G$

outputs
\begin{itemize}
    \item $T_2$: updated trace
    \item $D_2$: updated distribution graph
\end{itemize}

Figure 6.7: Update the refinement after a removal from the conceptual graph.
step consumes the entire graph and produces a new graph. Our dependency analysis will determine that any removal must prune every step in the trace, so will always require a full re-refinement. A more realistic set of refinement rules could encounter a similar problem by having randomly-distributed prerequisites and products. In that case, even a small local removal would cascade out to prune most of the trace. Such combinations are a weakness of our current trace-based update algorithm. However, we observe that even in these worst-cases the conceptual removal degrades to only the $O(m \log(n))$ prune followed by a full refinement. So problematic refinement rules or conceptual changes can, at worst, discard our performance and stability advantages over a full refinement. These are clearly conditions to be avoided, but do not affect the correctness of the update. As a result, these problematic cases need only be minimized by optimizing the rules and meta-models for trace-based update, rather than avoided entirely.

This process provides the second half of general conceptual updates. By performing a removal followed by an addition, our model transformation engine can express any change to the conceptual graph. In the next section we merge these two steps into a single $\text{updateConc}$ routine.

### 6.4 Conceptual Update

Any change in a graph can be expressed by the combination of a removal and an addition, as introduced in Chapter 5. Many conceptual adaptations will require both; for example, renaming a component consists of removing the edge binding it to the old name, and adding an edge binding it to the new one. To avoid repeated work, we merge the $\text{delConc}$ and $\text{addConc}$ functions into $\text{updateConc}$ in Figure 6.8.
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

function updateConc\((C_1, T_1, \Delta^-_C, \Delta^+_C) \rightarrow (T_2, D_2)\)

inputs
- \(C_1\): previous conceptual graph
- \(T_1\): previous trace
- \(\Delta^-_C\): conceptual edges removed by change
- \(\Delta^+_C\): conceptual edges added by change

# Apply the change to the conceptual graph
\(G = (C_1 - \Delta^-_C) \cup \Delta^+_C\)
\(T = T_1\)
\(\Delta^- = \Delta^-\)

for \(s\) in \(T_1\)
  if \(\Delta^- \cap (s\text{.consumed} \cup s\text{.required}) \neq \emptyset\)
    # This step depends on edges that are no longer available.
    \(T\text{.remove}(s)\)
    # Mark edges produced by this step as unavailable.
    \(\Delta^- = \Delta^- \cup s\text{.produced}\)
  else # This step is still possible.
    \(G = (G - s\text{.consumed}) \cup s\text{.produced}\)

# Refine the intermediate graph to a distribution.
\((G, T_H) = \text{transform}(G)\)
\(T_2 = T + T_H\)
\(D_2 = G\)

outputs
- \(T_2\): updated trace
- \(D_2\): updated distribution graph

Figure 6.8: Update the refinement after a change to the conceptual graph.
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

This function is the first part of our core algorithm. By exploiting trace-based update, it offers the expressiveness of flexibility graph rewriting, along with rapid and stable update. This combination allows us, for the first time, to use model transformation at runtime in adaptive distributed systems.

The arbitrary conceptual changes that can be applied via \texttt{updateConc} support most of the requirements of adaptive distributed systems. Indeed, based on the requirements in Section 1.2 it lacks only support for optimization and distribution adaptations. We will expand trace-based update to support distribution changes in the next section. Chapter 7 then extends the core algorithm to support optimization and further enhance expressiveness.

6.5 Distribution Removal

Partial failures in distributed systems are particularly hard to resolve in the presence of a high-level programming model. If failures are exposed to the application in their low-level form, for example that the TCP/IP socket to 192.168.0.4:5308 was closed, it is difficult for developers to evaluate the scope of the failure and respond strategically. However, exposing failures in a high-level form requires converting them up from the low-level form. A two-level runtime model provides a natural way of capturing this behavior: failures cause removals in the distribution model, which are automatically propagated back to the conceptual model. This allows the developer to work exclusively within the conceptual model, rather than delving into implementation details to let the application diagnose and repair failures.

In our review of model transformation techniques (Chapter 4) we found only a single example of a rewriting transformation which could efficiently perform
distribution removals. There, Xiong et al. [141] showed how to invert a deterministic transformation written in ATL, and propagate distribution changes back to the conceptual model. We use our explicit trace to support a similar approach in our, more general, non-deterministic transformation. Other alternatives for handling distribution removals require slow searches to reverse-engineer the distribution change, or place severe restrictions on the permissible refinement rules. Our trace and focus on distribution removals (corresponding to partial failure) again allow us to provide fast and stable updates with general rewriting rules.

Our process for applying distribution removals follows the flow in Figure 6.9. We first derive a conceptual change $\Delta_C^-$ that is sufficient to cause the observed distribution removal $\Delta_D^-$, based on the contents of the existing trace $T_1$. This derived change is then applied using the earlier algorithm for conceptual removals. That application often produces additional distribution changes $\Delta_{D2}^-$, before arriving at the final conceptual graph $C_2$, distribution graph $D_2$ and trace $T_2$. 
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

An artificial example of our distribution removal process is shown in Figure 6.10. Starting in Figure 6.10a, a distribution failure occurs which removes distribution edge $e_f$. $e_f$ is produced in $s_4$, so will never exist if $s_4$ does not occur. $s_4$ consumes $e_d$, so will never occur if it does not exist. Continuing this path of prerequisites in Figure 6.10b identifies $e_b$ in the conceptual model. Therefore, removing $e_b$ from the conceptual model if sufficient to remove $e_f$ from the distribution. The conceptual change of removing $e_b$, in Figure 6.10c, propagates downward, following the process described earlier, and removes both $e_f$ and $e_e$. This collateral removal ensures that the pruned trace (only $s_1$) is valid for $C_2$. Final refinement adds $s_5$, yielding a new distribution edge $e_g$ in Figure 6.10d.

This update is significantly different from those driven by conceptual changes, because it will modify both the conceptual and distribution graphs. However, it carries the same requirement that the resulting trace must be valid and complete for the resulting conceptual graph. It also adds the requirement that the distribution
change be *respected*. That is, the removed distribution edges $\Delta_D^-$ must be reflected in the conceptual graph, rather than simply being restored by $\Delta_{P2}$.

We formalize *respected* to mean that at least one conceptual prerequisite of every removed distribution edge must itself be removed by $\Delta_C^-$. Where a removed edge was originally present in the conceptual graph, this means it must simply be removed from the conceptual graph. In the more common case where a removed edge was produced by a step, at least one conceptual prerequisite of that step must be removed. This recursive definition means, more intuitively, that the resulting refinement will not use the same steps to recreate the removed distribution edges.

The `delDist` function in Figure 6.11 performs a distribution removal in two steps: it first derives a conceptual removal $\Delta_C^-$, and then applies that conceptual removal using `updateConc`. As we have previously argued, `updateConc` will always produce a valid and complete trace. The correctness of `delDist`, therefore, hinges on the sufficiency of $\Delta_C^-$ to respect the requested distribution change. Our definition of *respected* requires that at least one conceptual prerequisite of the removed distribution edges be itself removed by $\Delta_C^-$. By propagating $\Delta^-$ backwards through the trace, `delDist`, in fact, removes all conceptual prerequisites of the removed distribution edges. This ensures a valid and complete result, but will often remove much more of the conceptual graph than strictly necessary. In Section 7.4 we extend our process with semantics of the meta-models and refinement rules, so that $\Delta_C^-$ is smaller and more meaningful to the developer.

The processing time required to apply a distribution removal is, again, dependent on both the scope of the change and the refinement rules. `Fiia` rules are small and local, limiting the number of conceptual prerequisites of each distribution edge to
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

function delDist\((C_1, T_1, D_1, \Delta_{\bar{D}}) \rightarrow (C_2, T_2, D_2)\)

inputs
- \(C_1\): previous conceptual graph
- \(T_1\): previous trace
- \(D_1\): previous distribution graph
- \(\Delta_{\bar{D}}\): distribution edges removed by adaptation

\(\Delta^- = \Delta_{\bar{D}}\)

for \(s\) in reverse \(T_1\)
  if \(\Delta^- \cap s.\text{produced} \neq \emptyset\)
    # This step produces an edge which must no longer appear.
    # Mark the prerequisites of this step to be removed.
    \(\Delta^- = \Delta^- \cup (s.\text{consumed} \cup s.\text{required})\)

    # Edges produced by this step will never be considered again.
    # Remove them to avoid bloating the \(\Delta^-\) set.
    \(\Delta^- = \Delta^- - s.\text{produced}\)

# The collected prerequisites are sufficient to cause the required
# distribution change.
\(\Delta^-_{\bar{C}} = \Delta^-\)
\(C_2 = C_1 - \Delta^-_{\bar{C}}\)

\((T_2, D_2) = \text{updateConc}(C_1, T_1, \emptyset, \Delta^-_{\bar{C}})\)

outputs
- \(C_2\): updated conceptual graph
- \(T_2\): updated trace
- \(D_2\): updated distribution graph

Figure 6.11: Update a refinement after a removal from the distribution graph.
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

a small constant. Further, the distribution removals are caused by partial failures, which are themselves local phenomena. As a result, the derived conceptual removal \( \Delta^- \) is itself typically small and local, so applied quickly by \texttt{updateConc}. In the rare cases where these assumptions do not hold, \( \Delta^- \) may grow to size of \( C_1 \) or \( D_1 \). This increases the runtime complexity of the derivation from \( O(m) \) to \( O(m \log(n)) \), where \( m \) is the number of steps in the trace and \( n \) is the maximum of the number of edges in \( C_1 \) and \( D_1 \). The resulting large \( \Delta^- \) mitigates this increase, because the refinement in \texttt{updateConc} will execute faster on the reduced graph.

This operation provides the first approach through which an unordered, non-deterministic, rewriting transformation can quickly apply distribution removals. It completes our core algorithm, and basic support for arbitrary conceptual changes and distribution removals.

6.6 Summary

The procedures presented in this chapter provide fast and stable updates following conceptual changes or distribution removals. This satisfies many of our earlier requirements for model transformation at runtime, allowing our adaptive distributed systems toolkit to combine a high-level programming model with extensive runtime adaptation.

In each case, the existing trace is exploited to avoid a full refinement. This provides a valuable performance improvement, because the graph pattern matching used during refinement is replaced with fast set operations. In addition, preserving parts of the trace reduces the risks of instability introduced by unordered, non-deterministic
CHAPTER 6. CORE TRACE-BASED UPDATE ALGORITHM

rules. As a result, our trace-based update algorithm provides novel capabilities to the extremely powerful family of unordered rewriting model transformations.

In spite of their improved performance and stability, we have explained how each update routine produces models and trace equivalent to a possible refinement. This ensures that our algorithm can be used safely for any meta-models and refinement rules satisfying the requirements of Chapter 5. In addition, we have explained several optional criteria which affect the performance and stability of the updates.

