Design-by-Contract for Software Architectures

Thesis submitted for the degree of
Doctor of Philosophy
at University of Leicester

by

Kyriakos Poyias
Department of Computer Science
University of Leicester

May 18th, 2014
Design-by-Contract for Software Architectures
Kyriakos Poyias

Abstract

We propose a design by contract (DbC) approach to specify and maintain architectural level properties of software. Such properties are typically relevant in the design phase of the development cycle but may also impact the execution of systems. We give a formal framework for specifying software architectures (and their refinements) together with contracts that architectural configurations abide by. In our framework, we can specify that if an architecture guarantees a given pre-condition and a refinement rule satisfies a given contract, then the refined architecture will enjoy a given post-condition.

Methodologically, we take Architectural Design Rewriting (ADR) as our architectural description language. ADR is a rule-based formal framework for modelling (the evolution of) software architectures. We equip the reconfiguration rules of an ADR architecture with pre- and post-conditions expressed in a simple logic; a pre-condition constrains the applicability of a rule while a post-condition specifies the properties expected of the resulting graphs. We give an algorithm to compute the weakest pre-condition out of a rule and its post-condition. Furthermore, we propose a monitoring mechanism for recording the evolution of systems after certain computations, maintaining the history in a tree-like structure. The hierarchical nature of ADR allows us to take full advantage of the tree-like structure of the monitoring mechanism. We exploit this mechanism to formally define new rewriting mechanisms for ADR reconfiguration rules. Also, by monitoring the evolution we propose a way of identifying which part of a system has been affected when unexpected run-time behaviours emerge. Moreover, we propose a methodology that allows us to select which rules can be applied at the architectural level to reconfigure a system so to regain its architectural style when it becomes compromised by unexpected run-time reconfigurations.
Acknowledgements

Wow, what a journey and so many people I have to thank. With no hesitation, the first person to thank is Emilio Tuosto. It has been such a rewarding experience to be supervised by someone so inspirational and hard working. This PhD trip has been designed-by-contract and the weakest pre-condition to my successful thesis submission has been your invaluable help both as a supervisor but also as a friend. Formally, given a production $p = \text{PhD Trip}$ and a post-condition $\varphi = \text{Success}$, the weakest pre-condition computed is

$$W(p, \varphi) = \text{Emilio for Supervisor}.$$ 

Thank you so much!

I am also extremely grateful to the help and guidance of Reiko Heckel whose great experience and wide background made our conversations and yearly vivas so interesting. Despite his very busy schedule he was always available whenever I required his assistance. I would also like to thank my examiners Andrew Fish and Stephan Reiff-Marganiec for their amazing suggestions and for the ultra long viva discussions we had.

A huge thank you to everyone in the Department of Computer Science here at the University of Leicester, thanks for making this place feel so like home. I can honestly say every single person in here provided me with assistance, advice, friendship and some incredibly good times during my studies. Thanks especially to Artur Boronat, Stuart Kerrigan, Daniela Petrisan, Igor Razgon, Stephan Reiff-Marganiec, Tom Ridge, and Emilio Tuosto who I worked with as a teaching assistant and for being so friendly and understanding during the busy periods of my research.

I would also like to thank my fellow GTAs and PhD colleagues here in Leicester who might not know it but affected my life in so many positive ways. I will fondly remember (and in some cases try to forget) our great parties, the Wimbledon final, our amazing Friday nights for which I missed most of my Saturdays, and their great support especially during this tough sleepless writing up period. Special thanks should go to Julien Lange for being the best PhD brother I could have hoped for. You are
the best bro! Abdulla Al Sokkar for being a generally chilled-out guy, Gabriela and Octavian for being so supportive during the hard writing up period, Stelios for being such a great neighbour, and also Andrea Vandin, and Alceste Scalas whose visit here in Leicester was brief but enough to set the pillars for the new revolutionary BEER conference.

The continuous support and encouragement from all my good friends back home kept me going through some tough and frustrating times, and I would very much like to acknowledge Skevi, Stalo, Xenia, John(s), Nearchos, Peter, Prokopis, and Stavros. A souvla is in order guys.

To my parents and grandmother back in Cyprus, I would like to say thank you, thank you, thank you sooo much! You definitely deserve this more than I do. To Andreas, my brother thank you for being my best friend in both the ups and the downs of my entire life.

Finally, and most importantly, I would like to thank Demetria. Her incredible patience, support, encouragement, and unwavering love were the solid foundations the past few years of my life have been built on. You were the fuel that kept me going.
CONTENTS

List of Figures vi

1 Introduction 1
   1.1 Problem and Motivation ................................. 1
   1.2 Proposed Solution ...................................... 5
       1.2.1 Contributions Outline ............................. 9
   1.3 Thesis Outline .......................................... 10
       1.3.1 Thesis Road-map .................................... 11

2 Related Work 13
   2.1 Software Architectures ................................. 14
   2.2 Architectural Description Languages .................. 15
   2.3 Architectural Design Rewriting ........................ 18
   2.4 Design-by-Contract for Software Architectures ........ 20
   2.5 Self-adaptive Systems ................................. 23

3 Background Information 26
   3.1 A realistic Scenario .................................... 26
   3.2 An overview of Architectural Design Rewriting ......... 28
       3.2.1 ADR Graphs ......................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.2</td>
<td>Typing in ADR</td>
<td>31</td>
</tr>
<tr>
<td>3.2.3</td>
<td>Design Productions</td>
<td>33</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Run-time reconfigurations</td>
<td>37</td>
</tr>
<tr>
<td>4</td>
<td>A variant of ADR</td>
<td>41</td>
</tr>
<tr>
<td>4.1</td>
<td>Typing Graph and Replaceability Mapping</td>
<td>42</td>
</tr>
<tr>
<td>4.2</td>
<td>Tracking ADR Architectural Reconfigurations</td>
<td>44</td>
</tr>
<tr>
<td>4.3</td>
<td>A New Rewriting Mechanism</td>
<td>49</td>
</tr>
<tr>
<td>4.3.1</td>
<td>Applying the new rewriting mechanism</td>
<td>53</td>
</tr>
<tr>
<td>5</td>
<td>Introducing AcDR</td>
<td>58</td>
</tr>
<tr>
<td>5.1</td>
<td>A logic for ADR</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Design-by-Contract for ADR</td>
<td>64</td>
</tr>
<tr>
<td>5.3</td>
<td>Extracting contracts for ADR productions</td>
<td>67</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Prelude</td>
<td>67</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Auxiliary mappings</td>
<td>70</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Weakest pre-condition algorithm</td>
<td>79</td>
</tr>
<tr>
<td>5.4</td>
<td>Correctness of the algorithm</td>
<td>88</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Proofs for Theorem 1</td>
<td>88</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Proofs for Theorem 2</td>
<td>98</td>
</tr>
<tr>
<td>5.5</td>
<td>AcDR in Action</td>
<td>101</td>
</tr>
<tr>
<td>6</td>
<td>Enforcing Architectural Styles</td>
<td>108</td>
</tr>
<tr>
<td>6.1</td>
<td>Methodology for recovering the architectural style</td>
<td>109</td>
</tr>
<tr>
<td>6.2</td>
<td>Flight Booking System</td>
<td>113</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Enforcing the Style with a Simple Invariant</td>
<td>115</td>
</tr>
<tr>
<td>6.2.2</td>
<td>A more Complicated Invariant</td>
<td>119</td>
</tr>
<tr>
<td>6.3</td>
<td>Adaptive Network Connectivity on Ships</td>
<td>120</td>
</tr>
<tr>
<td>6.3.1</td>
<td>Enforcing the Style Using the Tracking Tree</td>
<td>124</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion</td>
<td>129</td>
</tr>
<tr>
<td>7.1</td>
<td>Contributions</td>
<td>129</td>
</tr>
</tbody>
</table>
CONTENTS

7.2 Evaluation of Contribution ........................................ 132
7.3 Future Work ...................................................... 133

Bibliography ........................................................... 137
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>System’s Development Cycle</td>
<td>2</td>
</tr>
<tr>
<td>3.1</td>
<td>Maritime wireless mesh infrastructure</td>
<td>28</td>
</tr>
<tr>
<td>3.2</td>
<td>Architectural Elements</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>ADR Type Graph</td>
<td>32</td>
</tr>
<tr>
<td>4.1</td>
<td>Term to graph morphisms and maps (cf. Definition 10)</td>
<td>50</td>
</tr>
<tr>
<td>4.2</td>
<td>ADR Graph</td>
<td>53</td>
</tr>
<tr>
<td>4.3</td>
<td>Tracking tree and environment associated to the Graph in Figure 4.2</td>
<td>54</td>
</tr>
<tr>
<td>5.1</td>
<td>Asserted design productions</td>
<td>65</td>
</tr>
<tr>
<td>5.2</td>
<td>Auxiliary map - Case Analysis</td>
<td>73</td>
</tr>
<tr>
<td>5.3</td>
<td>Type Graph where $e \in {\text{O, RO}}$ and $e' \in {\text{URO, \ldots}}$</td>
<td>102</td>
</tr>
<tr>
<td>5.4</td>
<td>Design Productions</td>
<td>103</td>
</tr>
<tr>
<td>6.1</td>
<td>Flight Booking Type Graph</td>
<td>113</td>
</tr>
<tr>
<td>6.2</td>
<td>Flight booking scenario</td>
<td>114</td>
</tr>
<tr>
<td>6.3</td>
<td>Ships disconnecting scenario</td>
<td>123</td>
</tr>
<tr>
<td>6.4</td>
<td>Enforcing the Ship Connectivity Style</td>
<td>124</td>
</tr>
<tr>
<td>6.5</td>
<td>Tracking forest $T_2$ and the corresponding environment $T_2$</td>
<td>126</td>
</tr>
<tr>
<td>1</td>
<td>Definition ((Hyper)graphs [18])</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Definition (Graph Morphisms [18])</td>
<td>31</td>
</tr>
<tr>
<td>3</td>
<td>Definition (ADR graphs)</td>
<td>31</td>
</tr>
<tr>
<td>4</td>
<td>Definition (Typed Graph morphisms)</td>
<td>33</td>
</tr>
<tr>
<td>5</td>
<td>Definition (Productions)</td>
<td>34</td>
</tr>
<tr>
<td>6</td>
<td>Definition (ADR Replaceability Mapping)</td>
<td>43</td>
</tr>
<tr>
<td>7</td>
<td>Definition (Tracking productions)</td>
<td>45</td>
</tr>
<tr>
<td>8</td>
<td>Definition (Bow tie relation)</td>
<td>49</td>
</tr>
<tr>
<td>9</td>
<td>Definition (Tree of a variable)</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>Definition (From Term to Graph)</td>
<td>51</td>
</tr>
<tr>
<td>11</td>
<td>Definition (Applying reconfiguration rules)</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>Definition (ADR logic)</td>
<td>59</td>
</tr>
<tr>
<td>13</td>
<td>Definition (Satisfaction relation)</td>
<td>60</td>
</tr>
<tr>
<td>14</td>
<td>Definition (Asserted productions)</td>
<td>64</td>
</tr>
<tr>
<td>15</td>
<td>Definition (Applying asserted productions)</td>
<td>65</td>
</tr>
<tr>
<td>16</td>
<td>Definition (Environment)</td>
<td>71</td>
</tr>
<tr>
<td>17</td>
<td>Definition (Auxiliary map)</td>
<td>74</td>
</tr>
<tr>
<td>18</td>
<td>Definition (False Conditions)</td>
<td>77</td>
</tr>
<tr>
<td>19</td>
<td>Definition (Transformer ( wd ))</td>
<td>80</td>
</tr>
<tr>
<td>20</td>
<td>Definition (( wp ) predicate transformer)</td>
<td>82</td>
</tr>
<tr>
<td>21</td>
<td>Definition (Formulae in context)</td>
<td>86</td>
</tr>
<tr>
<td>22</td>
<td>Definition (Parsing)</td>
<td>112</td>
</tr>
</tbody>
</table>
CHAPTER 1

1.1 Problem and Motivation

Modern applications are very rarely developed as “stand-alone” software; as a matter of fact, even simple applications are nowadays open in the sense that they are typically able to connect and/or be integrated with other applications such as those in service-oriented or cloud computing. The complexity of large scale software and distributed systems combined with the openness characteristics of modern applications [20], calls for rigorous methods at very early stages of their development. Because of their very interactive nature, such systems are starting to be studied under new angles [42]. For instance, software needs to adapt to the (often unpredictable) changes of the (virtual and physical) environment they operate in. The use of high-level designs and software architectures assumes a prominent role to guarantee quality and correctness of systems.
To support architecture-based development, formal modelling notations and tools for analysis and development that operate on architectural specifications are needed.

Figure 1.1 represents the four main phases in the development cycle of software and distributed systems. Initially (phase 1), the software engineer needs to identify the informal specifications of a system, for instance using Unified Modelling Language (UML) [35]. After the informal specifications the engineer has the option of formally designing the system (phase 2). The formal design of a system starts by specifying the architectural aspects which (as shown in Figure 1.1) have an impact on the system’s behaviour and therefore later on its coordination. Next (phase 3), the engineer can specify models for controlling different kinds of communications between participants; for instance by giving a formal model in a process algebra (e.g. the $\pi$ - calculus [66])
that can later be translated into a program (e.g. Java or Haskell) (phase 4). Phases 2 and 3 are optional when developing a system and including them in the formal development cycle is of paramount importance when dealing with systems with multiple concerns, requiring dynamic communication between various components [63]. As explained in Chapter 2, phase 2 can be used to precisely define the structural and behavioural aspects of systems and the rules associated to the way the systems can adapt during their execution.

In order to formally analyse and design the architectures of dynamic systems we consider Architectural Description Languages (ADLs) (cf. Section 2.2). ADLs used to design modern autonomic systems [48] like for instance ACME [39], Darwin [57], ADR [18] (cf. Chapter 2), have to be able to guarantee software quality, and correctness by being able to adapt from their initial designs, and also predict the possible problems that could arise during the execution of such systems. In fact, it is necessary for an ADL to guarantee some degree of self-adaptation on applications that can be dynamically composed, and compromised by run-time changes. We will not make an attempt to define the concept of autonomic or self-adaptive software, but intuitively, autonomic systems can be thought of as those systems that can adapt to dynamic changes of the (physical and/or execution) environment into which they act. This intuition is hard to transfer to software (architectures) as argued in [13]. For our purposes it is enough to consider applications that may run into an array of unexpected states which is so wide, it would be impractical to capture in a complete specification [19].