Our core trace-based update algorithm provides several novel capabilities, but still falls somewhat short of a complete solution for model transformation at runtime. First, the rule syntax excludes several common features used to simplify transformations (forbidden preconditions and repeated sections). Second, as we observed in Section 6.5, distribution removals tend to cause excessively large conceptual removals. Third, we have not provided a method to specify optimizations or automatically guide the refinement toward good distributions. We address the first two of these shortcomings in Chapter 7 by extending the rule syntax and behavior of distribution removals. Much of our future work resolves around guiding the refinement to good solutions, but we present the existing techniques used by Fiia.Net in Chapter 8.
Chapter 7

Extended Trace-Based Update Algorithm

In the previous chapter we have presented our core algorithm for trace-based update. This approach provides unique support for model transformation at runtime, and our application to adaptive distributed systems. However, to simplify the discussion and functions, the core algorithm omits several features which are useful in applying trace-based update.

In this chapter we present several extensions to our core trace-based update algorithm. We first refactor the earlier functions for conceptual update and distribution removal, to ease support for other extensions. We then continue to address the shortcomings of our core algorithm noted at the close of Chapter 6.

We first present two extensions which enhance the expressiveness of our transformation rules. These sections present, respectively, support for forbidden preconditions and repeated sections. These two extensions make our transformation rules as expressive as those offered by other graph rewriting techniques, without sacrificing
the novel capabilities of trace-based update. As a result, the transformation engine used in Fiia.Net can execute the full Fiia refinement rules, without the simplifications used in our earlier examples.

Finally, we address the problem we described in Section 6.5 that distribution removals tend to discard too much of the conceptual model. We resolve this problem by augmenting the rules with some semantic information, to indicate which of the prerequisites should be removed. This reduces the impact of distribution changes, and makes the resulting conceptual changes follow the intuitive structure of the refinement.

For each of these extensions, we explain how our earlier theory and algorithms can be modified to support them. At the close of this chapter, we review the capabilities of our extended trace-based update algorithm from the perspective of developing adaptive distributed systems toolkits.

### 7.1 Refactoring

The extensions we introduce through the remainder of this chapter make numerous changes to our processes for resolving conceptual updates and distribution models. To simplify those changes, we first examine and refactor the existing functions for resolving conceptual updates and distribution removals.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

function updateConcStep\((G, \Delta^-, s) \rightarrow (G, s)\)

inputs
- \(G\) : previous graph
- \(s\) : step to be updated

\(r = s.\text{rule}\)

# Check whether this step depends on edges that are no longer available.
if \(\Delta^- \cap (s.\text{consumed} \cup s.\text{required}) \neq \emptyset\)
    return \(s = \text{undef}\)

\(G = (G - s.\text{consumed}) \cup s.\text{produced}\)

# If the step is no longer possible, this function will
# return an unchanged graph and \(s = \text{undef}\).
outputs
- \(G\) : updated graph
- \(s\) : updated step

Figure 7.1: Update a step as part of resolving a change in the conceptual graph.

7.1.1 Conceptual Update

First, consider the process used to apply conceptual updates, originally defined in Figure 6.8. The main loop walks through the trace, determining whether each step is still possible. To simplify the later extensions we partition this process into two functions: \texttt{updateConcStep} (Figure 7.1), which is responsible for checking and applying the steps; and \texttt{updateConc} (Figure 7.2), which performs the outer iteration and maintains the trace. In addition, we add code in \texttt{updateConc} to support modifying the steps, which will be needed in Section 7.3.
function `updateConc`(\(C_1, T_1, \Delta^-_C, \Delta^+_C\)) \(\rightarrow\) \((T_2, D_2)\)

**inputs**
- \(C_1\) : previous conceptual graph
- \(T_1\) : previous trace
- \(\Delta^-_C\) : conceptual edges removed by change
- \(\Delta^+_C\) : conceptual edges added by change

# Apply the change to the conceptual graph
\(G = (C_1 - \Delta^-_C) \cup \Delta^+_C\)
\(T = T_1\)

for \(s\) in \(T_1\)

  # new: Delegate checking and updating the step.
  \(s' = \text{updateConcStep}(G, \Delta^-, s)\)

  if not \(\text{defined}(s')\)
    # This step is no longer possible.
    \(T\).remove\(s\)
    \(\Delta^- = \Delta^- \cup s.\text{produced}\)

  else
    # This step is still possible.
    \(G = (G - s'.\text{consumed}) \cup s'.\text{produced}\)

    # new: If the step changed, update the trace and \(\Delta^-\) set.
    if \(s \neq s'\)
      \(T\).replace\(s, s'\)
      \(\Delta^- = \Delta^- \cup (s.\text{produced} - s'.\text{produced})\)
      \(\Delta^- = \Delta^- \cup (s'.\text{consumed} - s.\text{consumed})\)

# Refine the intermediate graph to a distribution.
\((G, T_H) = \text{transform}(G)\)
\(T_2 = T + T_H\)
\(D_2 = G\)

**outputs**
- \(T_2\) : updated trace
- \(D_2\) : updated distribution graph

Figure 7.2: Update the refinement following a change in the conceptual graph.
function delDistStep(Δ−, s) → (Δ−)

inputs
- Δ−: set of edges which are no longer available
- s: step to be checked

if Δ− ∩ s.produced ≠ ∅
    # This step produces an edge which must no longer appear.
    # Mark the prerequisites of this step to be removed.
    Δ− = Δ− ∪ (s.consumed ∪ s.required)

    # Edges produced by this step will never be considered again.
    # Remove them to avoid bloating the Δ− set.
    Δ− = Δ− − s.produced

outputs
- Δ−: updated set of available edges

Figure 7.3: Update a step as part of resolving a removal in the distribution graph.

7.1.2 Distribution Removal

Second, consider the process used to apply distribution removals, originally defined in Figure 6.11. The main loop walks backward up the trace, propagating the removal through the producer of each edge to its prerequisites. We partition this process into two functions: delDistStep (Figure 7.3), which is responsible for checking each step and marking its prerequisites; and delDist (Figure 7.4), which performs the outer iteration.

These enhanced functions will be further extended through the remainder of this chapter, to the actual procedures deployed in Fiia.Net.
function delDist(C_1, T_1, D_1, \Delta_D^-) \rightarrow (C_2, T_2, D_2)

inputs
C_1: previous conceptual graph
T_1: previous trace
D_1: previous distribution graph
\Delta_D^-: distribution edges removed by adaptation

# Derive a sufficient conceptual change.
\Delta^- = \Delta_D^-
for s in reverse T_1
  \Delta^- = delDistStep(\Delta^-, s)

# Apply the derived conceptual change.
\Delta_C^- = \Delta^- \cap C_1
C_2 = C_1 - \Delta_C^-
(T_2, D_2) = updateConc(C_1, T_1, \emptyset, \Delta_C^-)

outputs
C_2: updated conceptual graph
T_2: updated trace
D_2: updated distribution graph

Figure 7.4: Update the refinement following a removal in the distribution graph.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

7.2 Forbidden Preconditions

The patterns, rules and processes defined in the previous chapters support only positive matches. That is, they can only specify edges which must exist for a match to occur, rather than edges which must not. This simplifies our earlier theory and procedures, but limits the expressiveness of the transformation. Forbidden preconditions are often used in model transformations and common throughout the Fiia refinement rules. As a result, they are an important step toward our goal of supporting expressive transformation rules.

In this section we extend our earlier definitions to support forbidden, as well as positive, preconditions. We first turn to Chapter 5 and extend the definition of transformation rules and transformation tracing. With those features in place we augment our process for handling conceptual updates.

7.2.1 Transformation

In Section 5.3 we defined a language for expressing graph transformation rules as sets of edges and a map of variables. We add forbidden preconditions as a set of patterns associated with the rule. After the rule’s prerequisite pattern matches, each forbidden pattern is checked. If any forbidden pattern matches, the rule itself does not match.

The forbidden patterns inherit variables from their rule’s embedding. That is, any variable in the rule’s embedding will be substituted into the forbidden pattern. Conversely, variables defined by a forbidden pattern are not exposed to either the rule or other forbidden patterns.

For example, consider Figure 7.5, the Fiia refinement rule that “anchors” a conceptual component to a physical node. When the prerequisite pattern of this
full rule matches, the variable $c$ will be bound to some conceptual component in the graph. The transformation process will then search for the forbidden pattern using the same value of $c$. As a result, the forbidden pattern will match some Node $m$ which already anchors the component. The effect of this combination is that the rule will match some node $n$, and a conceptual component $c$ that is not anchored to any node. Notice that using $n$ instead of $m$ in the forbidden pattern would not work: rather than forbidding $c$ from being anchored to any node, it would only forbid $c$ from being already anchored to the particular $n$.

A modified variant of this rule was shown earlier in Figure 5.5. That variant consuming one of its prerequisite edges to avoiding repeated “anchors” without needing a forbidden precondition. It is similarly possible to design most rules to avoid forbidden precondition. However, writing the forbidden precondition explicitly
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

```plaintext
# Append to matchRule in Figure 5.6

# Check forbidden preconditions.
for p2 in r.forbids
    e2 = matchPattern(G, p2, e)
    if defined e2 then return e = undef

Figure 7.6: Attempt to match a rule with forbidden preconditions in a graph.
```

makes rules easier to under and less error-prone: Figure 5.5 implicitly requires that anchored conceptual components not be “in” any setting. With our new forbidden preconditions, Figure 7.5 simply states that the component must not be unanchored.

This expanded definition of transformation rules also requires a modification in the procedure used to match rules. After an embedding for the rule is found, the graph must be checked against the forbidden preconditions. This is performed by appending the check to matchRule, as shown in Figure 7.6.

The performance impact of forbidden preconditions on this procedure is, fortunately, minimal. matchRule performs a pattern match on the rule’s prerequisite pattern before considering the forbidden preconditions. As a result, the common case of a non-matching rule will return without ever trying to match the new forbidden patterns. Even when the new calls are executed, they are much faster than matching the rule’s prerequisite pattern. This is the case for two reasons: first, the forbidden patterns are typically small, involving only a few variables and edges; second, the variables inherited from the rule’s embedding drastically limit the search space.

With our transformation rules extended to support forbidden preconditions, we now press forward to define how the refinement is updated following a change to the conceptual graph.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

# Insert in updateConcStep (Figure 7.1)
# before \( G = (G - s.\text{consumed}) \cup s.\text{produced} \).

# Check forbidden preconditions.
for \( p \) in \( r.\text{forbids} \)
    \( e = \text{matchPattern}(G, p, e) \)
    if defined \( e \) then return \( s = \text{undef} \)

Figure 7.7: Update the refinement after a change to the conceptual graph.

7.2.2 Conceptual Update

Extending our transformation to support forbidden preconditions removes a simplifying assumption made while defining conceptual addition in Section 6.2. Previously, adding edges to a conceptual graph could only add new steps. Now, an added edge might trigger the forbidden precondition of a rule, and make an existing step impossible in the normal refinement. Fortunately, our procedure for conceptual update in Section 6.4 already deals with pruning impossible steps. This routine need only be modified to check forbidden preconditions as well as the earlier prerequisites.