Furthermore, openness magnifies even more the complexity of such software. Open applications are subject to unexpected reconfigurations that may hinder their execution
and drive computations into erroneous states in an unanticipated manner. Not only it is crucial to detect those states of the computation at an architectural level (cf. Chapter 5), but it is necessary to design applications able to re-establish correct configurations in presence of misbehaviour. For example, the reaction to the failure of a server $S$, may redirect the requests of its clients to another server $S'$.

Another problem that can arise in those cases is that the run-time reconfigurations may compromise the alignment with the expected abstract architecture. In the client-server scenario mentioned above, the choice of $S'$ may cause the violation of some architectural constraints designed e.g. to balance the load.

Software architectures specify the architectural style of applications; their structure and interconnections. Ordinary computation can change the state, but they are very rarely allowed to modify the architectural style. Formal approaches aim to devise robust engineering practises to form reliable software products to mitigate the issues described above. Arguably, those approaches focus on software behaviour; correctness and efficiency of software play in fact a crucial role not only in critical systems but also in daily-life applications. In the design phase, semi-formal methods are typically adopted; as an example, the use of modelling languages is combined with design patterns [29] to devise a model that can be checked. This approach may involve formal techniques (e.g., type or model checking) to guarantee properties of applications while non-formal techniques (or tools not supported by formal approaches) are typically used to tackle architectural design aspects. Our research agenda envisages the combination of those approaches with techniques to address the issues above at the design level. We believe that a rigorous treatment at the design level would allow one to identify and solve many
problems that are currently tackled only by inspecting or testing code.

This is why the work we present in this thesis to lift Design by Contract (DbC) techniques to the architectural level (phase 2 in Figure 1.1). DbC is currently being used for formal distribution models, such as [9] (phase 3 in Figure 1.1), and also for the implementation part of a system’s development cycle [65] (phase 4 in Figure 1.1). Even though some authors [28] used DbC at design level, it is only regarding the computational aspects of systems. DbC for structural and architectural aspects is still something to be studied.

1.2 Proposed Solution

Our approach hinges on a formal language for specifying software architectures, their refinements, and their style. Methodologically, we adopt ADR [18] as our architectural description language.

ADR is a rule-based formal framework for modelling (the evolution of) software architectures. As surveyed in Section 3.2, ADR models systems as (hyper)graphs that is a set of (hyper)edges sharing some nodes; respectively, edges represent components (at some level of abstraction) while nodes represent interaction ports. Also, ADR features refinement rules (known as design productions) of the form

\[ L \rightarrow R \]

where \( L \) is a (hyper)edge and \( R \) a (hyper)graph meant to replace \( L \) with \( R \) within a given graph. These design productions define the architectural style (cf. Section 3.2)
as presented in (1.1) below. As also discussed in Chapter 2 the architectural style of a system plays a crucial role in software architectures [76].

\[
\text{architectural style} = \text{production rules} \tag{1.1}
\]

Furthermore, in ADR, a system corresponds to a configuration of elements (i.e. nodes and edges) that can be related to the architecture graph components and expected to respect the architectural style specified by the refinement rules. Such elements can interact through their connections according to run-time interactions (run-time reconfigurations) not represented at the architectural level. As a matter of fact, ADR features complex reconfigurations that cannot be captured by production rules. Such complex reconfigurations can be envisaged as a model of run-time evolution of systems that describe what complex rearrangements could happen during execution. This is technically done by specifying term rewriting rules in an algebra where terms are interpreted as proofs of the style of graphs. In Section 3.2 we present a streamlined version of [18] for a better understanding of the technicalities around ADR.

This thesis presents a technical development of ADR [69, 70, 71] called AcDR after Architectural contract-based Design Rewriting. AcDR provides a formal framework to tackle the architectural/structural aspects of system’s design and allow systems to adapt to the various configurations that violate its architectural style. The methodology presented in Chapter 6 proposes a solution to the problems identified in Section 1.1 and uses high-level designs of software architectures to automatically compute reconfigurations that restore desirable architectural properties (expressed as logical invariants).

Before the application of the methodology in Chapter 6 though, certain adaptations
had to be introduced to ADR. Initially, we introduce a monitoring mechanism that fully exploits the features of ADR. The refinement we propose in Chapter 4 regards the rewriting mechanism of ADR and, more importantly, its underlying monitoring capability. Indeed, as observed in [18], a distinctive aspect of ADR is that it features the canonical view of software architectures in terms of connected architectural elements as well as a hierarchical view of software architectures that is paramount in the design phase. Although [18] advocates the use of the architectural view as a useful mechanism to be exploited in complex reconfigurations, no actual formalisation has been provided on how this could be achieved and the parsing features of ADR were only sketched in [18]. More precisely, in Chapter 4 we start by changing the original rewriting mechanism of ADR. In AcDR, we generalise the original rewriting mechanism of ADR by featuring a more liberal way of specifying which edges can be rewritten (in ADR the type of an edge determines if it is replaceable or not). Besides some simplification in the technical presentation of ADR (which is now more uniform), such generalisation brings in extra flexibility. In fact, a replaceable edge can be refined by introducing new versions of a rule that differs only for the “replaceability” of some edges. Also, we equip AcDR with a monitoring mechanism that is also exploited to define an efficient parsing of ADR graphs. Our monitoring mechanism keeps track of the application of reconfiguration rules and uses such information when the graph has to be parsed (to identify the part that violates a style). The application of a reconfiguration rule affects such information that need to be updated accordingly.

In Chapter 5 we propose a design-by-contract (DbC) approach to tackle the problems discussed in Section 1.1. Initially, we introduce a simple propositional logic tailored
on ADR that allows us to check various properties on ADR graphs. A technical contribution of Chapter 5 is to generalise ADR with asserted productions, that is refinement rules of the form

\[ \{\psi\}L \rightarrow R\{\varphi\} \]  

(1.2)

where the intuition of (1.2) is that the replacement of \( L \) with \( R \) in a graph that satisfy the pre-condition \( \psi \), yields a graph satisfying the post-condition \( \varphi \). In ADR [18], architectural styles are formalised in terms of productions that describe the legal configurations of systems. We generalise this by envisaging architectural styles as set of productions together with invariants (expressed as formulae of our logic) which can be thought of as contracts that architectures have to abide by. Pre- and post-conditions are expressed in a simple logic for hyper-graphs (cf. Section 5.1). Intuitively, (1.1) now becomes

\[ \text{architectural style} = \text{production rules} + \text{reconfiguration rules} + \text{invariants} \]  

(1.3)

where an invariant is the property the designer requires of the application.

Our main contribution in Chapter 5 is an algorithm that computes a weakest precondition \( \psi \) out of a post-condition \( \varphi \) and a design production. We prove a theorem that guarantees that the application of the rule to a configuration satisfying \( \psi \) yields a configuration satisfying the post-condition \( \varphi \). This algorithm can be used to compute a reconfiguration if the current configuration violates the invariant. An example is given in Section 5.5 where we use asserted productions and the algorithm to distill weakest pre-conditions on them to model the semantic aspects of the Java remote method

1.2. Proposed Solution 8
invocation mechanism.

Finally, we devise a methodology in Chapter 6 that uses the new AcDR properties to identify when a reconfiguration compromised a system’s architectural style and also provide strategies for reestablishing its style.

### 1.2.1 Contributions Outline

The thesis organically presents results that have been published as reported below.

**An extension to ADR** (cf. Chapter 4) [71]

1. New Replaceability Mapping for ADR components

2. Monitoring Mechanism for ADR configurations (both design time and run-time)

3. New rewriting mechanism for ADR

**AcDR** (cf. Chapter 5) [69, 70]

1. Simple propositional logic for specifying graph invariants

2. Introduction of DbC for ADR

3. A weakest pre-condition algorithm tailored for ADR design productions

4. Interesting theorems such as the fact that any graph satisfying the weakest pre-condition of a design production is guaranteed to satisfy its post-condition after the application of the production.

**Re-enforcing architectural style of systems** (cf. Chapter 6) [69, 70, 71]
1. Identification of style violation

2. Utilisation of new rewriting mechanism to identify which part of the system has been compromised

3. Proposition of various ways to re-enforce the architectural style.

In the thesis we use a new running example on real-time communication of applications on-board ships (cf. Section 3.1) which has not been used in any of our published papers so far.

1.3 Thesis Outline

We structure the rest of the thesis as follows:

In Chapter 2 we discuss the related work. This includes work that inspired us to focus on specific properties and frameworks and also how other attempts that provide a formal approach to adaptive software architectures compares to our work.

Chapter 3 provides some background information required for a better understanding of the rest of the thesis. This includes an introduction to a running example used throughout the thesis, and also a short overview of ADR as it was originally presented in [18].

Chapter 4 introduces a variant of ADR and defines our monitoring approach as well as the new rewriting mechanism hinging the monitoring mechanism.

Chapter 5 introduces AcDR by initially presenting a simple logic for ADR in Section 5.1 and then our design-by-contact approach on design productions in Section 5.2.
In Section 5.3 we introduce and define the weakest pre-condition algorithm, and in Section 5.4 we show the proofs of the theorems presented in Section 5.3. In Section 5.5 we illustrate our approach in how asserted productions can guarantee design properties by considering a simplified version of Java’s remote method invocation (JRMI) [44].

In Chapter 6 we describe a methodology that relies on the new rewriting mechanism presented in Chapter 4 and the weakest pre-condition algorithm in Section 5.3 to recover architecture styles compromised by run-time reconfigurations. An application of the methodology is given in Section 6.2 using a flight booking system as an example and also in Section 6.3 using our running example.

Finally, Chapter 7 presents some concluding remarks and future work related to the thesis.

1.3.1 Thesis Road-map

For the readers’ convenience we provide a flexible road-map showing the dependencies between each chapter of the thesis. Going from left to right, we advise the readers to read the introduction in order to get a general idea of the motivations and the contributions of our work. The readers can then skip Chapter 2 if they are not interest about any other similar approaches, but in order understand the technical contributions of the rest of the thesis Chapter 3 is very important. Chapters 4, and 5, are self contained and do not depend on each other, but for the readers to grasp the recovering mechanism introduced in Chapter 6 they have to first examine Chapters 4 and 5. Finally, Chapter 7 depends on the rest of the chapters as it provides interesting future
directions related to the technicalities presented.
In this chapter we discuss related work. This includes work that inspired us to focus on specific properties and frameworks and also how other attempts to provide a formal approach to adaptive software architectures compares to our work. We categorise the related work into:

1. Software Architectures (cf. Section 2.1)
2. Architectural Description Languages (cf. Section 2.2)
3. Architectural Design Rewriting (cf. Section 2.3)
4. Design-by-Contract for Software Architectures (cf. Section 2.4)
5. Self-adaptive Systems (cf. Section 2.5)
CHAPTER 2. RELATED WORK

2.1 Software Architectures

Software architectures are very important when dealing with systems with multiple concerns requiring dynamic communication between various components [63]. The use of high-level designs and software architectures assumes a prominent role to guarantee software quality and correctness [43, 40, 68] and can be used to reach to appropriate agreements concerning the structural and behavioural aspects of systems and the rules associated to it. Furthermore, they can provide a solid way to formally document various dynamic systems both in terms of their functional and non functional aspects [5].

More formal and accurate paradigms are of great importance for handling and coordinating the various components during the design phase of large complex systems. As explained in several attempts [63, 68] to classify software architectures, two factors that stand out are dynamicity, and flexibility. Systems nowadays should be able to evolve and adapt to new uses very frequently. More precisely systems should be able to adapt to two architectural categories of problems:

- **Architectural erosion** occurs by consistent violations of the architecture [68]. These violations occur due to software ageing, damaging a software’s structural integrity and affecting its design principles [24, 4]. If such violations keep piling up and are not dealt in time this could lead to disastrous results.

- **Architectural drift** is due to inconsistencies between the intended architecture and the actual architecture. Violations on the intended architecture may not be considered violations on the actual architectures. Such problems can possibly lead to the lack of conformance and clarity of the system, making it harder to identify
violations of the obscured architecture [68].

Software architectures need to support refinement at the level of components and connectors with features for their incremental addition, removal, replacement, and reconnection in a configuration [62]. Allowing though, these amendments to occur freely could be problematic for large open/autonomic systems [58, 63]. For instance, critical issues arise, in the coordination of large projects, when software is developed by many teams of programmers (possibly working distributively), or it has to be dynamically combined [6, 53]. For instance, it is well known that many software projects are cancelled before completion due to architectural drift and sometimes maintenance may easily reach 70%-90% of the total cost of software [51].

Designing correct software architectures requires sophisticated analysis to make sure it satisfies correctness criteria [67]. It is very important to consider modular [75], maintainable, customizable software architectures that can adapt to new uses throughout their life-cycle.

2.2 Architectural Description Languages

Architecture Description Languages (ADLs) emerged as a promising way to formally describe some essential features of software architectures. It is though, almost generally accepted in the software engineering community that a standard architecture specification (description) should be comprised by three parts, namely components, connectors, and architectural configurations [62]. There exists a number of ADLs, some of which are classified and compared in [62] and each ADL has its own features addressing an
architecture from its own perspective. What is clear from the literature of various sur-
veys [22, 26, 41, 52, 62, 72, 73] is that apart from the components that comprise an
architecture, there is no conformity between researchers in what architectural aspects
an ADL should model. Even though, our goal in this thesis is not to provide a defini-
tion on what ADLs should express, we believe that system design should be done in a
formal and precise way, providing details about all the components of the system, the
way they can connect to each other and the constraints behind their interconnections.
We also argue that these constraints should be monitored and well defined during not
only the generation of the architecture but also its execution. Furthermore, we identify
some of the main properties ADLs should have through [22, 26, 41, 52, 62, 72, 73] and
pinpoint the ones we consider to be indispensable in our opinion.

- **Understandable Specifications:** On top of all the other properties, software
architectures are the means of communications between system developers and
system designers. As correctly specified in [61] ADLs should make the structure
of a system easy to understand without the need to study every single component
of the system. Some ADLs, like Darwin [57], MetaH [8], and Rapide [56] overload
connectors with a lot of information and parameters making it very hard to follow
and understand. Despite their good tool support for automating some of the
developers load it is very important for an ADL to offer a clear understanding of
the basic structural properties of the system.

- **Scalability and Hierarchical style:** In our opinion scalability and hierarchical
design are two properties that should coincide. ADLs must be able to support and
provide design mechanism that can handle large systems that might grow through-
out their life cycle. It is therefore paramount for an ADL to offer a hierarchical flavour to the architecture and like with ADR [18], Aesop [37], and C2 [60] allow systems to be viewed from different levels. By abstracting away complicated structures it allows for a more efficient way of designing systems in a modular fashion. For instance, large systems require groups of designers and engineers to be able to work independently on different components and reconfigurations of the system and the hierarchical nature offered by some ADLs encourages these actions since composition of components [67] can later be done more efficiently when designers are able to abstract away from their local constraints.

- **Design Time Evolution:** Refers to the architecture generation and the offline changes of an architecture. ADLs need to support evolution at the level of components and connectors and be able to specify and control the way components interact with each other at design time [70, 71]. This type of specifications are not important just for the generation of the architecture but also due to the fact that they set the *architectural style* that should be reflected in the run time reconfigurations of the system [30].

- **Run Time Configuration:** ADLs should be able to model configurations regarding system’s execution. These configurations have to follow the specifications and the architectural style set earlier (see design time evolution). For example, component interaction cannot be performed in a way that violates the standards set by the architecture.

- **Adaptability** As argued in the properties specified so far it is very important
for an ADL to have the capability to model architecture evolution both for the
generation level but also for the execution [70, 71]. However, being able to model
dynamic changes does not guarantee that the obtained system will preserve the
architectural style set by the ADL. Furthermore, it is generally accepted that one
cannot predict every misbehaviour that will compromise a system in terms of its
architectural style [63]. ADLs though should be able to provide strategies for
recovering from such stages especially when dealing with critical systems [10].

These properties were used as a road map throughout this thesis in our aim to provide
a framework that satisfies all the special characteristics ADLs should have.

2.3 Architectural Design Rewriting

We consider a specific ADL called Architectural Design Rewriting (ADR for short) [18]
and in the context of this thesis present how we evolved ADR to produce a more fully
fledged language called AcDR [70, 71]. ADR has been inspired by various approaches
ranging from process calculi to deal with reconfigurable component based architectures
[2], graphical models of process algebras [31], and graph-based approaches to architec-
tural styles [64].

We have chosen ADR due to its key features namely, its algebraic presentation,
its hierarchical design, its inductively defined rewrite rules (c.f. Chapter 3), and also
because it is the perfect candidate when dealing with reconfigurable software architec-
tures. Architectures are designed hierarchically by a set of edge replacement rules called
design productions which allow top-down and bottom-up application (c.f. Chapter 4),
as well as well-formed (in terms of style) composition of architectures. While design productions have a graphical notation they also have an algebraic representation for setting the style of the system. The idea is to encode productions as terms and utilise those terms during execution for setting the run-time reconfigurations. This gives a great advantage as it provides a guarantee that reconfigurations are style-preserving by construction.

The main advantages of ADR and the reason we selected ADR is that it already fulfils up to a certain extent the properties identified earlier in Section 2.2. By internalising all the parameters defining the architectural style within its algebraic form and also by taking advantage of its hierarchical structure, ADR can offer the developer an architectural description in various levels of abstraction.

ADR was initially presented in [18] but since then it was used in different ways and adapted to suit various scenarios [12, 14, 16, 17]. For instance, in [14], ADR was used as a formal model for architectural, and business design. The authors showed how ADR can be used for formalising crucial aspects of the UML4SOA [34] modelling language; a UML profile for designing service-oriented software. UML4SOA requires suitable formalisms to constrain the possible evolutions of a system that may compromise its architectural style. ADR’s style preserving nature offers the perfect candidate and its typing mechanism and design productions, can support the architectural constraints of UML4SOA. In [17] ADR was applied to the SENSORIA Referencing Modelling Language [32] showing its expressiveness and wide applicability by taking advantage of ADRs hierarchical and inductive features to model the complicated reconfigurations required in a service oriented environment. A variant of ADR called Hierarchical Design
Rewriting (HDR) is proposed in [16]. The authors introduce derived operators to allow the representation of partial architectural information. The main contribution of [16] is the prototypical implementation of ADR in Maude [21] proving that ADR does not only offer solid theoretical foundations but also a tool-supported framework for the design and analysis of software architectures.

2.4 Design-by-Contract for Software Architectures

Design-by-contract (DbC) was initially introduced by Meyer [65] for implementation purposes to discourage the use of defensive programming; redundant checks inside the code e.g. unwanted if statements. DbC provides a systematic approach to deal with abnormal and unexpected cases via the use of contracts. Such contracts are attached to the desired operations in software and include pre- and post-conditions, where a pre-condition should hold before an operation is executed and the post-condition should hold after the operation is executed.

We advocate the use of DbC [65] to formally guarantee properties of software designs so to enable developers to specify and check properties at design time rather than enforcing them at later stages. Methodologically, we adopt ADR [18] as our architectural description language where, as surveyed in Section 3.2, architectural refinement rules have the form $L \rightarrow R$ where $L$ is an edge and $R$ a (hyper) graph meant to replace $L$ with $R$ within a given graph.

In Chapter 5 we generalise a variant of ADR (c.f. Chapter 4) with asserted productions, that is refinement rules with pre- and post-conditions. Roughly, asserted
productions take the form

$$\{\psi\}L \rightarrow R\{\varphi\} \quad \text{where } \psi \text{ and } \varphi \text{ are the pre- and post-conditions} \quad (2.1)$$

As also explained in Chapter 1, the intuition is that (2.1) can be applied only to graphs satisfying $\psi$ to obtain a graph satisfying $\varphi$. The application of an asserted production to a graph satisfying $\psi$ though does not necessarily yield a graph satisfying $\varphi$ (this can be trivially noted by taking a production with $\bot$ as post-condition). We therefore present in Chapter 5, an algorithm that computes the weakest pre-condition given a post-condition and a production. The algorithm guarantees that the application of a production $p$ to a graph satisfying its weakest pre-condition will yield a graph that is guaranteed to satisfy its post-condition.

Formal approaches based on architectural styles to control architectural reconfigurations have been proposed, amongst others, in [3, 64, 47]. In those proposals reconfigurations are typically applied uniformly across the design. In [3] separate component specifications are used to describe the behavioural aspects of a component before and after a reconfiguration. They then encapsulate the dynamic changes caused by the reconfiguration in the specification of the connectors, achieving separation of concerns. In [64] graph grammars and hyper-edge replacements are used to represent styles in terms of graph configurations freely generated by some productions. It is therefore not easy to specify conditions to extract subsets of such graph-languages. Similar to [64], in [47] they also define architecture styles using hyper-edge context-free graph grammars. In this case context-free grammars limit the expressiveness of architecture styles.

Our work mitigates this effect by means of asserted productions that provide a
finer control on the applicability conditions as done in other graph-transformation approaches. For instance, our approach is similar to the one in [45] where graph programs are extended to programs over high-level rules with application conditions; on such programs weakest pre-conditions can be defined automatically. Nevertheless, [45] aims at verifying computational properties of systems rather than architectural ones and does that in a different way only after generating the various state systems. Another example is in [28], where contracts are used in the design phase and are used to generate JML assertions; this is a rather interesting approach as it abstracts low level computational aspects at design level through DbC to monitor the execution of systems. In [46] pre- and post-conditions are used (as goals) to specify requirements for service composition. The authors propose a notion of partial matching to compute which part of a goal is satisfied by the composition and what is the remainder. In [33], constraints that specify pre- and post-conditions for the behavioural aspects of software components are considered. The authors propose a formalism that identifies recurring specification patterns and automatically generates visual constraints through those patterns. The pattern mechanism presented automates the manual and human dependent contract specification and the visual constraints can serve many of the ADL properties we have set in Section 2.2 (especially Understandable Specifications).

DbC has also been used in other levels of a systems life cycle. [9] introduced an idea similar to what we introduce for the architectural aspects of a system but for the lower level specifications of it. They propose a DbC approach using assertions for the specification, verification, validation and monitoring of distributed multi-party protocols, using π-calculus. Also [55] examines how DbC techniques can be used to provide precise
specifications needed to enable components to be replaceable and reusable. The paper uses UML to express structural relationships and associates DbC to OCL constraints to express design contracts. In these attempts they provide pre- and post-conditions using OCL for UML diagrams. This approach works in a way that dictates how the system should be developed aiding in its “correct” development. The difference to our approach is that we are dealing with contracts that control the architectural aspects of the design and generation of the system but not its implementation at the moment. This opens new future routes though, discussed in Chapter 7, since our work enables the study of a formal link between design-level contracts and lower-level ones (like those in [55, 9] for instance).

2.5 Self-adaptive Systems

In Chapter 6 we introduce a methodology that utilises the work presented in Chapters 4 and 5 to provide strategies for recovering systems when their architectural style has been compromised.

Different approaches to specify self-managing systems are surveyed in [10]. The authors group the different approaches according to their ability to select different reconfigurations that should occur to re-establish a correct state. They present three types of selections, called pre-defined selection (a reconfiguration is chosen prior to the execution based on a pre-defined selection), constrained selection from a pre-defined set (a reconfiguration designed for the given situation is chosen) and unconstrained selection (unconstrained choice regarding the appropriate change to make). All the approaches
presented in the survey lie in either of the former two categories and according to [10], none of the approaches surveyed fall in the unconstrained selection category. Our approach does not lie in the pre-defined nor in constrained selection categories. It is not clear to us if our approach can be considered an unconstrained selection. In fact, we do not choose the reconfigurations to apply according to the misbehaviour expected at run time. Instead we use our weakest pre-condition algorithm to identify which of the existing configurations (not designed for the specific violation) can re-establish the architectural style of our system. We remark that most of the rules given at design time typically are meant to specify the architectural style of a system, not its misbehaviour (for instance, in ADR this might be addressed with reconfiguration rules rather than productions). However, even if some productions were introduced to tackle (or prevent) some misbehaviour, our approach enables such rules to be used also for unexpected violations.

In [38] constraints on the architecture are used to guarantee invariants of systems. More precisely, reconfigurations can occur only if such constraints are not violated. This is not always realistic in open systems; therefore they do not impose limitations on runtime reconfigurations and search for new reconfigurations that can lead the system into a desired state.

In [23] an assume-guarantee mechanism is adopted to provide a learning algorithm which provides an assumption satisfying a sufficient condition in order for the component to guarantee the given invariant. This is achieved by model checking every component of the system against an invariant. This is similar to the weakest pre-condition we present in this thesis but instead of computing the weakest assumption
for every component of the system we compute the weakest pre-condition for every design production. We can later use our algorithm for applying the methodology described in Chapter 6 for identifying the possible design production(s) (if any) to aid in fixing the architectural violation of the system.

In [7] the authors present an approach for designing safe systems by inspecting whether certain reconfigurations can lead to invalid graphs that represent invalid systems. This is achieved by verifying that the backward application of reconfigurations to a forbidden graph pattern cannot lead to a graph pattern representing a safe system (a set of forbidden graph patterns model an invariant). This is an interesting approach as it can provide a safe system in the sense that it cannot lead to a state that violates a structural invariant by the use of reconfigurations but it is very complex to handle unexpected system failures.

In [27] self-healing systems are modelled by specifying different types of rules; for the ideal system behaviour, for different predictable failures and for fixing the different failures identified earlier. This approach is different to what we propose in this thesis as they design the rules according to the misbehaviour they expect at run time and do not necessarily handle unexpected failures or changes of the system.
CHAPTER 3

BACKGROUND INFORMATION

This chapter gives an overview of all the prerequisites and background information required for understanding the concepts that will be presented in the rest of the thesis. Section 3.1 presents a running scenario that will be used throughout the thesis to give an intuitive understanding of our contributions. In Section 3.2 we give an overview of ADR as it was originally presented in [18].

3.1 A realistic Scenario

We start off by presenting a scenario used throughout the thesis to provide a better understanding of the contributions of our work.

We borrow our scenario from [54] where an IP-based network connectivity is proposed to support real-time communication of applications on-board ships. The motivation behind [54] is to (partially) replace, when possible, the very expensive and high-
latency satellite communications commonly deployed in the maritime industry. More precisely, this is achieved by placing long range wireless access points at the shores and allowing ships to connect either directly to the access points or to other connected ships close by. The fact the wireless network technologies have evolved so rapidly over the past years make this attempt feasible. As mentioned in [54], the 802.16 WiMAX standard [59] can allow high speed connections between devices placed 40km apart and the more recent 802.16m WiMAX-2 standard [1] can cover even larger distances.

We have chosen this example as it represents a real life situation but more importantly also because it will allow us to discuss some of the main characteristics of the framework we developed in a realistic setting. As mentioned in Chapter 1, software systems are no longer static and become more and more dynamic; because of their very interactive nature, such systems are starting to be studied under new angles [42]. A key aspect of the chosen scenario is that the network topology may change due to unexpected conditions of the (physical) environment. For instance, a ship may have to adjust its route due to adverse weather conditions. Such changes require the application to be able to adapt itself to the (often unpredictable) changes of the environment it operates in. The term *autonomic computing* has been coined to mark such systems [48], which present new degrees of complexity since they require high levels of flexibility and adaptiveness [49].
3.2 An overview of Architectural Design Rewriting

This section gives a streamlined version of [18]. We initially give a better understanding of what constitutes the architectural style of a system and define ADR graphs in Section 3.2.1. We then present ADR typing in Section 3.2.2 and finally in Sections 3.2.3, and 3.2.4 we present the rewriting mechanism of ADR. More precisely we formally define design productions in Section 3.2.3, and run-time reconfiguration in Section 3.2.4 and try to explain them using simple examples.
CHAPTER 3. BACKGROUND INFORMATION

3.2.1 ADR Graphs

Architectural elements comprise the vocabulary of an architecture and a suitable way to represent architectural elements is using a type graph (cf. Section 3.2.2) with an edge and a node for every type of edge and node element, respectively. In Figure 3.2 and Example 1 we present the architectural elements of our scenario and in Figure 3.3 we present the type graph.