Figure 7.7 shows our extended procedure for conceptual updates. In addition to checking the step’s prerequisites, this routine also iterates through the step’s forbidden preconditions. If any of these forbidden preconditions match, the “possible” variable is cleared and the step removed from the trace.

7.2.3 Summary

We have shown an extension of our earlier formalism and core trace-based update algorithm to support forbidden preconditions. This requires modifying the routines used to match and apply rules, as well as that responsible for conceptual updates.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

Distribution removals can never trigger a forbidden pattern except as part of their derived conceptual removal, so no change in that procedure is required.

These forbidden preconditions add a feature that is present in many transformation languages, and has proven useful in the Fiia rules. Because most variables in a forbidden pattern are inherited from the step’s embedding, searching for the forbidden patterns is very fast. In our experiments with Fiia.Net, shown in Chapter 9, these costs remain negligible even for large systems.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

7.3 Repeated Sections

Our second enhancement to the expressiveness of our transformation rules is to allow parts of the rule to be repeated. This makes it possible to express repeated operations without dividing them across multiple rules. These repeated sections behave like the “match zero-of-more” multiplicities common in regular expressions and other pattern languages.

In this section we further extend our formalism and procedures to support repeated sections. We first enhance our definition of transformation rules and the associated trace. We then turn to our procedures for applying conceptual updates and distribution removals.

7.3.1 Transformation

Our first step toward supporting repeated sections is to extend the rule specification. As with forbidden preconditions, repeated sections are nested within the rule. However, the repeated sections are nested child rules – with consumes, requires and produces edges – rather than merely patterns. When the parent rule is applied, each repeated section matches and applies as many times as possible.

All of the repeated sections must be independent. That is, within a particular application of the parent rule, an edge produced or consumed by one repeated section may not be referenced by another repeated section. Like forbidden patterns, repeated sections inherit variables from their parent rule’s embedding and keep their own private.

Figure 7.8 shows part of the Fiia rule used to split a conceptual component into its native implementation and a helper responsible for concurrency control and
consistency maintenance (a CCCM). The repeated section moves all incoming ports to the CCCM, while outgoing ones are left on the component.

When the parent rule matches, its embedding contains values for \( n, c, t \) and \( h \). The repeated section then searches for \( p \) and \( p_t \) which satisfy its own prerequisite pattern. “TargetPort”, used to restrict the value of \( p_t \) is shorthand for the set of Fiia port types which are considered incoming. Every time the repeated section matches, it is applied to move the port \( p \) from the component \( c \) to the CCCM \( h \).

Because repeated sections match zero or more times, they do not affect whether or not the parent rule will match. As a result, only \texttt{applyRule} needs to be modified to extend our transformation. Figure 7.9 shows the additional operations needed.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

# Append to applyRule in Figure 5.7

# Track the repeated sections in the step.
s.repeated = an empty list

# Apply repeated sections as many times as possible.
for r2 in r.repeats
    while true
        e2 = matchRule(G, r2, e)
        if not defined(e2) then break

        (G, s2) = applyRule(G, r2, e2)
        s.repeated.append(s2)

Figure 7.9: Apply a rule with repeated sections to a graph.

to match, apply and record repeated section. After creating the step normally, the repeated sections are matched and applied like rules, with the steps appended into a “repeated” sequence.

The “repeated” sequence is stored separately from the consumed, required and produced sets of the step. This simplifies the update functions to be presented shortly, but requires that we also revise the applyTrace function from Figure 5.10 to support the repeated sections. This revised function shown in Figure 7.10. Because the “repeated” sequence is in the same form as a trace – a list of steps – we can simply invoke applyTrace recursively on it.

The performance impact of repeated sections is surprisingly minimal. Moreover, repeated sections actually refine fast than the equivalent repeats constructed as multiple rules. This is the case because many variables repeated sections are inherited from the parent rule, so drastically limit the necessary search space. In addition,
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

function applyTrace\(G, T\) → \(G\)

inputs
  \(C\) : starting conceptual graph
  \(T\) : existing trace

\(G = C\)

for \(s \in T\)
  # Ensure all prerequisites are available
  assert \(s\).consumed ⊆ \(G\)
  assert \(s\).required ⊆ \(G\)

  # Apply the step to the graph
  \(G = (G \setminus s\).consumed) \cup s\).produced\)

  # new: Apply the repeated children to the graph
  \(G = applyTrace(G, s\).repeated\)

\(D = G\)

outputs
  \(D\) : resulting distribution graph

Figure 7.10: Apply a trace with repeated sections to a graph.

typical repeated sections, such used by the \(Fiia\) rules, are much smaller than normal rules.

From this extended transformation, we now continue to explain our processes for resolving conceptual updates and distribution removals in the presence of repeated sections.
7.3.2 Conceptual Update

A simple approach to implementing repeated sections would be to execute them only during forward refinement, and prune steps whenever their repeats should change. This is unsatisfactory for our purposes, because conceptual changes regularly affect the number of repeats in FiiA rules. For example, when an incoming port is added to or removed from a component, it will change the number of ports moved by the instantiate rule (Figure 7.8). Such a change happens whenever the FiiA.Net programming model creates or deletes a connection, so even our short scenario in Chapter 2 encounters more than 30 cases. Removing the entire instantiate step in all of these cases would cascade to prune numerous other refinement steps, sacrificing our performance and stability advantages. However, a more fine-grained approach can modify the existing steps to maintain our repeated sections.

To effectively support greedy repeated sections in our engine, we need to reexamine how conceptual updates are propagated through the trace. Previously, the trace steps introduced in Chapter 5 have been treated as immutable: once created by the applyRule function, a step was never altered until being discarded entirely. To deal efficiently with repeat sections, our procedures must now modify steps to reflect the changing numbers of repeats.

Maintaining the “repeated” list of a step is very similar to maintaining a full trace. However the repeated sections must be independent. That is, repeated sections within a rule can not have interdependencies. As a result removing one repeat will never invalidate others, unlike the cascades in trace steps during conceptual changes. Figure 7.11 shows the changes to updateConcStep needed to update the
# Append to updateConcStep in Figure 7.1

# Check for repeated sections which are now impossible.
for s2 in s.repeated
    s2' = updateConcStep(G, s2)
    if not defined(s2')
        s.repeated.remove(s2')
    else
        G = (G − s2'.consumed) ∪ s2'.produced
        s.repeated.replace(s2, s2')

# Apply repeated sections as many times as possible.
for r2 in r.repeats
    while true
        e2 = matchRule(G, r2, e)
        if not defined(e2) then break

        (G, s2) = applyRule(G, r2, e2)
        s.repeated.append(s2)

Figure 7.11: Update a step as part of resolving a change in the conceptual graph.

“repeated” list. It proceeds in two steps: removing any repeated sections which are now impossible; and then adding any newly-possible repeated sections.

7.3.3 Distribution Removal

The new “repeated” sequence in the steps also requires changes in the process of resolving distribution removals. Previously in Figure 7.3 only the step’s produced edges needed to be compared against the edges being removed. Now the step’s repeated sections must also be checked, as their produced edges may need to be removed (Figure 7.12). Because the entries in the “repeated” sequence are steps we can again use a recursive function call, this time to delDistStep.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

# Insert at the beginning of delDistStep in Figure 7.3
# Check whether any repeated sections must be removed.
for s2 in s.repeated
    \[ \Delta^- = \text{delDistStep}(\Delta^-, s_2) \]

Figure 7.12: Update a step as part of resolving a removal in the distribution graph.

7.3.4 Summary

In this section we have shown an extension of our earlier formalism and trace-based update algorithm to support repeated sections. This requires changes to transformation rules and trace, as well as the various routines responsible for applying conceptual updates and distribution removals.

Repeated sections add a feature that is present in many transformation languages, and used extensively by the Fiia rules. Because most variables in each repeated section are inherited from their parent rule’s embedding, search for and updating the repeated sections is surprisingly fast. In our experiments with Fiia.Net, shown in Chapter 9 these costs remain negligible even for large systems.
7.4 Crucial Preconditions

As we observed in Section 6.5, distribution removals using the core trace-based update algorithm tend to cascade to discard large portions of the conceptual graph. The cause of this problem is quite simple: while our definition requires removing \textit{at least one} conceptual prerequisite of each removed distribution edge, our core algorithm (in Figure 6.11 and Figure 7.4) removes \textit{all} conceptual prerequisites. This is correct for our formal purposes, but destroys an unnecessarily-large region of the distributed system. The core trace-based update algorithm will typically shut down every node remotely involved with the failed portion of the system.

This earlier approach was used because it is simple to introduce, and because choosing a good set of prerequisites requires knowledge about the meaning of the meta-models. The distribution removal should propagate back to remove conceptual edges in a way which is meaningful to the application developer. For example, if a TCP/IP network connection drops, that failure should appear to the developer as the associated connectors being disconnected. This determination can not be made completely automatic, so requires manual review of every refinement rule.

In this section we augment our refinement rules and algorithm to reduce the impact of distribution removals. We start by identifying the \textit{crucial} prerequisites of each rule. These edges represent the parts of the prerequisite that logically cause the rule to apply, from the perspective of a developer in the conceptual model.

In \texttt{Fiia}, many refinement rules center around consuming a single model element, and producing one or more low-level elements. For example, \texttt{Fiia}'s instantiate rule (used as an example in Figure 7.8) consumes a single conceptual component, and produces a distribution component with some supporting infrastructure. Treating
function \texttt{delDistStep}(\Delta^-, s) \rightarrow (\Delta^-)

\textbf{inputs}
\begin{itemize}
\item $\Delta^-$: set of edges which are no longer available
\item $s$: step to be checked
\end{itemize}

\textbf{outputs}
\begin{itemize}
\item $\Delta^-$: updated set of available edges
\end{itemize}

# Check whether any repeated sections must be removed.
\begin{verbatim}
for $s_2$ in $s$.repeated
    $\Delta^- = \texttt{delDistStep}(\Delta^-, s_2)$
\end{verbatim}

if $\Delta^- \cap s$.produced $\neq \emptyset$

    # This step produces an edge which must no longer appear.
    # \textbf{new}: Mark only the crucial prerequisites of this step to be removed.
    $e = s$.embedding
    \begin{align*}
    \text{let } e'(i) &= \begin{cases} e(i) & \text{if } i \in \text{dom}(e), \\
    i & \text{otherwise} \end{cases} \\
    c &= \{ \langle e'(s), e'(l), e'(t) \rangle | \langle s, l, t \rangle \in r.cruical \} \\
    \Delta^- &= \Delta^- \cup c
    \end{align*}

    # Edges produced by this step will never be considered again.
    # Remove them to avoid bloating the $\Delta^-$ set.
    $\Delta^- = \Delta^- - s$.produced

Figure 7.13: Update a step as part of resolving a removal in the distribution graph.

the consumed edges as crucial will often produce a conceptual change that matches a developer intuition – like disconnecting a conceptual connector when the underlying TCP/IP network connection drops. Where this property does not hold, we manually mark crucial prerequisites for \textit{Fiia.Net}.