Example 1. In the maritime wireless mesh infrastructure introduced in Section 3.1 the architectural elements are AP after access point, APS for many access points connected together, CS for the connectivity scheduler, ShipON for a connected ship, ShipOFF for a disconnected ship, FleetON for many ships connected, FleetOFF for representing many disconnected ships, and finally Chain for a chain of connected ships.

Example 2 explains the visual notation of ADR’s architectural elements.

Example 2. The visual notation of ADR is summarised using Figure 3.3; edges are drawn as boxes (labelled or decorated with images) which carry information about com-
ponents and nodes as circles (possibly in different colours). Edges and nodes are typed (in the sense of graph typing cf. Section 3.2.2) and edges have an arity which is defined by the number of nodes they are connected to.

When designing a software or distributed system, it is very important to consider the concept of architectural style [74]. Architectural styles allow one (i) to specify (reusable) design patterns, (ii) to confine the parts to be reconfigured, and (iii) to control the architectural changes. Architectural styles will be explained more extensively in Chapter 5 where we introduce invariants to confine even more the style of an architecture and allow the system to be reconfigured under certain criteria.

In ADR the vocabulary of a style is given by the type graph, while the legal interconnections are defined by productions (design-time reconfigurations) presented in Section 3.2.3. Both the vocabulary and a set of rules indicate how components can be legally interconnected (productions) to form the architectural style.

In the following (borrowed from [18]), \( \mathcal{N} \) and \( \mathcal{E} \) are two countably infinite and disjoint sets (of nodes and edges respectively), \( X^* \) is the set of finite lists on a set \( X \), and \( \bar{x} \) ranges over \( X^* \). Also, abusing notation, we sometimes use \( \bar{x} \) to indicate its underlying set of elements.

**Definition 1** ((Hyper)graphs [18]). A (hyper)graph is a tuple \( G = \langle V, E, t \rangle \) where \( V \subseteq \mathcal{N} \) and \( E \subseteq \mathcal{E} \) are finite and \( t : E \rightarrow V^* \) is the tentacle function connecting edges \( e \in E \) to a list of nodes; the arity of \( e \) is the length of \( t(e) \).

The typing of a graph over another is formalised in terms of graph morphisms. Hereafter, we write \( e(\bar{u}) \in G \) for \( e \in E_G \), \( t_G(e) = \bar{u} \subseteq V_G \); also, given a graph \( G \), \( V_G \), \( E_G \), and \( t_G \) respectively denote the nodes, the edges, and the tentacle function of \( G \).
Definition 2 (Graph Morphisms [18]). Let $G$ and $H$ be two graphs, a morphism from $G$ to $H$ is a pair of functions $(\sigma_V : V_G \to V_H, \sigma_E : E_G \to E_H)$ s.t. $\sigma_V$ and $\sigma_E$ preserve the tentacle functions, i.e. $\sigma_V^* \circ t_G = t_H \circ \sigma_E$, where $\sigma_V^*$ is the homomorphic extension of $\sigma_V$ to $V_G^*$.

3.2.2 Typing in ADR

In ADR, graphs are typed over a fixed type graph via typing morphisms. As usual an ADR graph $G$ is typed over a type graph $\Gamma$ through $\tau_G$ if $\tau_G$ is a morphism from $G$ to $\Gamma$. Hereafter, we fix a typed graph $\Gamma$ and tacitly assume that all graphs $G$ are typed over $\Gamma$ via a morphism $\tau_G$. Intuitively, $\Gamma$ yields the vocabulary of the architectural elements to be used in the designs; moreover, $\Gamma$ specifies how these elements can be connected together (e.g., as in Example 3 and Figure 3.3). Definition 3 below recalls how type graphs were defined originally for ADR [18].

Definition 3 (ADR graphs). Let $\Gamma$ be a type graph equipped with a map $\theta : E_\Gamma \to \{0, 1\}$; we call $e \in E_G$ terminal if $\theta(\sigma(e)) = 0$ and non-terminal if $\theta(\sigma(e)) = 1$. An ADR graph $G$ is a (hyper)graph typed over $\Gamma$ through $\tau_G$ if $\tau_G$ is a morphism from $G$ to $\Gamma$.

In words, ADR’s [18] edges are defined in the type graph as terminal or non-terminal. A terminal edge represents an atomic component that cannot be further refined; non-terminal edges instead can be thought of as components amenable to be refined. The ADR framework resembles string grammars where terminal symbols correspond to terminal edges and non-terminal symbols to non-terminal edges.
CHAPTER 3. BACKGROUND INFORMATION

Figure 3.3: ADR Type Graph

Example 3. Take the type graph $\Gamma = (V, E, t)$ (Graph $\Gamma$ is visually presented in Figure 3.3) where

$V = \{\bullet\} \subseteq \mathcal{N}$, $E = \{\text{Chain}, \text{ShipON}, \text{ShipOFF}, \text{FleetON}, \text{AP}, \text{APS}, \text{CS}\} \subseteq \mathcal{E}$, and

$t : \text{CS} \mapsto (\bullet)$, $t : \text{AP} \mapsto (\bullet, \bullet, \bullet)$, and $t : e \mapsto (\bullet, \bullet)$ for each $e \in E \setminus \{\text{CS, AP}\}$

Example 4. The graph $G = (\{u_1, u_2, u_3, u_4\}, \{s_1, ms_1, ms_2\}, t')$ (visually presented in Example 5) where $t'$ is defined as $t' : s_1 \mapsto (u_2, u_1)$, $t' : ms_1 \mapsto (u_3, u_2)$, and $t' : ms_2 \mapsto (u_4, u_2)$ can be typed on $\Gamma$ by $\tau_G$ mapping all the nodes to $\bullet$, $ms_1$ and $ms_2$ to $\text{FleetON}$, and $s_1$ to $\text{ShipON}$.

Type and typed graphs have a convenient visual notation. Nodes are circles and edges are drawn as (labelled) boxes; tentacles are depicted as lines connecting boxes to circles; conventionally, directed tentacles indicate the first node attached to the edge and the others are taken clockwise. The edges in a graph are either single- and double-lined boxes; the former represent terminal edges while the latter represent non-terminal
ones. The visual notation for typed graphs include the graph and its typing morphism. Nodes are paired with their types while an edge label $e : e'$ represents the fact that the typing morphism maps the edge $e$ of the graph to the edge $e'$ of the type graph.

In Example 5 we show the visual representation of graph $G$ textually presented in Example 4.

**Example 5.** Graph $G$ is visually presented below

![Graph G](image)

**Definition 4** (Typed Graph morphisms). A morphism between $\Gamma$-typed graphs $f : G_1 \rightarrow G_2$ is a typed graph morphism if it preserves the typing, i.e. such that $\tau_{G_1} = \tau_{G_2} \circ f$.

### 3.2.3 Design Productions

A key aspect of ADR is to envisage systems as ensembles of *designs*, that is components with *interfaces*. Designs are supposed to be generated by means of productions and can be subject to run-time reconfigurations modelled as *reconfiguration rules* (see Section 3.2.4).

Design Productions can be considered as a set of rules indicating how components can be legally interconnected. Any architecture is constructed following a style that
corresponds to a term built out of design productions: the term represents the way in which the architecture was constructed (by observing its LHS) and what the term returns (RHS) is the architecture itself. Intuitively, they can be thought of as rewriting rules that, when applied to a graph $G$, replace a non-terminal (hyper)edge of $G$ matching $L$ with a fresh copy of $R$ (we remark that our morphisms are type-preserving).

**Definition 5** (Productions). A (design) production $p$ is a tuple $\langle L, R, i : V_L \to V_R \rangle$ where $L$ is a graph consisting only of a non-terminal edge attached to distinct nodes; $R$ is an ADR graph (with both terminal and non-terminal edges); the nodes in $\text{Im}(i)$ (the image of $i$) are called interface nodes.

In other words a design production is an assembly of basic components that have a typed interface. The idea is that the type of the interface edge represents the abstract component class, while its tentacles represent the exposed nodes.

The next example defines a production that will be used later (e.g., in Example 7).

**Example 6.** Take the following graphs:

\[
G_L = \langle \{a, b\}, \{ch\}, ch \mapsto (a, b) \rangle
\]

\[
G_R = \langle \{u_1, u_2, u\}, \{sh, fl\}, t_R : \begin{cases} 
sh \mapsto (u, u_2) \\
fl \mapsto (u_1, u) 
\end{cases} \rangle
\]

with $ch$ of type type *Chain*, $sh$ of type *ShipON*, and $fl$ of type *FleetON*. (Note that $G_L$ is a single-edge graph.)
The production $\text{shipChain} = \langle G_L, G_R, i_{\text{shipChain}} \rangle$ has $G_L$ and $G_R$ as left-hand side (LHS) and right-hand side respectively; the interface of $\text{shipChain}$ is given by the map defined as follows:

$$i_{\text{shipChain}}: \begin{cases} 
    a \mapsto u_1 \\
    b \mapsto u_2 
\end{cases}$$

namely, $a$ (resp. $b$) corresponds to the first (resp. second) node of $ch$.

Like ADR graphs, productions have an appealing visual representation that we illustrate in the next example that depicts the production of Example 6.

**Example 7.** The graphical representation below represents a design production.

The dotted square and the dotted lines represent the LHS and the map $i_{\text{shipChain}}$: the name and type of the edge of the LHS is in the top-left corner of the dotted box and the name of the production is given on the top of the dotted square. The RHS of $\text{shipChain}$ is depicted inside the dotted box.

Design Productions implicitly equip designs with a hierarchical structure that can be formalised as the “derivation tree” determining the design. In fact, a set of ADR productions induces a multi-sorted algebraic signature $\Sigma$ where the sorts are the type edges in the type graph and the operations are the productions themselves, once a total
order on the edges in the RHS of the production is fixed. Hereafter, we fix such an order\(^1\) and, given the RHS \(R\) of a production, we write \(R[j]\) for the \(j\)-th edge in \(R\). With this construction, an ADR production becomes an operation with type

\[
E_1 \times \ldots \times E_n \rightarrow L \tag{3.1}
\]

where \(E_k\) is the type of the \(k\)-th edge in the RHS of the production (according to the chosen order on edges in the RHS) and \(L\) is the type of the edge in the LHS. In other words, an ADR production like in (3.1) can be envisaged as an operation in an *algebra of designs* that builds a design \(G\) of type \(L\) out of designs \(G_k\) of type \(E_k\) (for \(1 \leq k \leq n\)). This corresponds to a “bottom-up” development (whereby designs are assembled out of other components) and, as observed in [18], it parallels the “top-down” generation of designs (similar to context-free grammars) reviewed in Definition 5. Moreover, one could consider the terms (with sorted variables to model partial designs) built on \(\Sigma\) and adopt the obvious operational interpretation: \(G\) is obtained by replacing the \(j\)-th edge in the RHS with \(G_j\) (and connecting the interface nodes appropriately). The elements of such a term algebra correspond to the proof that a given design can be assigned some type.

**Example 8.** The production *shipChain* in Example 7 yields the operation

\[
\text{shipChain} : \text{ShipON} \times \text{FleetON} \rightarrow \text{Chain}
\]

assuming that in the chosen order, *ShipON* is smaller than *FleetON*.

\(^1\)The chosen order is completely arbitrary and does not affect the construction described above.
The next example illustrates how productions are applied to graphs; the technical details will be given in Definition 15, page 65 for asserted productions, which encompass ADR productions.

**Example 9.** Consider the production \textit{shipChain} of Example 7. Below, the unique edge of type \textit{Chain} is replaced by an instance of the RHS of \textit{shipChain}.

\begin{center}
\includegraphics[width=0.5\textwidth]{example9.png}
\end{center}

\textit{Note that the rest of the graph (consisting only of the edge a) including the interface nodes is left unchanged while a fresh node u5 is created.}

### 3.2.4 Run-time reconfigurations

ADR exploits the algebraic view of productions (cf. Section 3.2.3) to model complex architectural reconfigurations that cannot be captured by productions. In fact, the design can evolve when components have to be removed, added, or assembled in a different way. For instance, in our running example, the system must deal with connected or disconnected ships or collections of ships that can also migrate from one access point...
Architectural reconfigurations are naturally modelled as transformation of elements in the Σ-term algebra (cf. Section 3.2.3) with variables Σ, X (where X is the set of variables). Formally, this is achieved by defining a term rewriting system on TermΣ,X; namely, a reconfiguration rule takes the form

$$t \rightarrow t'$$  \hspace{1cm} (3.2)

where $t, t' \in \text{Term}_{\Sigma,X}$ are linear terms (that is each variable occurs at most once in $t$ and similarly for $t'$) and the variables occurring in $t'$ also occur in $t$.

**Example 10.** Given the design production `manyShips`

$$\text{manyShips}: \text{FleetON} \times \text{FleetON} \rightarrow \text{FleetON}$$

and combining with the operation in Example 8, associated to the production in Example 7, one could build the term

$$\text{shipChain}(x_1, \text{manyShips}(x_2, x_3))$$

of type `Chain` where, $x_1$ is of type `ShipON`, while $x_2$ and $x_3$ are of type `FleetON`. 

3.2. An overview of Architectural Design Rewriting
Below we give an example of simple reconfiguration and for readability abbreviate the productions clusterChain, cluster, shipChain and manyShips with clCh, cl, shCh and maCh respectively.

**Example 11.** Consider the following productions:

\[
\text{cluster: } AP \times \text{FleetON} \rightarrow APS
\]

\[
\text{clusterChain: } APS \times APS \rightarrow APS
\]

We can define the reconfiguration rule given below which transforms the LHS graph underneath it to the one on the RHS.

\[
\text{swapShips: } clCh(cl(x_1,x_2), cl(y_1,y_2)) \rightarrow clCh(cl(x_1,y_2), cl(y_1,x_2))
\]
In Example 11 one should observe that, unlike in the application of ADR productions, the identities of all the edges are preserved when applying the reconfiguration rule \texttt{swapShips}. From the reconfiguration and the productions provided one can identify that variables $x_1$, and $y_1$ are both for type \texttt{AP}, while variables $x_2$, and $y_2$ are both for type \texttt{FleetON}. To apply a reconfiguration one would have to initially identify all possible sub-graphs that can be typed as $x_1$, $x_2$, $y_1$, and $y_2$. This can only be done by visiting the graph and trying to identify all possible candidate sub-graphs. For instance, when looking for the sub-graph matching $x_2$ one can start from $s_1$, but since it is not of type \texttt{FleetON}, one can try with the sub-graph made of $s_1$ and $ms_1$, then the sub-graph made of $s_1$, $ms_1$, and $ms_2$ and so on. Note that even if a sub-graph of the appropriate type is found for a variable it is not necessarily the appropriate one when put together with the sub-graphs chosen for the other variables. All the sub-graphs matched to the variables should also match the architectural style of the reconfiguration. It is important to understand that this could become very inefficient when dealing with large complicated graphs (cf. Chapter 4).