\textbf{Figure 7.13} shows the extended routine for applying distribution removals. Rather than marking all consumed and required edges of the step for removal (as in
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

Figure 7.3, it uses \textit{r} crucial to retrieve the set of crucial prerequisites from the rule. These crucial prerequisites are then mapped through the step’s embedding, and added to edges to be removed. As long as at least one of the rule’s prerequisites is crucial, this approach maintains the correctness of our earlier core algorithm.

The choice of crucial prerequisites is not always as intuitive as our examples above, and so will benefit from further experimentation. If too few prerequisites are selected, the changes to the conceptual model may be too subtle for an application to easily monitor. On the other hand, if too many prerequisites are selected, a large portion of the conceptual model will be deleted, and isolating the underlying failure will be difficult. In \textit{Fiia}, most rules have a small number of prerequisites which are “obviously” crucial, and which, we hope, will make the origin of a failure easy for developers to understand.

7.5 Summary

In this chapter we have presented several extensions to our formalism and core algorithm. The algorithms and their extensions are collected in Appendix A. Through careful manipulation of the trace, these extensions preserve the performance and stability of trace-based update. At the time, our extended transformation rules are comparable to other graph rewriting techniques, and provide complete support for \textit{Fiia}. In addition, distribution removals can now be reflected up to the conceptual model in the way most comprehensible to developers. In \textit{Fiia} this takes the form of tracing failures back to the conceptual model elements which are now impossible to implement.
CHAPTER 7. EXTENDED TRACE-BASED UPDATE ALGORITHM

These extensions address many of the limitations we observed in Chapter 6 without sacrificing the unique advantages of trace-based update. This full extended algorithm provides the core of our Fiia.Net toolkit. In the next chapter we examine how Fiia.Net applies our novel model transformation algorithm to maintain an adaptive distributed system.
Chapter 8

The Fiia.Net Implementation

Through the earlier chapters of this thesis we have detailed our trace-based update algorithm. Our algorithm provides a novel combination of features, which allows model transformation at runtime to support adaptive distributed systems. To gain practical experience with our approach, we have constructed the *Fiia.Net* toolkit for adaptive groupware. *Fiia.Net* provides a programming model based on the *Fiia* conceptual model. It uses trace-based update to maintain a corresponding distribution model, which is then implemented by a specialized adaptive middleware.

In this chapter we explain the design and implementation of the *Fiia.Net* toolkit. We first describe the toolkit’s high-level structure, and how its functionality is distributed across nodes. We then delve into the major subsystems, examining how they bring our vision to life.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

8.1 Architecture

The core functionality of Fiiia.Net is divided among six subsystems, shown in Figure 8.1. These subsystems bridge from the developer’s high-level view, through Fiiia models and refinement, to instantiated objects on physical nodes.

Developers write code for application components, which are insulated from both distributed systems details and the internals of Fiiia.Net. An application component is usually an instance of a normal C# class: the node manager calls the constructor, and then uses properties and methods to interconnect different components. In Section 8.2 we describe how developers write application components.

Some application components need to inspect, monitor or adapt the system. These make calls into the architect, which serves as a high-level wrapper over the Fiiia conceptual model. In Section 8.3 we explain this high-level API, using examples from our furniture layout demo.
The application components also include infrastructure used to build the actual distributed system. A variety of general-purpose infrastructure components are included with Fiiia.Net, and developers can add specialized versions. An infrastructure component is an instance of a C# class that implements one of several special interfaces. These interfaces grant the component more control over its low-level connections.

The refinery uses model transformation at runtime, via our trace-based update algorithm, to maintain consistency between the conceptual graph and distribution graph. This bridge provides the novel capabilities of Fiiia.Net: the architect – and, in turn, developers – specify and understand change at the conceptual level, while the node manager deals with the concrete distribution level. In Section 8.4 we describe the refinery’s implementation of refinement rules and trace-based update.

Figure 8.2a shows the flow of communication during normal adaptations. When application components need to modify the adaptive distributed system, typically to express a user-driven change, they make calls into the architect. The architect translates these modifications into batches of changes to the conceptual graph. The single refinery uses model transformation at runtime, and our trace-based update algorithm, to...
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

maintain consistency between the conceptual graph and distribution graph. When the distribution graph changes, it broadcasts batches of changes to all node managers. The node managers then reconfigure objects on their local nodes to implement the distribution.

When part of the distribution fails, this process is reversed. One or more node managers detect the problem and update the distribution graph, removing failed elements. The refinery is triggered by these changes, and modifies both graphs to restore consistency. This, in turn, causes event broadcasts describing both conceptual and distribution changes.

8.2 Application Components

Application developers using Fiia.Net construct their distributed system from application components. These components are written by the developers, or drawn from pre-existing libraries. At runtime, the Fiia.Net toolkit creates and connects these components as described by the conceptual model.

Developing an application component is very similar to writing a traditional C# class. Indeed, unless the component needs to inspect or modify the conceptual model, it can be completely unaware of Fiia.Net. Application components that need more control, for example to create other components, access the Fiia static class.

In our furniture layout scenario, the quotation provides an example of a simple application component. This class is written in the normal C# style, as shown in Figure 8.3 without any Fiia.Net-specific changes.
class Quotation : IQuotation
{
    // This notification is fired when the list changes.
    public event EventHandler Changed;

    // Query or modify the items (implements IQuotation).
    public void Clear() { ... }
    public void Add( string name, int cost ) { ... }
    public string[] GetAll() {
        return (string[]) items.Clone();
    }

    // Private state of the list.
    private string[] items;

    public Quotation() {
        items = new string[0];
    }
}

Figure 8.3: Source of the furniture layout Quotation class

When such a component is created Fiia.Net inspects the native class – Quotation in this example – to determine how it can be connected to other objects. These available connection points include:

- Interfaces implemented by the class (e.g. IQuotation).
- Properties and fields, which may be either read-only or read-write.
- Events, which C# uses to sequentially call multiple methods (e.g. Changed).
- Single methods, which are often used as the targets of events (Clear).

Any connection point can be used with either Fiia call or event connectors. This flexibility allows developers to use many normal C# classes as components, such as forms created in the GUI builder.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

```csharp
var data = new Quotation();
var view = new QuotationViewer();

view.Items = data;
data.Changed += view.Refresh;
```

Figure 8.4: Quotation and viewer created in a single process.

```csharp
var list = Fiia.NewStore<Quotation>();
var view = Fiia.OnNode( "Clive’s PC" )
            .NewActor<QuotationViewer>();

Fiia.CallConnect( view[ "Items" ], list[ "IQuotation" ] );
Fiia.EventConnect( list[ "Changed" ], view[ "Refresh" ] );
Fiia.SyncConnect( list, "Shared Quotation" );
```

Figure 8.5: Quotation and viewer created in a Fiiia.Net application.

In the furniture layout scenario, Sally’s application creates components for the quotation on Clive’s PC (Figure 2.3b). This conceptual adaptation is simple, but implementing it requires establishing new communication channels, copying data across the network and creating several remote objects. However, because Fiiia.Net abstracts these issues, the code in Sally’s application is quite simple. For a traditional C# application, running in a single process, creating and connecting the objects could be written as shown in Figure 8.4.

A similar structure in Fiiia.Net would be requested by calls to the architect, which is accessible to application code through the Fiiia static class. This sample is shown in Figure 8.5. These operations following the same terminology as the Fiiia conceptual model. For example, Fiia.CallConnect creates a call connection (→) between the two specified components. This sample is part of the transition between scenes 1 and 2 in our scenario from Chapter 2.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

This Fii.Net version improves on the single-process C# code in several important ways:

- The “view” component is explicitly anchored on Clive’s PC. This is all that is required to ensure the component uses Clive’s display and input devices, rather than Sally’s multi-touch table.

- EventConnect specifies that the connection between list.Changed and view.Refresh is non-blocking.

- SyncConnect requests that the quotation be kept identical to other lists in the group “Shared Quotation”. Fii.Net will automatically duplicate the existing quotation, and keep all copies updated.

Like this example, our experience suggests that most application code can be written without using Fii.Net-specific features. Where the application needs to specify behavior beyond that of normal C#, it makes calls to the Fiia static class. This division means that developers can often build and test components in normal applications, or reuse existing code.

8.3 Architect

When developers request conceptual adaptations they use a high-level API based on the Fiiia conceptual model, like the Fiiia.NewStore example above. Because this API shares its structure and terminology with the Fiiia conceptual model, the code written to perform adaptations is very similar to the transitions between scenes when Fiiia is used as a design notation, (e.g. in Chapter 2).
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

The structure of the Fiia meta-models is quite different from the graphs upon which our trace-based update is defined. To bridge this gap, the architect component transparently converts conceptual adaptions into corresponding graph edits. Queries to the conceptual model are likewise converted to graph searches. The graph form of a Fiia model is never exposed to application developers.

This conversion provides the basis for implementing adaptations. For example, consider the QuotationViewer component created in Figure 8.5. The architect takes the single conceptual model operation (NewActor), and generates a corresponding change to the conceptual graph. In this case it adds the following graph edges, where QV is the unique name generated for the component:

```
QV is Actor
QV in CurrentSetting
QV type QuotationViewer
Clives_PC anchors QV
```

This edit produces four new edges in the conceptual graph, which describe the requested component. The CurrentSetting label is replaced by the setting in which the calling code is executing.

8.4 Refinery

After the architect constructs a graph edit and commits it to the conceptual graph, the refinery must apply the update. To accomplish this, it uses the extended trace-based update algorithm defined in Chapter 7. Each refinement rule used by the refinery is defined in a C# class. The API for writing this rules is based on the textual rule definitions shown earlier.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

```csharp
public sealed class AnchorRule : Rule {
    public AnchorRule() {
        RestrictVariable("?compType", "ConcComponent");
        Require("?comp is ?compType");
        Forbid(new Pattern()
            .Require("?anyNode anchors ?comp");
        Require("?stng is Setting");
        Require("?comp in ?stng");
        Require("?node is Node, in ?stng");
        Produce("?node anchors ?comp");
    }
}
```

Figure 8.6: Definition of the Anchor refinement rule.

For example, Figure 8.6 shows the source code definition of the anchor rule, discussed previously in Figure 7.5. All names starting with a question mark are variables, which, by default, may be mapped to any value. The `RestrictVariable` method limits a variable to one of the specified values, as in the $r.\nu$ mapping. In this case, it specifies that the anchored component must be a “ConcComponent” – shorthand for the conceptual component types Store, Reactor, or Actor.

Calls to the `Require`, `Produce` or `Consume` methods add edges to the rule’s corresponding set of edges. The parameter to these calls is a string specifying the source, edge label, and target of the new edge. In some cases a shorthand is used to specify multiple edges: “?node is Node, in ?stng” indicates the two edges “?node is Node” and “?node in ?stng”.

134
public sealed class InstantiateRule : Rule
{
    public InstantiateRule()
    {
        RestrictVariable("?compType", "ConcComponent");

        Consume("?comp is ?compType");
        Consume("?node is Node, anchors ?comp");

        Require("?stng is Setting");
        Consume("?comp in ?stng");

        Produce("?cccm is Cccm, in ?node");
        Produce("?comp is UserComponent, in ?node");
        Produce("?cccm localTo ?comp");

        Repeat(new Rule()
            .RestrictVariable("?portTypeTarget", "TargetPort")
            .Consume("?port on ?comp")
            .Require("?port is ?portTypeTarget")
            .Produce("?port on ?cccm");
        )
    }
}

Figure 8.7: Definition of the Instantiate refinement rule.

The Forbid method takes a pattern object describing a forbidden precondition. In this example, it specifies that the ?comp must not currently be anchored to a node. A pattern object provides only the RestrictVariable and Require methods, since it does not express produced or consumed edges.

The instantiate rule in Figure 8.7 was previously introduced in Figure 7.8. This rule definition adds a repeated section using the Repeat method. Repeat takes a full Rule object, which is usually defined inline, as shown. This repeated block moves all target ports from the component to its newly-created helper CCCM. “TargetPort” is a shorthand for any of the CallTarget, SubscrTarget, and Sync port types.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

The refinery’s model transformation system is implemented in C#, like the rest of FIIA.Net. To provide good performance during refinement, we applied several simple optimizations:

- Graphs are stored as multiple hashtables, indexing on both forward and reverse edges. These provide nearly-$O(1)$ queries, insertion and deletion; as well as $O(n)$ intersection and union.

- Each pattern used by the rules is compiled in advance to a simple bytecode, which the recursive-descent pattern matcher executes. This avoids the cost of parsing the pattern and looking up variable names for every search.

- When the refinery executes a rule, it collects all matches and tries to apply the rule on each of them. This differs from the procedure shown in Figure 5.9, which uses only the first match. Because refinement is unordered and non-deterministic, these different orderings are equally correct.

8.5 Optimization

One of the major advantages of our approach to adaptive distributed systems is decoupling the high-level programming model from the precise implementation chosen for the distributed system. By using the FIIA refinement rules, our model transformation system can produce a wide variety of possible distribution models for each conceptual model. This allows FIIA.Net to support many different implementations of an application, and switch between them both automatically and with guidance from application developers. However, this flexibility is a double-edged sword: choices
made in the Fiia refinement can produce both very good implementations, and very bad implementations.

A simple example of this problem is seen when Fiia assigns components to physical nodes. Components which communicate often should be clustered on the same nodes. Likewise, those with large memory or processing requirements should be spread across more powerful nodes. The Fiia refinement rules do not capture these preferences: any assignment of components to nodes is equally valid.

In Fiia.Net we have added several features which help guide the refinement to a good implementation, and allow the application developer to supply additional hints. First, we introduce several annotations, through which the application developer, toolkit or user can dictate aspects of the implementation. Second, we add strategy modules to the refinery, which can automatically manipulate the Fiia models and annotations.

8.5.1 Annotations

Part of the challenge of using model transformation to generate adaptive distributed systems is controlling the refinement. Developers and automatic systems must be able to influence aspects of the distribution, ideally without an intimate knowledge of the refinement rules. For example, Fiia can implement an event stream using either a centralized queue or a distributed broadcast protocol. This choice is not controlled by the conceptual model, so Fiia does not provide a direct way to override it.

In Fiia.Net we customize the refinement and implementation using annotations, which are arbitrary key-value pairs attached to conceptual elements. For example, an event stream can be tagged with annotations to indicate that it must be centralized,
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

or that it must be distributed using a particular protocol. We use three kinds of annotations, which are processed by different parts of the refinery:

- **Conceptual annotations** are high-level instructions that describe quality attributes. For example, connectors can be marked as “low latency”, to request that their endpoints be placed on the same node. In a future version of Fiia.Net we could mark the event stream above as high bandwidth, requiring privacy or with low tolerance for jitter to influence the implementation.

- **Refinement annotations** provide instructions to specific Fiia refinement rules. For example, the event stream above can be marked with “centralized only” or “distributed only” to suppress some of the possible refinements. We currently specify refinement annotations manually, but hope that future work can automate them through strategy modules and conceptual annotations.

- **Distribution annotations** specify the exact infrastructure components used in a distribution. This family of annotations is suggested by Fiia [101] to allow developers to choose between different implementations of infrastructure components, and plug in customized version. For example, Fiia.Net offers special threading support for several types of components (multi-threaded, single-threaded, Microsoft Windows forms, Microsoft XNA games) which is controlled by a distribution annotation.

We have used annotations in several of our applications. The furniture layout demo took advantage of “low latency” connectors, and threading support for both Microsoft Windows forms and Microsoft XNA games. Fletcher [51] used distribution annotations to plug-replace and evaluate various consistency maintenance algorithms.
Our limited experiences reflect positively on the power and simplicity of annotations. However, we have not extensively explored the capabilities offered by annotations, or rigorously evaluated them.

### 8.5.2 Strategy Modules

The second part of our optimization approach is provided by strategy modules added to the refinery. These modules monitor changes in the conceptual graph, and then automatically generate appropriate annotations.

For example, one of the standard strategy modules in Fiia.Net provides the “low latency” conceptual annotation. This module monitors the conceptual graph for new components, and then uses the connections of the new component to guess what node it should be instantiated on. The strategy module then adds a refinement annotation which forces the new component onto the selected node.

### 8.5.3 Summary

Annotations and strategy components provide our first steps into automatically generating good implementations of adaptive distributed systems, and allowing developers to easily tune them.

Our experiences are positive, and we have used both in our applications and evaluation. However, we have conducted a rigorous evaluation of the ease with which annotations can be used or their expressive power.
CHAPTER 8. THE FIIA.NET IMPLEMENTATION

8.6 Experiences with Fiia.Net

The current Fiia.Net implementation is the fourth iteration of research prototype based on Fiia. In each we have experimented with different developer APIs and underlying infrastructure, moving toward a system suitable for formal developer studies. Fiia.Net satisfies many of the requirements for widespread studies, but still has limitations that would confound the results.

Other members of our research lab have applied Fiia.Net to create several interesting applications. These include the furniture layout application presented in Chapter 2, and the Raptor game prototyping tool [119]. We have also developed several smaller examples, including text-based chat, collaborative drawing and document editing programs, and mock-ups of voice- and video-conferencing. This experience has highlighted the power of model transformation at runtime, but identified several shortcomings in our prototype implementation.

We found several differences between native objects and Fiia.Net components that cause problems for developers. The most severe is in the handling of passed objects: like many other transparent remote procedure call systems, including .Net Remoting, Fiia.Net will silently copy parameters and return values. This means that modified objects may or may not affect their origin, depending on whether the procedure call happens to cross a node boundary. Tilevich et al. [128] have approached part of this problem in Java using a technique they describe as “call-by-copy-restore”: copying modified parameters back across the node boundary when an method invocation returns. We implement a similar approach for C# ref and out parameters, which overwrite the passed variable. A complete solution to this problem appears to require a programming language, like C++, which natively supports pass-by-copy.
A second major difference appears when developers are creating and manipulating Fiia.Net components. Rather than calling constructors and passing references, they must make calls into our developer API. This API exposes the current Fiia conceptual model, so invoking it requires substantial knowledge of Fiia. We believe that a visual editor for Fiia diagrams could help developers understand the effect of changes, and synthesize code to call our API. If coupled to the live system, this editor could also be a valuable aide for experimentation and debugging.

Finally, our applications have uncovered weaknesses in the adaptive distributed middleware underlying Fiia.Net. This middleware is responsible for implementing the current distribution model and reporting any partial failures back into it. When adaptations occur, the middleware must reconfigure both itself and any application-specific components. This reconfiguration is quite complex: components regularly need to be copied or moved between nodes, and connectors are often changed while calls over them are outstanding. Our current middleware supports adding and removing components and local connectors quite reliably. However, we have observed exceptions and deadlocks during more complicated inter-node adaptations. Resolving these problems safely will require enhancements to Fiia that guide how the adaptations are performed. This is similar to research on dynamic variability, such as that by Bencomo et al. [13], which precisely models the process of low-level adaptations.

8.7 Summary

We have developed Fiia.Net to provide a practical platform for experimenting with trace-based update and model transformation at runtime. By building from
the high-level conceptual model we have been able to provide a programming model for adaptive groupware that integrates well with C#. Behind this simple programming model lies the Fiia.Net refinery, which generates and maintains an optimized distribution from the conceptual model.

In this chapter we have outlined various contributions of the Fiia.Net toolkit. Next we return to our problem statement in Section 1.3 and evaluate how well our toolkit and algorithm far when confronted with real adaptive distributed systems.
Chapter 9

Evaluation and Experience

In the first chapters of this thesis we introduced the promise and complexities of adaptive distributed systems, and particularly adaptive groupware. These systems pose many challenges to developers, some inherited from distributed systems or human-computer interactions, and some altogether new. In Chapter 1 we proposed several requirements for adaptive groupware toolkits, based on the substantial body of related literature. These requirements suggest an approach based on model transformation at runtime. However, coupling this model transformation to a live distributed system demands a novel combination of capabilities and performance. The transformation must be expressive enough to capture the complex decisions involved in implementing distributed systems, while still rapidly performing updates as the models change.

Our core contribution in this thesis is the novel trace-based update algorithm defined in Chapters 5, 6 and 7. In Section 9.1 we will evaluate how this algorithm satisfies the needs of model transformation at runtime.
CHAPTER 9. EVALUATION AND EXPERIENCE

To determine whether model transformation at runtime is useful for adaptive distributed systems, we have applied trace-based update in our Fiia.Net toolkit for adaptive groupware. Fiia.Net, described in more detail in Chapter 8, wraps the pre-existing Fiia meta-models and refinement rules with a customized programming API and distributed infrastructure. In Section 9.2, we discuss our experiences with the Fiia.Net tool, and many of the lessons learned during its development.

9.1 Trace-Based Update

For our trace-based update to serve as a foundation for model transformation at runtime, it must satisfy the five requirements stated in Section 1.2: the transformation rules must be expressive enough to describe complex nested decisions, the transformation must support conceptual and distribution adaptations, and updates must be performed quickly and stably.

We evaluate these requirements in two groups. First, we qualitatively compare the expressiveness and support for conceptual and distribution adaptations of trace-based update with the alternatives reviewed in Chapter 4. Second, we present quantitative experiments to demonstrate the speed and stability of conceptual and distribution adaptations performed using trace-based update.

9.1.1 Qualitative Analysis

The model transformation literature presents a huge variety of transformation techniques, with many different combinations of features. In general, as described in Chapter 4, these divide into rewriting-based approaches, which offer very expressiv
transformations, and relational approaches, which offer bidirectionality and fast updates.

Transformation Expressiveness

To provide good support for adaptive distributed systems, and other applications of model transformation at runtime, the transformation must support complex nested decisions. This allows a single conceptual model to describe more than one possible distribution model, while the individual refinement rules remain relatively simple. This is the case in Fiia, where most of the 34 refinement rules are very simple. However, their combination can support a tremendous variety of possible distributions.

Our review of the existing literature suggests that refinements with such a large semantic gap between conceptual and distribution models, are best approached using rewriting rules and intermediate elements. Addressing the same gap in a relational transformation would require explicitly breaking the transformation into numerous layers, or an exponential increase in the number of rules. Even extended relational techniques like TGGs would require complex matching of “link” nodes.