Once the LHS of the reconfiguration is mapped to a sub-graph of the graph like for instance the LHS graph shown above, then we replace that sub-graph with the RHS of the reconfiguration preserving the sub-graphs matched for every variable.
We present a variant of ADR that generalises the original rewriting mechanism of ADR (cf. Section 3.2) based on terminal and non-terminal edges. In our variant, an edge can be rewritten if it is marked as replaceable in the graph. Besides some simplification and more uniformity in the technical presentation of ADR, such generalisation brings in extra flexibility. In fact, a replaceable edge can be refined by introducing new versions of a rule that differs only for the “replaceability” of some edges (cf. Section 4.1).

In Section 4.2 we introduce a monitoring mechanism that is also exploited to define an efficient parsing of ADR graphs. Our monitoring mechanism keeps track of the changes that occur to the system by the application of design productions and reconfiguration rules.

In Section 4.3 we present a new more efficient way of reconfiguring the system that allows us to keep the monitoring mechanism updated accordingly even during the execution of a system.
4.1 Typing Graph and Replaceability Mapping

As detailed in Section 3.2.2, ADR graphs are typed over a fixed type graph via typing morphisms. Definition 3 in Section 3.2.2 gives a precise definition of ADR typed graphs where edges are set to be either terminal or non-terminal.

We propose a variant of ADR that removes the notion of terminal and non-terminal edges from ADR’s type graph. Our variant eliminates such distinction; an edge can be rewritten if it is marked as “replaceable” in the graph. Therefore, we allow edges of the same type to be rewritten or not depending on how they are marked in the graph. Technically, this is achieved by introducing a replaceability mapping that states whether an edge at a specific moment in the graph is replaceable or not. Besides some simplification in the technical presentation of ADR (which is now more uniform), such generalisation brings in extra flexibility. In fact, a replaceable edge can be refined by introducing new versions of a rule that differs only for the “replaceability” of some edges. As shown in Example 12, an edge is not typed at the beginning as terminal or non-terminal but instead in a graph one could have several instances of the same type of edge marked as replaceable while several other instances marked as non-replaceable depending on the situation. We note that the original ADR rewriting mechanism can still be obtained: if one decides that an edge type is non-terminal, then all edges of that type have to be replaceable while for types of terminal edges, all edges have to be marked as non-replaceable.

The type graph definition of ADR in Definition 3, page 31 is now simplified as it no longer includes the terminal and non-terminal mapping. Definition 6 below, formally defines the replaceability mapping explained earlier. This is similar to the terminal and
non-terminal mapping with the difference that the mapping is now on the graph of the system and not its type graph.

**Definition 6 (ADR Replaceability Mapping).** Let $G$ be a graph. Then $\theta : E_G \to \{0, 1\}$ is a replaceability map; an edge $e \in E_G$ is (resp. non) replaceable wrt $\theta$ iff $\theta(e) = 1$ (resp. $\theta(e) = 0$). Abusing notation we will implicitly assume that any graph $G$ is equipped with a replaceability map, which we will denote by $\theta_G$.

For convenience, we keep the same visual notation used for ADR’s terminal and non-terminal edges; now single- and double-lined boxes represent non-replaceable and replaceable edges respectively. Below we show an example of a graph where two instances of an edge of the same type have a different replaceability mapping. This was not possible in the original specifications of ADR as all instances of a edge had to respect the typing mapping.

**Example 12.** This example shows two instances of an edge of type AP with different replaceability mapping. Edge $a_1$ is replaceable whereas edge $a_2$ is non-replaceable.
4.2 Tracking ADR Architectural Reconfigurations

This section exploits the algebraic presentation of ADR production and reconfiguration mechanisms and combines them together with a tracking mechanism that is used to recover possible run-time misbehaviour.

Definition 7 below formalises our tracking mechanism using some trees to record graphs’ evolution due to productions and reconfigurations respectively. We introduce some technical machinery first.

We consider forests of trees with vertices drawn from a set $\mathbb{N}$ (hereafter we will call the nodes of the trees vertices in order to distinguish them from the graph’s nodes); if $f : X \to Y$ is a partial map then we write $f(x) \uparrow$ when $f$ is undefined on $x$ and we let $\text{dom } f = X \setminus \{x \in X \mid f(x) \uparrow\}$. Hereafter, we fix a finite set of productions $\mathcal{P}$ to denote all the productions of the system. A tracking Environment $\mathcal{T}$ is pair of two injective finite partial maps

$$\mathcal{T}^{(1)} : \mathbb{N} \to \mathcal{E} \times \mathbb{N}^*, \quad \text{and} \quad \mathcal{T}^{(2)} : \mathbb{N} \to \mathcal{P},$$

and we use $\mathbf{0}$ to denote the empty environment (that is the environment undefined on all $n \in \mathbb{N}$).

Basically, given a forest $T$, we use an environment $\mathcal{T}$ (such that $\text{dom } \mathcal{T}$ is the set of vertices of $T$) to decorate each vertex of $T$ with two attributes:

- $\mathcal{T}^{(1)}(n)$ assigns an edge with its list of nodes to the vertices of $T$, and
- $\mathcal{T}^{(2)}(n)$ assigns a production to the vertex $n$ in $T$. 

4.2. Tracking ADR Architectural Reconfigurations 44
It is convenient to write $T(n) \overset{\text{as}}{=} e(\bar{x}) \cdot p$ when $T^{(1)}(n) = e(\bar{x})$ and $T^{(2)}(n) = p$. Also, in the following we use a notation inspired by object-oriented programming to manipulate trees; more precisely, we consider trees $T$ (and their nodes $n$) as objects and write $T.addTree(n, T_1, \ldots, T_k)$ to add the trees $T_h$, for $1 \leq h \leq k$, as sub-trees of $T$ by rooting them at the vertex $n$ in $T$; that is, the resulting tree will be $T$ where vertex $n$ is the root of new children $T_1, \ldots, T_k$. Also, we let $\deg n$ denote the degree of a vertex $n$, $n[j]$ its $j$-th child, and (abusing notation) we allow ourselves to identify trees consisting only of a root with the root vertex.

**Definition 7** (Tracking productions). Let $G_0, \ldots, G_m$ be a sequence of graphs s.t. for each $0 \leq j < m$, $G_{j+1}$ is obtained from $G_j$ by applying a production $p_j \in \mathcal{P}$ with morphisms $\sigma_j' : L_j \to G_j$ and $\sigma_j : R_j \to G_{j+1}$ where $L_j$ and $R_j$ are the LHS and RHS of $p_j$, respectively.

We associate to each $G_j$ a tracking forest $T_j$ and a tracking environment $\mathcal{T}_j$ as follows:

- let $r$ be the number of edges in $G_0$, forest $T_0 = n_1, \ldots, n_r$ consists of $r$ single-vertex trees with roots $n_1, \ldots, n_r$ taken pairwise distinct. Environment $\mathcal{T}_0$ is defined as the map that takes the $m$-th vertex in the forest $T_0$ to the $m$-th edge of $G_0$; formally,

$$\mathcal{T}_0[n_m \mapsto e_m(\bar{x}_m) \cdot ↑] \quad \text{for} \quad 1 \leq m \leq r$$

where $e_m = G_0[m]$ and $\bar{x}_m$ are the nodes in $G_0$ that $e_m$ is attached to;

- Let $k_j$ be the number of edges in the RHS of $p_j$ (that is, $k_j$ is the cardinality of
$E_{R_j}$, and let $n$ be the inverse image of $\sigma'_j(e_j(\bar{x}_j))$ through $\mathcal{T}_j^{(1)}$ s.t.

$$\mathcal{T}_j^{(1)}(n) = \sigma'_j(e_j(\bar{x}_j))$$

and, for $1 \leq l \leq k_j$, let $T'_i$ be a tree made of just a fresh vertex, then

$$T_{j+1} = T_j \text{.addTree}(n, T'_1, \ldots, T'_{k_j}).$$

Environment $\mathcal{T}_{j+1}$ is obtained by updating $\mathcal{T}_j$ in the following way:

\[
\mathcal{T}_{j+1} = \begin{cases} 
\mathcal{T}_j[n \mapsto \sigma'_j(e_j(\bar{x}_j)) \cdot p_j, \ T'_l \mapsto \sigma_j(R_j[l]) \cdot \uparrow \ | \ l = 1, \ldots, k] & \text{if } E_{R_j} \neq \emptyset \\
\mathcal{T}_j[n \mapsto \sigma'_j(e_j(\bar{x}_j)) \cdot p_j, \ T'_1 \mapsto \uparrow \cdot \uparrow] & \text{if } E_{R_j} = \emptyset 
\end{cases}
\]

Despite some technical intricacy, Definition 7 is conceptually simple. Basically, we add to $T_j$ as many fresh vertices as the edges in the RHS of the production $p_j$; such vertices become the children of the vertex $n$ in $T_j$ associated with the LHS $\sigma'_j(e_j)$. Accordingly, the environment $\mathcal{T}_{j+1}$ updates $\mathcal{T}_j$ recording edges and productions associated to $n$ and the fresh roots of $T'_i$. Observe that each forest $T_j$ has $r$ trees, with $r$ the number of edges of $G_0$. Indeed, the evolution of $G_0$ involves only the replacement of such edges (and those produced by such replacements). Therefore, we can record the application of $p_j$ to $G_j$ in a node of one of the trees representing the evolution of one of the initial edges of $G_0$.

Example 13. Consider the productions shipChain and manyShips from Example 10, page 38.
The left most graph, say $G_0$, represents the initial graph and $T_0$ its corresponding tree mapped to an initial environment $\mathcal{T}_0$ (cf. Definition 7) shown below.

\[
\begin{align*}
\mathcal{T}_0 : & \begin{cases} 
x \mapsto \left[ a(u_1, u_2, u_3), \uparrow \right] 
y \mapsto \left[ c(u_4, u_2), \uparrow \right] 
\end{cases} 
\end{align*}
\]

By applying \texttt{shipChain} to $c$ we obtain the next graph, say $G_1$. Since the RHS of \texttt{shipChain} generates two edges we to add two new vertices to $T_0$ as children of the vertex corresponding to $c$ to obtain $T_1$. Finally, we update the map of $y$ in $\mathcal{T}_0$ to add the production applied to $c$ and add two new mappings for the fresh vertices of $T_1$ to get $\mathcal{T}_1$. 

4.2. Tracking ADR Architectural Reconfigurations
shown below.

$$
T_1: \\
x \mapsto [a(u_1, u_2, u_3), \uparrow] \\
y \mapsto [c(u_4, u_2), shipChain] \\
y_1 \mapsto [s(u_5, u_2), \uparrow] \\
y_2 \mapsto [ms(u_5, u_4), \uparrow]
$$

We now repeat this procedure for the application of `manyShips` as well and update $T_1$ in the same fashion as before. The updated environment corresponding to $T_2$ is

$$
T_2: \\
x \mapsto [a(u_1, u_2, u_3), \uparrow] \\
y \mapsto [c(u_4, u_2), shipChain] \\
y_1 \mapsto [s(u_5, u_2), \uparrow] \\
y_2 \mapsto [ms(u_5, u_4), manyShips] \\
y_3 \mapsto [ms_1(u_5, u_4), \uparrow] \\
y_4 \mapsto [ms_2(u_5, u_6), \uparrow]
$$

Hereafter, we will show the environment $T$, within the tree $T$ as shown in the example below.

**Example 14.** This example shows a simplified way of representing the tree $T$ together with the environment $T$. We borrow the tree $T_2$ and the environment $T_2$ from Example 13 to show this visual simplification and we will use this format in the rest of the thesis.
4.3 A New Rewriting Mechanism

Following the recording mechanism in Definition 7 we define a new rewriting mechanism for ADR reconfigurations. Our approach hinges on tracking trees similar to those in Definition 7. The new rewriting approach will also allow us to keep track of changes due to reconfigurations.

Given a term \( t \in \text{Term}_{\Sigma, X} \) (cf. Section 3.2.4, page 37) and a tracking tree \( T \), the bow tie relation \( t \bowtie T \) holds iff \( t \) and \( T \) are isomorphic “up to the leaves” of \( t \); more precisely, the tree obtained by considering just the internal nodes of \( t \) is isomorphic to \( T \). This is formalised in Definition 8.

**Definition 8 (Bow tie relation).** The relation \( \bowtie \) in defined as

\[
t \bowtie T \iff t \in X \text{ or } T^2(\diamond) = p \land t = p(t_1, \ldots, t_k) \land \bigwedge_{j=1}^{k} t_j \bowtie[\diamond_j]
\]

where \( \diamond \) is the root of \( T \).

Given a reconfiguration rule \( \rho : t \rightarrow t' \), relation \( \bowtie \) given in Definition 8 allows us to identify which parts of a graph match the LHS of \( \rho \) exploiting the correspondence between tracking trees and graphs. A sub-tree \( T' \) of \( T \) matches \( t \) iff \( t \bowtie T' \). Assuming
CHAPTER 4. A VARIANT OF ADR

Figure 4.1: Term to graph morphisms and maps (cf. Definition 10)

t ⊴ T' holds then given a variable x occurring in t, Definition 9 defines the sub-tree of T' (say T_x) corresponding to x. This is obtained by applying t △◁ T' and is formalised below.

**Definition 9** (Tree of a variable). Let x be a variable occurring in t, ◦ be the root of T, 1 ≤ j ≤ k, and x ∈ t_j denote that t_j contains variable x then

\[
t △◁ T = \begin{cases} 
T & \text{if } t = x \\
t_j △◁ ◦[j] & \text{if } T(2)(◦) = p \land t = p(t_1, \ldots, t_k) \land x ∈ t_j 
\end{cases}
\]

returns the subtree of T corresponding to x if there is a path from the root of t to x.