Our basic expressiveness objective is to support the complete Fiia refinement rules. Phillips and Graham defined Fiia based on a generic visual rewriting language [105], which predates the trace-based update algorithm presented in this thesis. Phillips’ Ph.D thesis [104] uses a typical forward-only rewriting transformation, with negative preconditions and multiplicities. All of these features are provided by the complete trace-based update algorithm.
In short, we believe that rewriting rules and intermediate elements are required to support a practical refinement. Both are provided by trace-based update.

**Conceptual Update**

Support for conceptual updates is relatively common in the model transformation literature, and complete across the techniques we reviewed earlier. However, model transformation at runtime requires that the update be performed quickly and with few unnecessary changes to the distribution model. Of the techniques we reviewed, only those using parallel update satisfied these criteria, and they failed the expressiveness requirement.

The behavior of trace-based update during a conceptual update depends on both the refinement rules and the particular change. If rules are small and with local prerequisites, as in Fiia, and the change is small and with local effects, as typical in adaptive distributed systems, the resulting update will be very fast and stable. When these conditions fail to apply, trace-based update may degrade to a complete refinement, which is relatively slow and entirely unstable.

The first condition, that the rules are small and local, is easy to ensure when designing the refinement. Small and local rules will also be easier for rule authors and users to understand, which motivated their use in Fiia. Rule authors can mitigate the cost of larger rules by ensuring that the prerequisites change rarely, or that few other rules can depend on them. Further experience with trace-based update will provide more specific guidelines about appropriate rule design, and its performance and stability impacts.
CHAPTER 9. EVALUATION AND EXPERIENCE

The second condition, that changes by small and with local effects, is impossible to guarantee. Most changes in adaptive distributed systems satisfy this conditions: they are triggered by a user action, and impact only a small part of the system. However, occasional changes will be large. For example, a group collaborating over a document could decide to quit and play a networked game instead. Many such large changes, like this one, are intended to discard much of the existing distributed system. From both user interface and distributed systems perspectives, a few seconds of latency is expected.

As a result, we believe that trace-based update will perform conceptual updates quickly and stably in common cases for adaptive distributed system. In cases where it will degrade, this is either avoidable, in the case of poor rules, or not a detriment to the user experience, in the case of large conceptual changes.

Distribution Update

Support for distributed updates is relatively rare in model transformation techniques. It is usually addressed by using a relational technique with bidirectional rules. By limiting distribution updates to removals in model transformation at runtime – representing failure in the distributed system – we can instead use a more expressive rewriting technique.

Of our examples in Chapter 4, only one model transformation technique offered both expressive rules and distribution updates [71]. Its update depended on a slow logic system, and stability varied with the non-deterministic order in which rules were used.
As was the case for conceptual update, the behavior of trace-based update during a distribution update depends on both the refinement rules and the change. Small and local rules and changes, again, produce very fast and stable updates. Our addition of crucial preconditions in Section 7.4 limits the impact of small distribution changes, and helps ensure that the impacts will be meaningful to application developers.

Large distribution changes are worthy of special consideration. These will discard large portions of the conceptual model and distributed system. However, in our application to adaptive distributed systems, a large distribution change is indicative of a large failure. These will be rare compared to small, local failures. In addition, a large failure will often mean that the affected applications can not offer any reasonable repair, and will simply shut down with an error.

As a result, we believe that trace-based update will perform distribution updates quickly and stably for small failures. In the case of large failures, large parts of the distributed system will likely be non-functional anyway.

Discussion

As we have seen, trace-based update provides a significant advantage over other model transformation techniques by combining the expressiveness of rewriting rules and intermediate elements, with fast and stable updates. Because trace-based update builds upon the well-known semantics of unordered graph rewriting, many existing techniques can be used to enhance its future performance and features.

A particular limitation of our current trace-based update is the lack of explicit rule ordering. This contrasts with many graph rewriting tools that offer structure such as layers or a programmed control flow [20, 53]. Such constraints make it easier to
verify the completeness and termination of the transformation, but require considering dependencies external to the trace. In subsection 10.1.3 we propose a future extension of our repeated sections to allow explicit hierarchies of rules. This generalization offers many of the benefits of explicit rule ordering, while still permitting a fast trace-based update.

9.1.2 Experimental Analysis

Applying model transformation at runtime to control a live distributed system requires that updates to the transformation be exceptionally fast and stable. Changes in the models are often caused by user-initiated actions or in recovering from failure. Based on typical user interface latency guidelines, the entire transformation and distribution change should be completed in a few hundred milliseconds to a few seconds.

Large conceptual changes will typically require creating new network connections. As a result, even adaptations for which the user can tolerate a few seconds delay may need to be refined faster. Small conceptual changes are often entirely transparent to users, so will need to be refined and implemented in a fraction of a second.

Context

To evaluate the performance of our trace-based update, we turn to an application developed by David Smith [119], also under Nicholas Graham’s supervision. The Raptor game prototyping tool, shown in Figure 9.1 lets game designers control parts of the game world on a multi-touch table, while a tester plays the same game on a
Figure 9.1: The Raptor game prototyping tool in action

Figure 9.2: Conceptual model of the Raptor game prototyping tool.

Raptor creates several core components and then adds one component for each game entity. Figure 9.2 shows the conceptual model with a single car game entity. When a designer adds additional cars, each creates a car component with two connectors. Removing a car deletes the corresponding component and its connectors. These conceptual changes are quite simple, but will occur continually while Raptor is being used.
Throughout these experiments we use the Fiia.Net refinery, but never create the actual distribution. This allows us to easily measure transformation time, in isolation from confounding factors like network latency. Each transformation is performed in one thread on an Acer Aspire 5110 laptop (AMD TL-50 1.6GHz with 2GB RAM, running Windows XP and Microsoft .Net 3.5).

**Experiment and Results**

Figure 9.3 uses the Raptor example of randomly adding and removing cars to compare the performance of trace-based update to a full refinement. In both cases, the experiment gradually expands the game world from empty to 1000 entities, by randomly mixing 2000 additions with 1000 removals. The sizes of the refinement trace, conceptual graph and distribution graph grow linearly with the number of entities. At 1000 entities, the refinement reaches 8022 steps, with 16054 and 19076 edges in the conceptual and distribution graphs, respectively.
CHAPTER 9. EVALUATION AND EXPERIENCE

![Graph showing Adaptation Time (seconds) vs. Distribution Graph Edges]

Figure 9.4: Time required to apply small conceptual and distribution deletions.

Applying a full refinement after each adaptation rapidly becomes too expensive for interactive applications, peaking at nearly 5 seconds. The trace-based update algorithm performs much better, remaining below 500ms. This gap appears because the `prune` operation preserves almost all of the previous trace, so the ensuing `refine` requires few graph searches. Examining the execution of our algorithm, we find that almost all of the processing time used by trace-based update is spent applying the trace and checking repeated sections.

Figure 9.4 shows the performance of the same Raptor scenario when removing cars using either conceptual or distribution adaptations. Applying the adaptations in the conceptual model behaves like the earlier incremental addition test. The distribution adaptations are slightly different, because removing components from the distribution model indicates their failure. To ease application recovery, Fiia.Net preserves the connectors of failed components. This behavior is responsible for the slightly higher performance of the distribution test, and the additional 10000 distribution edges remaining afterward. Again, the performance is acceptable for interactive use.
CHAPTER 9. EVALUATION AND EXPERIENCE

Discussion

Both of these Raptor scenarios perform simple, but common, changes in the conceptual model. Each added car brings a component and two connectors, requiring only 16 conceptual graph edges. In spite of this simplicity the Fiia refinement rules permit many distributions: hundreds of possibilities for the core components, multiplied by ten for each added car.

However, many of these distributions are obviously poor. In Figure 9.2, there is one good distribution: placing all cars on the table node, along with the game editor and their scene. Fiia.Net uses a simple heuristic of trying to placing connected components on the same node, so will always choose this implementation. Other options are not controlled by Fiia.Net heuristics; for example, it will tend to replicate the scenes, but has no preference in how the communication between them is performed.

A full refinement will choose arbitrarily among alternative implementations. In the conceptual experiment above, the Fiia.Net heuristics restricted the full refinement to one of five distributions, among which is chose randomly. If we disable the Fiia.Net heuristics, the number of distributions balloons to exponential on the number of components. Using trace-based update, regardless of the heuristics, the only changes were for the single added car. These results show that even for simple examples, where the Fiia.Net heuristics exclude many possible distributions, trace-based update provides a significant improvement in stability.

With effort and more heuristics, a full refinement could be forced to complete stability. However many of these heuristics would be entirely arbitrary: for example, when implementing communication between replicated component Fiia.Net could
CHAPTER 9. EVALUATION AND EXPERIENCE

default to using a FIFO queue placed on the node with the lexicographically lowest name. These checks would be numerous, complex to create, and extremely prone to error and counter-intuitive behavior. With a heuristic using lexicographically lowest name, our example in Figure 9.2 would have varying distributions – and hence performance – depending on the computer names. We prefer to use simple and safe heuristics, which contribute to choosing good distributions. Improving the stability of full and separate refinements, such as between two runs of application, is delegated to future work.

9.1.3 Summary

These results show that our trace-based update algorithm and its implementation in Fiia.Net satisfy the requirements of model transformation at runtime. Our algorithm supports typical unordered refinement rules, including those used in Fiia. When changes occur in either conceptual or distribution models, the refinement is updated quickly and stably. This combination can be augmented with developer hints or heuristics where appropriate, but, unlike other alternatives, do not require them.

9.2 Fiia.Net Toolkit

We have developed the Fiia.Net toolkit to demonstrate model transformation at runtime for adaptive distributed systems. This research prototype builds upon trace-based update and the Fiia meta-models and refinement. As described in Chapter 8, Fiia.Net wraps its model transformation core with a developer API and substantial
distributed infrastructure. It allows components to be written in idiomatic Microsoft C#, and then created and connected by calls into the Fiia.Net runtime system.

To evaluate the performance of the Fiia.Net toolkit, we return to the furniture layout application described in Chapter 2. The first implementation of this application was developed by Banani Roy [137] using Fiia.Net to support adaptation and a wide range of possible implementations. For this experiment, we re-implemented it using Microsoft .Net Remoting, the transparent remote procedure call facility integrated in Microsoft .Net. Both versions contain nearly identical code. They differ only in their startup, and a few utility superclasses required by .Net Remoting. This simplicity is reassuring: it suggests that, in spite of the greater flexibility of Fiia.Net, programming for it is comparable to using a single process, and no worse than transparent RPC.

An important result of Fiia.Net’s flexibility is the ability to switch between different distributed implementations without modifying the application. We examine the runtime performance of our furniture layout application using the scene described in Section 2.2 as Sally edits furniture on her multi-touch table while Clive watches the changes from his PC.

9.2.1 Method

Three metrics were used to assess performance: frame rate, the number of times per second that the first-person viewer is able to update; feedback time, the time it takes Sally to see her changes; and feedthrough time, the time it takes Clive to see each of Sally’s changes. Frame rate and feedback time were measured directly in the first-person viewer and schematic editor, respectively. The feedthrough time was calculated
CHAPTER 9. EVALUATION AND EXPERIENCE

Figure 9.5: Case study Fiia.Net runtime performance as the time used by Sally’s computer to update the floor plan, plus the time required for Clive’s computer to fetch data and display a new frame.

We measured two conditions: communicating over a fast local-area network, and over the Internet to a slow DSL link. In both cases Sally’s PC was a 2.4GHz Intel Core 2 with 2GB RAM and an NVIDIA 6660 GPU. In the local-area case, Clive’s PC was a 1.6GHz AMD Turion X2 with 2GB RAM and an ATI X1300 GPU. In the wide-area case, Clive’s PC was a substantially faster 2.0GHz AMD Turion X2 with 3GB RAM and an NVIDIA 7000M GPU.

9.2.2 Results

The results of the experiment are shown in Figure 9.5. The .Net Remoting and Fiia Centralized variants use the same client-server data distribution: storing a single copy of the floor plan on Sally’s PC, and repeatedly querying it from both computers. The performance of these cases is not significantly different, suggesting that Fiia.Net’s extra flexibility does not harm runtime performance.
CHAPTER 9. EVALUATION AND EXPERIENCE

These centralized distributions do not fare well in the wide-area case. Here the added latency and bandwidth limitations drop Clive’s frame rate from just under 60 to less than 10 – from excellent to uncomfortably slow. Resolving this problem in .Net Remoting would require the developer to manually replicate the floor plan. In Fiia.Net, we simply changed one line of code, removing the annotation which forced it to use the inefficient centralized distribution.

This Fiia Replicated variant stores a copy of the floor plan on each PC, and automatically communicates updates across the network. In the local-area case Fiia Replicated performs equivalently to the other variants. Here the few dozen microseconds added by Fiia.Net replication are overshadowed by the network latency. Moving to the wide-area case shows the advantage of replicated data: Clive’s frame rate and the feedthrough time worsen slightly, but remain excellent.

9.2.3 Discussion

The Fiia.Net toolkit supports developers of adaptive distributed systems by using model transformation at runtime. We have demonstrated the simplicity and performance of the furniture layout application implemented in Fiia.Net, and shown it easily optimized to different distributions.

Significant future work remains to make Fiia.Net practical for industrial use or formal development studies. However, our current research prototype has provided valuable experience with applying model transformation at runtime, and developing highly adaptive distributed systems.
CHAPTER 9. EVALUATION AND EXPERIENCE

9.3 Summary

The premise of this thesis is that model transformation at runtime is useful for adaptive distributed systems, and that trace-based update can support practical model transformation at runtime. In this chapter we have evaluated both of these claims, with encouraging results.

We show that our trace-based update algorithm satisfies the features required by model transformation at runtime. It provides expressive transformation rules, suitable for defining the process of implementing a distributed system. When adaptations occur in either the conceptual or distribution model, trace-based update can rapidly apply them with excessive side-effects. This combination of expressiveness and fast, stable, bidirectional update is a significant contribution to the state of the art.

Using our Fiia.Net research prototype, we present a case study which supports the usefulness of model transformation at runtime for adaptive distributed systems. We are able to express the structure and adaptation of our furniture layout scenario at a high level, independent of a particular implementation. This property is used to demonstrate easy optimization from a client-server distribution to one using replicated data, drastically improving performance over a wide-area network.

These results support the utility of model transformation at runtime and trace-based update. We have sketched the limitations of Fiia.Net, and presented various suggestions for future work to resolve them. In the next and final chapter, we reflect on the capabilities of trace-based update and suggest areas for future enhancement.
Chapter 10

Conclusion and Future Work

We have presented the trace-based update algorithm for model transformation at runtime. This algorithm is the first which makes model transformation at runtime practical for controlling adaptive distributed systems. It lets developers work exclusively within a high-level view, while building distributed systems that dynamically change their structure at runtime. This contrasts with previous model-driven approaches by supporting easy adaptation in response to user operations, changing runtime environments and partial failures.

We have implemented trace-based update within our Fiia.Net toolkit for adaptive groupware, using the pre-existing Fiia meta-models and refinements. In Chapter 9 we have shown that this combination provides novel and powerful capabilities. At the same time we have encountered limitations of our algorithm and Fiia.Net, which suggest topics for future research.

In this chapter we present suggestions for future work. We first examine several possibilities for extending trace-based update to more powerful refinement rules. Next we review opportunities to improve its performance.
10.1 Extended Refinement Rules

A major goal of this work is to support fast bidirectional updates to a model transformation without limiting the possible refinement rules. We have accomplished this well beyond the state of the art, but have also identified several opportunities for future enhancement.

Chapters 5 and 6 provide a formal core for the trace-based update algorithm. Our extensions into forbidden preconditions and repeated sections, in Chapter 7, are deliberately less rigorous. Future theoretical work should either formalize the extensions, or build from the core with the extensions as a guideline.

10.1.1 Programmed Guards

In Section 7.2 we showed how to extend the core trace-based update algorithm with forbidden preconditions. This change adds forbidden patterns to our rule syntax. These patterns must be checked whenever steps are reused with a different conceptual graph. However, these checks need not be limited to simple graph patterns. Indeed, they can perform arbitrary checks on the intermediate graph $G$ used in Figure 7.7.

An enhancement to trace-based update could permit these checks to be programmed explicitly. This would allow rules to perform more complex tests, and consult sources of data beyond the intermediate graph.

For example, rules could avoid creating incoming network connections to a node known to be behind a firewall. Similar checks are useful to avoid both impossible distributions and those with unfavorable performance. The information would be stored outside the conceptual graph, using a dedicated platform model [108].
Likewise, guards could be used to enforce mandatory access controls. Components would be assigned security profiles that place limits on how they communicate and where they can be instantiated, as proposed by previous work [126].

Authors of these guards would currently need to be careful to preserve the completeness of the refinement rules. An additional extension could use our algorithm for distribution removals to discard parts of the graph which are not completely refined, and propagate the failure back to the conceptual model.

### 10.1.2 Programmed Transformation

Throughout this thesis we have relied on declarative graph rewrite rules to perform all refinement. This is sufficient for Fiia, but many transformation techniques also support imperative code within their rules. These programmed transformations are particularly useful for manipulating attributes, often using substrings and concatenation.

An enhancement to trace-based update could easily support programmed transformations. Very few of our algorithms depend on how rules choose their consumed, required, and produced sets. Simple programmed transformations, which depend on only the embedding of the rule, could be supported by changing only the `applyRule` algorithm. More complex possibilities would require extensions similar to our handling of repeated sections, but be more difficult to update efficiently.

### 10.1.3 Hierarchical Transformations

As foreshadowed in handling repeated sections, our approach can be generalized to true nested rules. Rather than merely repeated sections, each rule could contain a
set of other rules. When applied, these compound parent rules would produce steps which contained a full trace. The trace-based update algorithm would then be used recursively to propagate changes through steps and their nested children.

This extension would provide a way to structure rules into logical blocks. This would simplify the Fiia refinement rules, as several sets of rules could then be combined. Currently, these groups are too complex to merge using the current repeated sections.

10.2 Accelerating Trace-Based Update

The performance of trace-based update, and our current simple implementation, are already acceptable for distributed systems involving several dozens of components. This size is acceptable for the small, coarse-grained applications we have developed. However, larger models and frequent adaptations can still overwhelm our transformation engine. Extending our Raptor-based evaluation from subsection 9.1.2 to ten thousand game entities, we find the time to add a single new entity becomes prohibitive. This linear growth is a natural result of the iteration over steps used in our update algorithms to determine which, if any, steps have been invalidated by the change.

Another challenge is distributed systems with very rapid adaptations. Raptor uses components to represent long-lived game entities, like the player’s car. A similar system could attempt to create components for short-lived entities, like bullets fired from a player’s gun in a first-person shooter. In this approach – not recommended for Fiia or Fiia.Net – a quarter-second delay for refinement updates is unacceptable.
CHAPTER 10. CONCLUSION AND FUTURE WORK

We have identified two promising approaches to improve the performance of trace-based update for such examples. The first uses an area of interest to speed pattern matching and invalidation tests. The second distributes the algorithm across multiple nodes or threads.

10.2.1 Area of Interest

Our performance evaluation in Chapter 9 showed the update time growing linearly with the size of the conceptual and distribution graphs, and the number of steps in the trace. Substantially all of this growth stems from checking which steps are now invalid or need their repeated sections updated. The majority of the steps will remain unchanged, because they are completely independent of the adaptation.

The performance of our algorithm can be improved by skipping these checks for steps that can not be affected by the adaptation. The determination would be based on the shortest graph distance between the step and the modified edges. If this distance is greater than can be bridged by the step’s forbidden preconditions and repeated sections – a small constant value for each rule – the step will be unchanged.

10.2.2 Parallel Algorithm

We apply trace-based update within a distributed system, so distributing computations across the different nodes is an attractive technique. This can be approached from a Fiia-based perspective, or as a general enhancement to trace-based update.

Most of the Fiia refinement rules operate on components which have been anchored to a particular node, either explicitly by the developer or automatically by an earlier rule. This provides a natural structure to partition the refinement:
CHAPTER 10. CONCLUSION AND FUTURE WORK

a global refinery allocates components to nodes, and each node is then responsible for refining its own local components. Care would be needed to handle connections between nodes and ensure that the system handles partial failure correctly.

Extending the trace-based update algorithm to use multiple nodes poses interesting theoretical and engineering questions. Combining traditional graph partitioning algorithms with the information in the trace should let a clever technique collect clusters of related conceptual edges, trace steps, and distributions edges. Distributed transformation and update could then operate efficiently on these clusters.

Finally, our current trace-based update algorithm can be accelerated using multiple processors in a single node. Pattern matching and rule application can be done optimistically in parallel, then rolled back if a conflict is found. During update, trace steps with no dependencies between them can also be checked and updated in parallel.