Definition 10 builds a graph γ(t) out of a term t ∈ Term_{Σ, X}. Definition 10 inspects t inductively and generates a graph corresponding to the productions associated to t. In the places of the variables of t, γ generates an edge of the appropriate type. More precisely γ(t) returns a triplet of fresh edges, nodes, and a mapping-function that relates variables of t the fresh edges generated. For clarity, in Figure 4.1 we provide a visual view of the morphism and mappings of Definition 10.

4.3. A New Rewriting Mechanism
**Definition 10 (From Term to Graph).** Let \( p = \langle L, R, i : V_L \to V_R \rangle \in \mathcal{P} \) and \( t \in \text{Term}_{\Sigma, \mathcal{X}} \).

\[
\gamma(t) = \begin{cases} 
(e, [n_1, \ldots, n_h], \eta : t \mapsto e) & \text{if } t \in \mathcal{X}, \\
((G_1 \cup \cdots \cup G_r)\sigma, \delta, \eta_1 ; \sigma \cup \cdots \cup \eta_r ; \sigma) & \text{if } t = p(t_1, \ldots, t_r), \\
\gamma(t_j) = (G_j, \delta_j, \eta_j) & \text{for } 1 \leq j \leq r
\end{cases}
\]

where in the first clause,

- \( e \) is a fresh edge of the type corresponding to \( t \),
- \([n_1, \ldots, n_h]\) are its \( e \)'s fresh pairwise distinct nodes and
- \( \eta \) is the mapping of the variable \( t \) to the fresh edge \( e \)

and in the second clause,

- \( \delta_j = [\bar{n}_{1j}, \ldots, \bar{n}_{lj}] \) for each \( 1 \leq j \leq r \) and
- \( \delta = \iota_L^{-1}(\{\bar{n}_1, \ldots, \bar{n}_k\}) \) where \([\bar{n}_1, \ldots, \bar{n}_k]\) are the nodes of \( L' \) and
- \( \sigma : \bar{n}_m \mapsto \iota_R(\bar{n}_{j,m}) \) where \([\bar{n}_{j,1}, \ldots, \bar{n}_{j,l_j}]\) are the nodes of the \( j \)-th edge of \( R \).

Definition 10 builds a graph \( \gamma(t) \) out of a term \( t \). Intuitively, \( \gamma \) inspects \( t \) “bottom-up” and it associates disjoint designs (graphs \( G_j \) with interfaces \( \delta_j \)) to each sub-term.
of \( t \) (note that fresh edges attached to fresh nodes are associated to each variable of \( t \)); then \( \gamma \) composes the disjoint designs according to the production \( p \) which is rendered by replacing the nodes through the substitution \( \sigma \) in Definition 10.

Definition 11 below establishes how to apply a reconfiguration rule to a graph.

**Definition 11** (Applying reconfiguration rules). Fix a reconfiguration rule \( \rho : t \to t' \) with \( X \) being the set of variables of \( t \), a graph \( G \), and a tracking tree \( T \) of \( G \) with the corresponding environment \( \mathcal{T} \); let \( T' \) be a sub-tree of \( T \) such that \( t \bowtie T' \). For \( x \in X \), let \( T'_x = t \bowtie_x T' \) be the sub-tree of \( T' \) corresponding to \( x \). An application of \( \rho \) to \( G \) wrt \( T \) is a graph \( G' = G[L \mapsto t] \) where

\[
\begin{align*}
   & \bullet \ G_L = \bigcup_{l=1}^m \mathcal{T}(n_l) \text{ where } n_1, \ldots, n_m \text{ are the leaves of } T', \text{ and} \\
   & \bullet \ G_x = \gamma(t')[\eta(x) \mapsto G_x \mid x \in X] \text{ where } G_x \text{ is the sub-graph of } G \text{ corresponding to } x \text{ and it is defined as } G_x = \bigcup_{l=1}^{h_j} \mathcal{T}(n_l) \text{ with } n_{j,1}, \ldots, n_{j,h_j} \text{ being the leaves of } T'_x.
\end{align*}
\]

Finally, given the sub-trees \( T'_x \) computed above, we replace the tree \( T' \) in \( T \) with a fresh sub-tree \( T'' \) corresponding to \( t' \) where we replace the vertices corresponding to the variables of \( t' \) with the appropriate sub-trees \( T'_x \). We then update the environment \( \mathcal{T} \) so that it maps all the nodes of \( T'' \), up to the sub-trees \( T'_x \) to the productions associated to them through \( t' \).

We observe that, using Definition 10, Definition 11 simply replaces the graph corresponding to \( \gamma(t) \) with the graph corresponding to \( \gamma(t') \) where the edges corresponding to the variables of \( t' \) are replaced by the corresponding sub-graphs of \( G \) identified through the proper morphisms.
4.3.1 Applying the new rewriting mechanism

In this section we show how the technicalities presented in Section 4.3 can be applied using our running example.

Consider graph $G$ given in Figure 4.2, the corresponding tracking tree $T$ with its associated environment $\mathcal{T}$ given in Figure 4.3, and the reconfiguration $\text{swapShips}$ given below (also presented in Section 3.2.4). For readability in this section we use $t$ and $t'$ to refer to the LHS and RHS of $\text{swapShips}$ respectively and also abbreviate the productions $\text{clusterChain}$, $\text{cluster}$, $\text{shipChain}$ and $\text{manyShips}$ with $\text{clCh}$, $\text{cl}$, $\text{shCh}$ and $\text{maCh}$ respectively.

The trees below show the reconfiguration term in a tree like format offering a clearer picture of the morphism between the LHS of $\text{swapShips}$ and the highlighted tree in Figure 4.3.

To apply $\text{swapShips}$ to $G$ we first have to identify a subtree of $T$ that matches $t$. We
\[
\text{swapShips} : \text{clCh}(\text{cl}(x_1, x_2), \text{cl}(y_1, y_2)) \rightarrow \text{clCh}(\text{cl}(x_1, y_2), \text{cl}(y_1, x_2))
\]

use the bow tie relation \(\bowtie\) and match \(t\) to a subtree \(T'\) such that \(t \bowtie T'\) holds. \(T'\) is the highlighted subtree of \(T\) in Figure 4.3. Using \(T'\) we find its corresponding sub-graph \(G_L\) in \(G\) by taking the union of all the edges mapped through \(T\) to the leaves of \(T'\) to
obtain:

The next step requires the use of \( T' \) to obtain the sub-tree of \( T' \) corresponding to each variable of \( t \) to get:

\[
\begin{align*}
t \sqsubseteq_{x_1} T' &= [a_1(u_1, u_2, u_3), \uparrow] \\
t \sqsubseteq_{y_1} T' &= [a_2(u_7, u_8, u_1), \uparrow] \\
t \sqsubseteq_{x_2} T' &= [c_1(u_4, u_2), \text{shipChain}] \\
&\quad [s_1(u_5, u_2), \uparrow] [ms(u_4, u_5), \text{manyShips}] \\
&\quad [ms_1(u_4, u_5), \uparrow] [ms_2(u_6, u_5), \uparrow] \\
t \sqsubseteq_{y_2} T' &= [c_2(u_9, u_8), \text{shipChain}] \\
&\quad [s_2(u_{10}, u_8), \uparrow] [ms_3(u_9, u_{10}), \uparrow]
\end{align*}
\]

We now take the union of all the edges mapped through \( T \) to the leaves of the sub-trees above \( (t \sqsubseteq_{x_1} T', t \sqsubseteq_{y_1} T', t \sqsubseteq_{x_2} T', t \sqsubseteq_{y_2} T') \), to return sub-graphs of \( G_L \) corresponding to each one. Observe that variables like \( x_2 \), and \( y_2 \), are not mapped.
to a *single* vertex of $T'$ but to *multiple* vertices.

So far we identified the sub-graph $G_L$ of $G$ matching $t$ and the sub-graphs of $G_L$ corresponding to the variables of the reconfiguration. The next step show how this information is used to apply the reconfiguration.

1. Using $\gamma$ we construct the graph $\gamma(t')$ corresponding to $t'$

\[
\gamma(t') : \quad \gamma(x_1) = e_1 : \text{AP}, \quad \gamma(y_2) = e_3 : \text{shCh}
\]

$\gamma(t')$ represents the graph corresponding to $t'$ where in the place of each variable it contains a dummy edge of the appropriate type.

The following steps show how Definition 11 is applied to return $\gamma(t')$.

- $\gamma(cl(x_1, y_2)) = ((e_1 : \text{AP} \cup e_3 : \text{shCh}) \sigma, [\bar{u}_1, \bar{u}_3], \eta_1; \sigma \cup \eta_3; \sigma)$
  - $\gamma(x_1) = e_1 : \text{AP}, [v_1, v_2, v_3], \eta_1 : x_1 \mapsto e_1$
  - $\gamma(y_2) = e_3 : \text{shCh}, [v_4, v_2], \eta_3 : x_1 \mapsto e_3$

- $\gamma(cl(x_2, y_1)) = ((e_2 : \text{AP} \cup e_4 : \text{shCh}) \sigma, [\bar{u}_2, \bar{u}_4], \eta_2; \sigma \cup \eta_4; \sigma)$
  - $\gamma(x_2) = e_2 : \text{AP}, [v_7, v_8, v_1], \eta_2 : x_2 \mapsto e_2$
  - $\gamma(y_1) = e_4 : \text{shCh}, [v_9, v_8], \eta_4 : y_1 \mapsto e_4$

2. Using the sub-graphs of $G_L$ corresponding to the variables we obtained earlier, we replace all the dummy edges in $\gamma(t')$ with the appropriate sub-graphs. This

4.3. A New Rewriting Mechanism
yield the graph $G'$. 

3. The final step requires that

(a) we replace $G_L$ in $G$ with $G'_L$;

(b) replace $T'$ with the tree $t'$ (shown earlier) where in the place of the variables we inject the corresponding sub-trees associated to each one ($t \sqsubseteq x_1 T'$, $t \sqsubseteq y_1 T'$, $t \sqsubseteq x_2 T'$, $t \sqsubseteq y_2 T'$);

(c) update $T$ by assigning production c1Ch to the vertex corresponding to the root of $t'$.

(d) assign the appropriate production (like in the previous case) to all other vertices of $t'$ excluding the leaves which are inherited from $T'$.
This chapter sets the pillars for applying DbC to ADR and introduces a new version of ADR called AcDR. Initially, in Section 5.1, we define a simple propositional logic that quantifies over types of edges and predicates on (in)equalities of nodes. Then, in Section 5.2, we incorporate the design-by-contact approach (logic) on design productions to introduce the notion of asserted design productions. By equipping design productions with pre- and post-conditions, the former constrains the applicability of the rules while the later specifies properties of the resulting graphs. Therefore, a design production can only be applied on a system if that system satisfies the assertion(pre-condition) associated with that production. Section 5.3.1 gives an introduction and a road map of the technicalities and definitions introduced in Section 5.3 where we introduce and define the weakest pre-condition algorithm. By using the algorithms one now can guarantee properties after the application of a production on the resulting graph. In Section 5.4 we show some interesting theorems regarding the weakest pre-condition algorithm and
provide their proofs. Finally Section 5.5 illustrates our approach on a simplified version of JRMI [44].

5.1 A logic for ADR

We use a simple logic tailored on ADR. Basically, our logic is a propositional logic to predicate on (in)equalitys of nodes. In the following we let $D, D', \ldots$ range over edges of $\Gamma$.

Definition 12 (ADR logic). Let $V$ be a countably infinite set of variables for nodes (ranged over by $x, y, z, \ldots$). The set $L$ of (graph) formulae for ADR is given by the following grammar:

$$
\psi, \varphi ::= x = y \mid \top \mid \neg \varphi \mid \psi \land \varphi \mid \forall D(\bar{x}).\varphi
$$

In formulae of the form $\forall D(\bar{x}).\varphi$, the occurrences of $y \in \bar{x}$ in $\varphi$ are bound, $\bar{x}$ has the length of the arity of $D$ and $\bar{x}$ are pairwise distinct.

Logic $L$ is parametrised with respect to the type graph $\Gamma$ used in quantification. Variables not in the scope of a quantifier are free and the set $fv(\varphi)$ of free variables of $\varphi \in L$ is defined accordingly; also, we abbreviate $x_1 = x_2 \land \ldots \land x_{n-1} = x_n$ with $x_1 = x_2 = \ldots = x_{n-1} = x_n$ and we define $\bot$ as $\neg \top$, $x \neq y$ as $\neg(x = y)$, $\varphi \lor \psi$ as $\neg(\neg \varphi \land \neg \psi)$, $\varphi \rightarrow \psi$ as $\neg \varphi \lor \psi$, and $\exists D(\bar{x}).\varphi$ as $\neg \forall D(\bar{x}).\neg \varphi$. 

5.1. A logic for ADR
Example 15. The formula

$$\phi_{ex} = \forall D(x,y).\exists D'(z).x = z$$

describes graphs such that each edge of type $D$ is connected to one of type $D'$ on the first tentacle.

The models of $\mathcal{L}$ are ADR graphs together with an interpretation of the free variables of formulae. Definition 13 below gives the satisfaction relation of $\mathcal{L}$.

Definition 13 (Satisfaction relation). An ADR graph $G$ satisfies $\varphi \in \mathcal{L}$ under the assignment $h : V \to V_G$ (in symbols $G \models_h \varphi$) iff

$$\varphi \equiv \top, \quad \text{or}$$
$$\varphi \equiv x = y \quad \text{and} \quad h(x) = h(y), \quad \text{or}$$
$$\varphi \equiv \neg \varphi' \quad \text{and} \quad G \not\models_h \varphi', \quad \text{or}$$
$$\varphi \equiv \varphi_1 \land \varphi_2 \quad \text{and} \quad G \models_h \varphi_1 \text{ and } G \models_h \varphi_2, \quad \text{or}$$
$$\varphi \equiv \forall D(\tilde{x}).\varphi \quad \text{and} \quad G \models_{h[\tilde{u} \mapsto \tilde{u}]} \varphi \text{ for any } e(\tilde{u}) \in G \text{ s.t. } \tau_G(e) = D \quad \text{or}$$

Note that in the last clause of Definition 13, each bound variable in $\tilde{x}$ is replaced with a node; also, observe that the typing morphism $\tau$ is used to check $\varphi$ on any edge of type $D$ of $G$.

Fact. For each $h, h' : V \to V_G$, if $h|_{\text{fv}(\varphi)} = h'|_{\text{fv}(\varphi)}$ then $G \models_h \varphi$ iff $G \models_{h'} \varphi$.

By the above property, in $G \models_h \varphi$ we can restrict to finite mappings $h$ that only assign variables in $\text{fv}(\varphi)$. Bound variables are assigned via the formula $\varphi$ and therefore, hereafter, we write $G \models \varphi$ when $\text{fv}(\varphi) = \emptyset$. 

5.1. A logic for ADR
In Example 16 we show a graph that satisfies the formula \( \phi_{ex} \) taken from Example 15, and one that does not satisfy the invariant.

**Example 16.** Recall the invariant

\[
\phi_{ex} = \forall D(x, y). \exists D'(z). x = z
\]

and consider the graphs

\[
G_{\text{valid}} = \bullet \xrightarrow{d_1 : D} u_1 \xleftarrow{d' : D'} \quad G_{\text{invalid}} = \bullet \xrightarrow{d_1 : D} u_1 \xleftarrow{d' : D'}
\]

then \( G_{\text{valid}} \) satisfies \( \phi_{ex} \) whereas \( G_{\text{invalid}} \) does not. Looking at \( \phi_{ex} \) we can observe that \( h \) is empty initially since \( \phi_{ex} \) is a closed formula.

We show the latter case. According to Definition 13, we have to find an edge of type \( D \) in \( G_{\text{invalid}} \) such that the equality in \( \phi_{ex} \) does not hold for any edge of type \( D' \) in \( G_{\text{invalid}} \) (under the corresponding assignment). In fact, choosing \( d_2 \), we get the assignment \( h : x \mapsto u_3 \) and \( d' \) (the only edge of type \( D' \) in \( G_{\text{invalid}} \)) would extend \( h \) with the assignment \( z \mapsto u_1 \) (and therefore invalidating the equality in \( \phi_{ex} \)).

Despite its simple syntax, our logic is quite expressive because it permits one to quantify over types of edges. Also, when considering design productions it allows even more expressive graph constraints, since for instance a replaceable edge of type \( D \) subsumes the set of graphs of type \( D \) (cf. Example 19).

In Example 17 below we use the running scenario provided in Section 3.1 for a more intuitive understanding. Recall that, edges of type \text{ShipON} and \text{ShipOFF} represent ships
connected or disconnected to the network respectively. Intuitively a ship is considered connected if it is either connected directly through an access point or if it is connected to another connected ship.

**Example 17.** The formula

\[
\phi_{ex1} = \forall ShipON(x, y). ((\exists AP(a, b, c). x = b) \lor (\exists ShipON(a, b). x = b \land y \neq a))
\]

describes a graph that for every edge of type \(ShipON\) there is either an edge of type \(AP\) or an edge of type \(ShipON\) connected to it on the first tentacle. Consider the graphs

\[G_1, G_2, G_3\]

then \(G_1\) and \(G_3\) satisfy \(\phi_{ex2}\) whereas \(G_2\) does not because \(s\) is not connected to any edge of type \(AP\) or \(ShipON\).

More interesting formulae are given in the next two examples.

**Example 18.** The formula

\[
\text{noEdge}(D) \overset{\text{def}}{=} \forall D(\bar{x}). \perp
\]

(5.1)
CHAPTER 5. INTRODUCING ACDR

characterises the graphs that do not contain edges of a given type.

Formulae of the form (5.1) will be used in Definition 20. Hereafter, we write

\[ \text{noEdge}(D_1, \ldots, D_n) \quad \text{for} \quad \text{noEdge}(D_1) \land \ldots \land \text{noEdge}(D_n) \]

The next example shows that, despite its simplicity, our logic is quite expressive when “taken modulo productions”. The production and the formula below characterise graphs that contain paths of edges of type $C$ between any two distinct nodes connected by an edge of type $D$.

**Example 19.** By the production below, a replaceable edge of type $C$ can be replaced by a chain of two edges of type $C$. The formula $\text{path } D \ C$ requires that any two different nodes attached to an edge of type $D$ are connected by an edge of type $C$.

\[
\begin{align*}
\kern-8pt & C \quad \kern-8pt \\
\kern-8pt & \bullet \xrightarrow{u_2} \boxed{c_1 : C} \xrightarrow{u} \boxed{c_2 : C} \xrightarrow{u_1} \bullet \\
\end{align*}
\]

\[ \text{path } D \ C \quad \text{def} \quad \forall D(x, y).x \neq y \rightarrow \exists C(u, v).(x = u \land y = v) \]

Note that even though there is no edge of type $D$ in the production, $\text{path } D \ C$ quantifies over edges of type $D$ in the graph.

5.1. A logic for ADR
5.2 Design-by-Contract for ADR

Our notion of contracts hinges on asserted productions, namely ADR productions decorated with pre- and post-conditions expressed in the logic $\mathcal{L}$ given in Section 5.1. As discussed in Chapter 1, an asserted production is a production decorated with a pre-condition $\psi$ and a post-condition $\varphi$ that "stipulates the contract" or the rewriting; intuitively, we would like to have that when the production is applied to a graph satisfying $\psi$ then the resulting graph satisfies $\varphi$.

Definition 14 (Asserted productions). Let $p = \langle L, R, i \rangle$ be an ADR production as in Definition 5, $\psi, \varphi \in \mathcal{L}$ and $h, h' : V \to \mathcal{R}$ be two assignments. An expression of the form

$$\{\psi, h\} p \{\varphi, h'\}$$

where $h(\text{fv}(\psi)) \subseteq V_L$ and $h'(\text{fv}(\varphi)) \subseteq V_R$

is an asserted production.

An asserted production generalises ADR productions (cf. Definition 5) and it intuitively requires that if $p$ is applied to a graph $G$ that satisfies $\psi$ then the resulting graph is expected to satisfy $\varphi$. The maps $h$ and $h'$ in Definition 14 allow pre- and post-conditions to predicate on nodes occurring in the LHS or the RHS of $p$.

An instance $G'$ of a graph $G$ is a graph $G'$ isomorphic to $G$ that does not share nodes or edges with $G$. Hereafter, given a graph $G$, and two edges $e$, and $e'$ we will use the notation $G[e \mapsto e']$ to indicate substitution of $e$ with $e'$ in $G$. The application of an asserted production to a graph consists of replacing an homomorphic image of the edge of the LHS with a new instance of the RHS and then connecting it to the interface nodes. This is formalised in the next definition and schematically illustrated.
Definition 15 (Applying asserted productions). Let $G$ be a graph, $p = \langle L, R, i \rangle$ be an ADR production, and $\sigma$ be a morphism from $L$ (the LHS of $p$) to $G$. An asserted production $\pi = \{\psi, h\} p \{\varphi, h'\}$ is applicable to $G$ via $\sigma$ iff $G \models \sigma \circ h \psi$.

Given an instance $R'$ of $R$ through the isomorphism $\iota : R \rightarrow R'$ a graph $G[\sigma(e) \mapsto R'']$ is the application of $\pi$ to $G$ with respect to $\sigma$ iff $R'' = R'[\iota(r) \mapsto \sigma(\iota^{-1}(r)) \mid r \in \text{Im}(\iota)]$. A production $\pi$ is valid when any application of $\pi$ to a graph satisfying the pre-condition of $\pi$ yields a graph satisfying the post condition of $\pi$.

Definition 15 generalises the hyper-edge replacement mechanism of ADR; in fact, $\{\top, \emptyset\} p \{\top, \emptyset\}$ applies exactly as normal ADR productions. Examples 20 and 21 show how asserted productions are applied to graphs.

Example 20. Consider the production shipChain given in Example 7 and the formula and the asserted production

$$\psi \overset{\text{def}}{=} \forall \text{Chain}(x, y).x \neq y \quad \text{and} \quad \pi \overset{\text{def}}{=} \{\psi, \emptyset\} \text{shipChain} \{\top, \emptyset\}$$
Consider also graph $G$ below

![Graph G]

Then, $\pi$ cannot be applied to $G$ because $G \not\cong \psi$ (under the unique morphism $\sigma$ from $L$ to $G$). In fact, $x$ and $y$ are mapped to the same node $u_2$ of $G$.

**Example 21.** The rewriting below is obtained by applying $\pi$ (*shipChain*) in Example 20.

![Rewriting]

According to Definition 15, the edge $c$ on the left is replaced by an isomorphic instance of $R$ preserving the interface nodes $u_2$ and $u_4$.

A production $\pi$ is *valid* when any application of $\pi$ to a graph satisfying the precondition of $\pi$ yields a graph satisfying the post condition of $\pi$. 

---

5.2. Design-by-Contract for ADR
5.3 Extracting contracts for ADR productions

Before the technical details of Section 5.3.2 and Section 5.3.3, we prefer to give the informal and intuitive account of Section 5.3.1 (together with the definition of some basic concepts).

5.3.1 Prelude

The application of an asserted production \( \{ \psi, h \} \ p \ \{ \varphi, h' \} \) to a graph satisfying \( \psi \) does not necessarily yield a graph satisfying \( \varphi \) (this can be trivially noted by taking a production with \( \bot \) as post-condition). We give an algorithm to compute the weakest pre-condition \( \psi \) out of a post-condition \( \varphi \) and a production \( p \); as in the seminal work of Dijkstra [25], our algorithm guarantees that the application of \( p \) to a graph satisfying \( \psi \) will yield a graph satisfying \( \varphi \) (cf. Theorem 1). In other words, \( p \) with pre-condition \( \psi \) and post-condition \( \varphi \) is valid (cf. Definition 15).

The next series of examples aims to give an intuition on the various conditions a graph should satisfy after the application of a production and how our algorithm computes weakest pre-conditions. The examples discuss what happens when the following two simple productions:

\[
\begin{align*}
p_1 : & \quad A \xrightarrow{u} \{d_1 : D_1, d_2 : D_2\} \\
p_2 : & \quad A \xrightarrow{u} \{d_1 : D_1, d_2 : D_2\}
\end{align*}
\]

are applied to a graph \( G \); in the examples, \( G_1 \) denotes the graph obtained by applying \( p_1 \) to \( G \) and likewise for \( G_2 \).
Example 22. As our post-condition we take the following formula

\[ \varphi_{ex\ 22} = \forall D_1(x_1, y). \forall D_2(x_2). x_1 = x_2 \]

stating that all edges of type \( D_1 \) and \( D_2 \) should all be connected to the same node.

- The application of \( p_1 \) to \( G \) will introduce two edges of type \( D_1 \) and \( D_2 \) respectively connected to the same fresh node. In this case, the weakest pre-condition should be \( \text{noEdge}(D_1, D_2) \) (that is, \( G \) should not contain edges of type \( D_1 \) or \( D_2 \)); in fact, such an edge could not be connected to the fresh node introduced by \( p_1 \) in \( G_1 \).

- By inspecting the RHS of \( p_2 \) it is easy to note that, regardless \( G \), \( G_2 \) will have an edge of type \( D_1 \) and one of type \( D_2 \) that are not connected. Therefore, the weakest pre-condition should be \( \bot \).

The following picture visualises the case of the first bullet of Example 22 where \( G \) is the 'cloud' on the left (with \( a \) of type \( A \)) and \( G_1 \) the one on the right of the arrow (where \( a \) disappears) and, in the latter graph, the filled elements are those freshly introduced by \( p_1 \) (their type is determined by the tentacles). If edge \( d \) in \( G \) is of type \( D_1 \) then \( G_1 \not\models \varphi_{ex\ 22} \) since filled and non-filled nodes cannot be equal.
Example 23. As our post-condition we consider now the following formula

\[ \varphi_{ex\ 23} = \forall D_1(x_1, y). \exists D_2(x_2). x_1 = x_2 \]

requiring that each edge of type \( D_1 \) has an edge of type \( D_2 \) connected to it.

- The RHS of \( p_1 \) satisfies \( \varphi_{ex\ 23} \) as it introduces two edges of type \( D_1 \) and \( D_2 \) attached to each other. So \( G_2 \) would violated \( \varphi_{ex\ 23} \) only if \( G \) had an edge of type \( D_1 \) not connected to any edge of type \( D_2 \). Therefore, the weakest pre-condition returned \( \varphi_{ex\ 23} \).

- Again, noting that the RHS of \( p_2 \) violates \( \varphi_{ex\ 23} \), we have that \( G_2 \not\models \varphi_{ex\ 23} \). Therefore, the weakest pre-condition should be \( \bot \).

The picture above also visualises the case of the first bullet of Example 23 if we assume that no edge of type \( D_2 \) is attached to the node connected to the first tentacle of \( d \); in fact, the application of \( p_2 \) will not change the connections of \( d \) in \( G_2 \) where \( G \) is the 'cloud' on the left (with \( a \) of type \( A \)) and \( G_1 \) the one on the right of the arrow (where \( a \) disappears) and, in the latter graph, the filled elements are those freshly introduced by \( p_1 \) (their type is determined by the tentacles). If edge \( d \) in \( G \) is of type \( D_1 \) then \( G_1 \not\models \varphi_{ex\ 23} \) since filled and non-filled nodes cannot be equal.

Example 24. As our post-condition we finally take the following formula

\[ \varphi_{ex\ 24} = \forall D_1(x_1, y). x_1 = z \]

that states that every edge of type \( D_1 \) must have the first tentacle connected to a node.
z introduced by the RHS of the production.

- To apply \( p_1 \) the free variable \( z \) can only be mapped to the node \( u \) in the RHS of the production (by Definition 15). If \( G \) had an edge, say \( d \), of type \( D_1 \) then \( G_1 \) could not satisfy \( \varphi_{ex\ 24} \) because the first tentacle of \( d \) could not be attached to the node that the application of \( p_1 \) freshly adds in \( G_1 \). Therefore, the weakest pre-condition should be \( \text{noEdge}(D_1) \) (that is, \( G \) should not contain any edge of type \( D_1 \)).