10.3 Summary

Trace-based update provides a practical algorithm for model transformation at runtime, which is already usable for adaptive distributed systems. Its formal foundation and rigorous structure will support many interesting enhancements, allowing us to improve the general behavior or specialize it to particular refinement rules. We look forward to approaching some of these questions in the years to come, and investigating other applications of model transformation at runtime.
References


Appendix A

Collected Functions
<table>
<thead>
<tr>
<th>Variables</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C$</td>
<td>conceptual graph as a set of edges</td>
</tr>
<tr>
<td>$C_1, C_2$</td>
<td>conceptual graph before and after adaptation</td>
</tr>
<tr>
<td>$D$</td>
<td>distribution graph as a set of edges</td>
</tr>
<tr>
<td>$D_1, D_2$</td>
<td>distribution graph before and after adaptation</td>
</tr>
<tr>
<td>$G, H$</td>
<td>graph as a set of edges</td>
</tr>
<tr>
<td>$T$</td>
<td>trace as a list of steps</td>
</tr>
<tr>
<td>$T_1, T_2$</td>
<td>trace before and after adaptation</td>
</tr>
<tr>
<td>$T_G, T_H$</td>
<td>trace from the working graph to the distribution</td>
</tr>
<tr>
<td>$R$</td>
<td>global collection of all refinement rules</td>
</tr>
<tr>
<td>$r$</td>
<td>a rule</td>
</tr>
<tr>
<td>$r.vars$</td>
<td>mapping of variables in the rule to their possible values</td>
</tr>
<tr>
<td>$r.consumes$</td>
<td>set of consumed edges, may contain variables</td>
</tr>
<tr>
<td>$r.requires$</td>
<td>set of required edges, may contain variables</td>
</tr>
<tr>
<td>$r.produces$</td>
<td>set of produced edges, may contain variables</td>
</tr>
<tr>
<td>$r.forbids$</td>
<td>forbidden preconditions (added in Section 7.2)</td>
</tr>
<tr>
<td>$r.crucial$</td>
<td>crucial prerequisites of the rule (added in Section 7.4)</td>
</tr>
<tr>
<td>$r.repeats$</td>
<td>repeated sections (added in Section 7.3)</td>
</tr>
<tr>
<td>$p$</td>
<td>pattern as a set of edges</td>
</tr>
<tr>
<td>$p.vars$</td>
<td>mapping of variables to their possible values</td>
</tr>
<tr>
<td>$e$</td>
<td>embedding; a mapping of variables to vertices in a graph</td>
</tr>
<tr>
<td>$s$</td>
<td>step</td>
</tr>
<tr>
<td>$s.rule$</td>
<td>rule from which this step was created (added in Section 7.2)</td>
</tr>
<tr>
<td>$s.embedding$</td>
<td>embedding from which this step was created (added in Section 7.2)</td>
</tr>
<tr>
<td>$s.consumed$</td>
<td>set of consumed edges</td>
</tr>
<tr>
<td>$s.required$</td>
<td>set of required edges</td>
</tr>
<tr>
<td>$s.produced$</td>
<td>set of produced edges</td>
</tr>
<tr>
<td>$s.repeated$</td>
<td>list of steps created from repeated sections (added in Section 7.3)</td>
</tr>
</tbody>
</table>
### A.1 Match Pattern

```plaintext
function matchPattern(G, p, e = ∅) → (e)

inputs
  G : input graph
  p : pattern to match
  e : initial variable mapping

# This function will match the pattern p against graph G, and
# return a mapping as described in Section 5.2
# If no mapping is possible, the function will return e = undef

outputs
  e : resulting embedding, or undef if none was found
```
A.2 Match Rule

function matchRule\(G, r, e = \emptyset\) \(\rightarrow\) \(e\)

inputs
\(G\) : input graph
\(r\) : rule to match

# Derive the prerequisite pattern of the rule.
p.edges = \(r\).consumes \(\cup\) \(r\).requires
p.vars = \{ \(i \mapsto r\.vars(i)\mid i \in \text{dom}(r\.vars), \exists(s, l, t) \in p\.edges \cdot (i = s \lor i = t)\} \}

# Find the pattern’s embedding as described in Section 5.2
# If no embedding is available, matchPattern will return undefined.
e = matchPattern\(G, p\)
if not defined\(e\) then return \(e = \text{undefined}\)

# Expand the variable mapping to include produced variables.
for \(i\) in \(r\.vars\)
  if not defined\(e(i)\)
    \(e = e \cup (i \mapsto \text{a new unused identifier})\)

# Check forbidden preconditions.
for \(p_2\) in \(r\.forbids\)
  \(e_2 = \text{matchPattern}\(G, p_2, e\)\)
  if defined\(e_2\) then return \(e = \text{undefined}\)

outputs
\(e\) : resulting embedding, or \text{undefined} if none was found
A.3 Apply Rule

function **applyRule**(*G*, *r*, *e*) → (*G*, *s*)

inputs

*G*: input graph
*r*: matched rule
*e*: embedding of *r* in *G*

# Create a step by replacing all variables with their values.
let *e'*(*i*) = \(\begin{cases} e(i) & \text{if } i \in \text{dom}(e), \\ i & \text{otherwise} \end{cases}\)

*s*.consumed = \{ ⟨*e'*(*s*), *e'*(*l*), *e'*(*t*)⟩ | ⟨*s*, *l*, *t*⟩ ∈ *r*.consumes \}
*s*.required = \{ ⟨*e'*(*s*), *e'*(*l*), *e'*(*t*)⟩ | ⟨*s*, *l*, *t*⟩ ∈ *r*.requires \}
*s*.produced = \{ ⟨*e'*(*s*), *e'*(*l*), *e'*(*t*)⟩ | ⟨*s*, *l*, *t*⟩ ∈ *r*.produces \}

# Record the rule and embedding used to create the step.
*s*.rule = *r*
*s*.embedding = *e*

*G* = (*G* − *s*.consumed) ∪ *s*.produced

# Track the repeated sections in the step.
*s*.repeated = an empty list

# Apply repeated sections as many times as possible.
for *r*₂ in *r*.repeats
  while true
    *e*₂ = **matchRule**(*G*, *r*₂, *e*)
    if not defined(*e*₂) then break
  
  (*G*, *s*₂) = **applyRule**(*G*, *r*₂, *e*₂)
  *s*.repeated.append(*s*₂)

outputs

*G*: output graph
*s*: generated step
function \texttt{transform}(C) \rightarrow (D, T)

inputs
\hspace{1em} C : conceptual graph
\hspace{1em} G = C
\hspace{1em} T = empty trace
\hspace{1em} while any rules match \( G \)
\hspace{2em} r = an arbitrary one of the matching rules
\hspace{2em} e = \texttt{matchRule}(G, r)
\hspace{2em} # Apply the rule to the graph
\hspace{3em} (G,s) = \texttt{applyRule}(G, r, e)
\hspace{3em} T.append(s)
\hspace{1em} D = G

outputs
\hspace{1em} D : distribution graph
\hspace{1em} T : trace of steps
A.5 Conceptual Update

function updateConc($C_1$, $T_1$, $\Delta^-_C$, $\Delta^+_C$) → ($T_2$, $D_2$)

inputs
- $C_1$: previous conceptual graph
- $T_1$: previous trace
- $\Delta^-_C$: conceptual edges removed by change
- $\Delta^+_C$: conceptual edges added by change

# Apply the change to the conceptual graph
$G = (C_1 - \Delta^-_C) \cup \Delta^+_C$
$T = T_1$
for $s$ in $T_1$
  # Update the step.
  $s' = \text{updateConcStep}(G, \Delta^-, s)$
  if not defined($s'$)
    # This step is no longer possible.
    $T$.remove($s$)
    $\Delta^- = \Delta^- \cup s$.produced
  else
    # This step is still possible.
    $G = (G - s'.consumed) \cup s'.produced$

    # If the step changed, update the trace and $\Delta^-$ set.
    if $s \neq s'$
      $T$.replace($s$, $s'$)
      $\Delta^- = \Delta^- \cup (s.produced - s'.produced)$
      $\Delta^- = \Delta^- \cup (s'.consumed - s.consumed)$

# Refine the intermediate graph to a distribution.
$(G, T_H) = \text{transform}(G)$
$T_2 = T + T_H$
$D_2 = G$

outputs
- $T_2$: updated trace
- $D_2$: updated distribution graph
function \texttt{updateConcStep}(G, \Delta^-, s) \rightarrow (G, s)

**inputs**
- \( G \): previous graph
- \( s \): step to be updated
  - \( r = s.\text{rule} \)

# Check whether this step depends on edges that are no longer available.
\[ \text{if } \Delta^- \cap (s.\text{consumed} \cup s.\text{required}) \neq \emptyset \]
\[ \text{return } s = \texttt{undef} \]

# Check forbidden preconditions.
\[ \text{for } p \text{ in } r.\text{forbids} \]
\[ e = \texttt{matchPattern}(G, p, e) \]
\[ \text{if } \texttt{defined}(e) \text{ then return } s = \texttt{undef} \]

\[ G = (G - s.\text{consumed}) \cup s.\text{produced} \]

# Check for repeated sections which are now impossible.
\[ \text{for } s_2 \text{ in } s.\text{repeated} \]
\[ s'_2 = \texttt{updateConcStep}(G, s_2) \]
\[ \text{if } \texttt{not defined}(s'_2) \]
\[ s.\text{repeated}.\text{remove}(s'_2) \]
\[ \text{else} \]
\[ G = (G - s'_2.\text{consumed}) \cup s'_2.\text{produced} \]
\[ s.\text{repeated}.\text{replace}(s_2, s'_2) \]

# Apply repeated sections as many times as possible.
\[ \text{for } r_2 \text{ in } r.\text{repeats} \]
\[ \text{while true} \]
\[ e_2 = \texttt{matchRule}(G, r_2, e) \]
\[ \text{if } \texttt{not defined}(e_2) \text{ then break} \]
\[ (G, s_2) = \texttt{applyRule}(G, r_2, e_2) \]
\[ s.\text{repeated}.\text{append}(s_2) \]

# If the step is no longer possible, this function will
\[ \text{return an unchanged graph and } s = \texttt{undef}. \]

**outputs**
- \( G \): updated graph
- \( s \): updated step
A.6 Distribution Update

function $\text{delDist}(C_1, T_1, D_1, \Delta^{-}_D) \to (C_2, T_2, D_2)$

inputs
- $C_1$: previous conceptual graph
- $T_1$: previous trace
- $D_1$: previous distribution graph
- $\Delta^{-}_D$: distribution edges removed by adaptation

# Derive a sufficient conceptual change.
$\Delta^{-} = \Delta^{-}_D$

for $s$ in reverse $T_1$
- $\Delta^{-} = \text{delDistStep}(\Delta^{-}, s)$

# Apply the derived conceptual change.
$\Delta^{-}_C = \Delta^{-} \cap C_1$
$C_2 = C_1 - \Delta^{-}_C$
$(T_2, D_2) = \text{updateConc}(C_1, T_1, \emptyset, \Delta^{-}_C)$

outputs
- $C_2$: updated conceptual graph
- $T_2$: updated trace
- $D_2$: updated distribution graph
function delDistStep($\Delta^-, s \rightarrow (\Delta^-)$)

inputs
- $\Delta^-$: set of edges which are no longer available
- $s$: step to be checked

# Check whether any repeated sections must be removed.
for $s_2$ in $s$.repeated
    $\Delta^- = \text{delDistStep}(\Delta^-, s_2)$

if $\Delta^- \cap s$.produced $\neq \emptyset$
    # This step produces an edge which must no longer appear.
    # Mark the crucial prerequisites of this step to be removed.
    $e = s$.embedding
    let $e'(i) = \begin{cases} e(i) & \text{if } i \in \text{dom}(e), \\ i & \text{otherwise} \end{cases}$
    $c = \{ \langle e'(s), e'(l), e'(t) \rangle | \langle s, l, t \rangle \in r$.crucial $\}$
    $\Delta^- = \Delta^- \cup c$

# Edges produced by this step will never be considered again.
# Remove them to avoid bloating the $\Delta^-$ set.
$\Delta^- = \Delta^- - s$.produced

outputs
- $\Delta^-$: updated set of available edges