- When applying \( p_2 \) with \( z \) mapped to the node \( u_1 \), the RHS of \( p_2 \) violates \( \varphi_{ex\ 24} \) because, regardless \( G \), the freshly \( G_2 \) will have an edge of type \( D_1 \) that is not connected to the node \( u_1 \). Therefore, the weakest pre-condition should be \( \perp \). If \( z \) is mapped on the node \( u \) of the RHS of \( p_2 \) then the previous case applies.

The picture above also visualises the case of the first bullet of Example 24 where \( G \) is the 'cloud' on the left (with \( a \) of type \( A \)) and \( G_1 \) the one on the right of the arrow (where \( a \) disappears) and, in the latter graph, the filled elements are those freshly introduced by \( p_1 \) (their type is determined by the tentacles). If edge \( d \) in \( G \) is of type \( D_1 \) then \( G_1 \not\models \varphi_{ex\ 24} \) since filled and non-filled nodes cannot be equal.

### 5.3.2 Auxiliary mappings

Before presenting our algorithm, we give some auxiliary definitions.

The algorithm requires one to inspect the post-condition \( \varphi \); to this purpose, we use an environment \( \mathcal{E} \) to record how variables of \( \varphi \) are quantified (Definition 16). Hereafter, bound variables in a formula are assumed distinct from its free variables and are bound only once.
Definition 16 (Environment). Let $\mathcal{N}$ denote the set of natural numbers. An environment $\mathcal{E}$ is a triple of finite partial maps

$$
\mathcal{E}^{(1)} : V \rightarrow \{\forall, \exists\} \times \mathcal{N}, \quad \mathcal{E}^{(2)} : V \rightarrow E_\Gamma \times \mathcal{N}, \quad \text{and} \quad \mathcal{E}^{(3)} : V \rightarrow \mathbb{N}
$$

such that the following conditions hold:

- if $\mathcal{E}^{(1)}$ is defined on $x$, so are $\mathcal{E}^{(2)}$ and $\mathcal{E}^{(3)}$

- if $\mathcal{E}^{(2)}(x) = (D, k)$ then $1 \leq k \leq d$ where $d$ is the arity of $D$

- if $x_1 \neq x_2$, $\mathcal{E}^{(1)}(x_j) = (q_j, l_j)$, and $\mathcal{E}^{(2)}(x_j) = (D_j, k_j)$ for $j = 1, 2$ then $l_1 = l_2$ implies $q_1 = q_2$ and $k_1 \neq k_2$.

Let $\mathcal{E}^{(i)}(x) \uparrow$ denote the fact that $\mathcal{E}^{(i)}$ is undefined on $x$ (for $i = 1, 2, 3$) and $\mathbf{0}$ denote the empty environment.

To avoid cumbersome parentheses, $\check{q}$ shortens $(q, l) \in \{\forall, \exists\} \times \mathcal{N}$ and similarly $\overline{k}$ abbreviates $(D, k)$, moreover we ignore the indexes $l$ and $k$ when immaterial. An environment $\mathcal{E}$ records how a variable $x \in V$ is quantified (if at all) and, when it is, $\mathcal{E}$ yields the details of the quantification and the interpretation of $x$. More precisely,

- if $\mathcal{E}^{(1)}(x) = \check{q}$ for some $(q, l) \in \{\forall, \exists\} \times \mathcal{N}$, then $\mathcal{E}^{(1)}$ specifies if $x$ is universally or existentially quantified, and that it is bound by the $l$-th quantifier,

- $\mathcal{E}^{(2)}(x) = \overline{k}$ specifies the type $D$ and the tentacle $k$ of the edge which $x$ is attached to in the quantification, and

- $\mathcal{E}^{(3)}$ assigns a node to $x$. 

5.3. Extracting contracts for ADR productions
It is convenient to write $\mathcal{E}(x) \overset{as}{=} q \, D \, G$ when $x$ is quantified by $q$ (that is $\mathcal{E}^{(1)}(x) = \overset{l}{q}$ for some $l \in \mathcal{N}$), attached to an edge of type $D$ (that is $\mathcal{E}^{(2)}(x) = \overset{k}{D}$, for some $k \in \mathcal{N}$), and mapped to a node of $G$ (that is $\mathcal{E}^{(3)}(x) \in V_G$); if $G$ consists of a node $n$, we simply write $\mathcal{E}(x) \overset{as}{=} q \, D \, n$. Also, we use "_" as a wild-card writing e.g. $\mathcal{E}(x) \overset{as}{=} q \, _{-} \, G$ when the type assigned to $x$ is not defined or it is immaterial (i.e., $\mathcal{E}(x) \overset{as}{=} q \, _{-} \, G$ abbreviates $\mathcal{E}^{(1)}(x) = q$ and $\mathcal{E}^{(3)}(x) \in V_G$).

During the analysis the algorithm adds to $\mathcal{E}$ all the possible assignments of the variables in $\varphi$, to both the nodes in $p$, and a fixed set of representative nodes not in $p$.

- The weakest pre-condition $\psi$ is the conjunction of the predicates returned by the transformers $\text{wd}_p^p$ (cf. Definition 19) and $\text{wp}_p^p$ (cf. Definition 20) on a post condition $\varphi$. The two transformers are very similar in they way they are defined but are used to handle two different types of scenarios.

  - The predicate transformer $\text{wd}$ checks if the RHS of a production satisfies the post-condition $\varphi$ regardless of the graph the production is applied to. For instance, this is useful to accommodate the cases where $\varphi$ requires the existence of edges (with some properties) generated in the RHS of the production $p$; then such post-condition $\varphi$ is guaranteed whatever graph the production is applied to.

  - The transformer $\text{wp}$ instead determines the conditions that depend on the graph the production is applied to. When the RHS of a production cannot guarantee the validity of its application in terms of $\varphi$ then $\text{wp}_p^p$ identifies the weakest pre-condition by considering the assignments in $\mathcal{E}$ on the represen-

---

5.3. Extracting contracts for ADR productions
For the equality cases ($x_1 = x_2$), Definition 19 and Definition 20 invoke the auxiliary map $eq_{p, x_1=x_2}(\mathcal{E})$ given in Definition 17 (described in the next section). For the sake of clarity, we collect the cases where the post-condition cannot be guaranteed (and for which $eq_{p, x_1=x_2}(\mathcal{E})$ returns false) in Definition 18.

Arguably, formulae of the form $x_1 = x_2$ constitute the most involved case when trans-
forming post-conditions. Therefore, Definition 17 below introduces the map \( eq_{p,x_1=x_2}(\mathcal{E}) \) used for the equality case by the predicate transformers given in Definition 19 and Definition 20. Figure 5.3.2 yields synoptic summary of Definition 17. Provided that the environment \( \mathcal{E} \) is defined on the variables \( x_1 \) and \( x_2 \), the map \( eq_{p,x_1=x_2}(\mathcal{E}) \) returns a formula as detailed below.

Hereafter, \( p = \langle L, R, i \rangle \) is a production and we write \( R^o \overset{\text{def}}{=} V_R \setminus \text{Im}(i) \) to denote the internal nodes of \( p \), and \( \bar{R} \overset{\text{def}}{=} \mathcal{N} \setminus V_R \) to denote the nodes outside \( p \).

**Definition 17** (Auxiliary map). *Given that \( \mathcal{E}^{(3)}(x_1) \) and \( \mathcal{E}^{(3)}(x_2) \) are not undefined,*
and $\psi_1, \psi_2 \in \mathcal{L}$, $eq^{\psi_1, \psi_2}_{p,x_1=x_2}(\mathcal{E})$ is defined as

$$eq^{\psi_1, \psi_2}_{p,x_1=x_2}(\mathcal{E}) = \begin{cases} 
\bot, & \text{if } \text{cond}_{\bot} \\
\top, & \text{if } \mathcal{E}^{(1)}(x_1) \uparrow, \mathcal{E}^{(1)}(x_2) \uparrow \text{ and } \mathcal{E}^{(3)}(x_1) = \mathcal{E}^{(3)}(x_2) \\
\top, & \text{if } \mathcal{E}(x_1) \models \exists \_ n, \mathcal{E}(x_2) \models \exists \_ n \text{ and } n \in R^o \text{ and } \mathcal{E}^{(3)}(x_1) = \mathcal{E}^{(3)}(x_2) \\
\top, & \text{if } \mathcal{E}(x_1) \models \forall \_ n, \mathcal{E}(x_2) \models \exists \_ n \text{ and } n \in R^o \\
\psi_1, & \text{if } \mathcal{E}^{(1)}(x_1) \uparrow \text{ and } \mathcal{E}(x_2) \models \forall \_ R^o \text{ and } \mathcal{E}^{(3)}(x_1) = \mathcal{E}^{(3)}(x_2) \\
\psi_1, & \text{if } \mathcal{E}(x_1) \models \forall \_ R^o \text{ and } \mathcal{E}(x_2) \models \forall \_ R \\
\psi_1, & \text{if } \mathcal{E}(x_1) \models \exists \_ n, \mathcal{E}(x_2) \models \forall \_ n \text{ and } n \in R^o \\
\psi_2, & \text{if } \mathcal{E}(x_1) \models \forall \_ n, \mathcal{E}(x_2) \models \forall \_ n \text{ and } n \in R^o \\
x_1 = x_2, & \text{otherwise}
\end{cases}$$

and, depending on $\mathcal{E}$, returns either $\psi_1$, $\psi_2$, $x_1 = x_2$, $\top$, or $\bot$.

The parameters $\psi_1$ and $\psi_2$ allow the map $eq^{\psi_1, \psi_2}_{p,x_1=x_2}(\mathcal{E})$ to adapt its behaviour to the different predicate transformers. In fact, we will see that $wd$ and $wp$ use $eq^{\psi_1, \psi_2}_{p,x_1=x_2}(\mathcal{E})$ in different ways providing different formulae for the parameters $\psi_1$ and $\psi_2$. We comment on the various cases of Definition 17:
• $eq_{p,x_1=x_2}(\mathcal{E})$ returns $\bot$ when $\text{cond}_\bot$; such condition is given in Definition 18 and it is explained below;

• $eq_{p,x_1=x_2}(\mathcal{E})$ returns $\top$ when in $\mathcal{E}$ both $x_1$ and $x_2$ are assigned to the same internal node of $R$ (the RHS of $p$) since the application of $p$ guarantees $x_1 = x_2$ for these cases;

• $eq_{p,x_1=x_2}(\mathcal{E})$ returns $\psi_1$ if variable $x_2$ is universally quantified and either
  
  - $x_1$ is undefined and equal to the node assigned to $x_2$ through $\mathcal{E}$;
  
  - $x_1$ is undefined while $x_2$ is assigned to an external node through $\mathcal{E}$;
  
  - $x_1$ is also universally quantified but assigned to an internal node whereas $x_2$ is assigned to an external one.

  - $x_1$ is existentially quantified and both $x_1$ and $x_2$ are assigned to an internal node.

In all such cases if the post-condition requires the equality of $x_1$ and $x_2$ then for $\mathcal{E}^{(1)}(x_2) = D_2$, $\psi_1$ would be $\text{noEdge}(D_2)$. This is because the existence of an edge of type $D_2$ in the graph ($G$) which $p$ is applied to will return a graph ($G'$) that violates $\varphi$. In the case that the post-condition requires the inequality of $x_1$ and $x_2$, $\psi_1$ will be $x_1 \neq x_2$ because the condition will have to be satisfied by $G$ in order to guarantee its validity on $G'$.

• $eq_{p,x_1=x_2}(\mathcal{E})$ returns $\psi_2$ if both variables $x_1$ and $x_2$ are universally quantified and assigned to the same internal node. Indeed, if the post-condition requires the equality of $x_1$ and $x_2$ then for $\mathcal{E}^{(1)}(x_1) = D_1$ and $\mathcal{E}^{(1)}(x_2) = D_2$, $\psi_2$ would be
noEdge\langle D_1, D_2 \rangle$. Similarly to the previous cases, this is because the existence of any edge of either type $D_1$ or $D_2$, in $G$ would violate $\varphi$ on $G'$. In case the post-condition requires the inequality of $x_1$ and $x_2$ then $\psi_2$ will be $\bot$ because the condition is already violated by the fact that in the RHS $x_1$ and $x_2$ are assigned to the same node.

- Finally, $eq_{\psi_1, \psi_2}^{p, x_1=x_2}(E)$ returns $x_1 = x_2$ for any case not specified above.

In Definition 18, $\text{cond}_\bot$ establishes the condition for the computation of the weakest pre-condition for $x_1 = x_2$ to return $\bot$.

**Definition 18** (False Conditions). $\text{cond}_\bot$ is defined as the disjunction of the following cases:

1. $E^{(1)}(x_1) \uparrow$, $E^{(1)}(x_2) \uparrow$ and $E^{(3)}(x_1) \neq E^{(3)}(x_2)$

2. $E(x_1) \equiv \exists \ - n$, $E^{(1)}(x_2) \uparrow$ and $n \notin R^o$

3. $E(x_1) \equiv \exists \ - R^o$, $E(x_2) \equiv \forall \ - n$ and $n \notin R^o$

4. $E(x_1) \equiv \ - R^o$, $E(x_2) \equiv \exists \ - R^o$ and $E^{(3)}(x_1) \neq E^{(3)}(x_2)$

5. $E(x_1) \equiv \exists \ - R^o$, and either $E^{(1)}(x_2) \uparrow$ or $E(x_2) \equiv \forall \ - R^o$ and $E^{(3)}(x_1) \neq E^{(3)}(x_2)$

6. $E(x_1) \equiv \exists \ - n$, $E(x_2) \equiv \forall \ - R^o$ and $n \notin R^o$

7. $E(x_1) \equiv \forall \ - n$, $E^{(1)}(x_2) \uparrow$ and either $n \in R^o$ and $n \neq E^{(3)}(x_2)$ or $n \in Im(i)$

8. $E(x_1) \equiv \forall \ - R^o$ and either $E(x_2) \equiv \exists \ - \bar{R}$ or $E(x_2) \equiv \exists \ - Im(i)$

5.3. Extracting contracts for ADR productions 77
9. $\mathcal{E}(x_1) \overset{as}{=} \forall D\text{ Im}(i), \mathcal{E}(x_2) \overset{as}{=} \exists \bar{R}$ and for an $e \in E_R$, $e$ has type $D$

10. $\mathcal{E}(x_1) \overset{as}{=} \forall D\text{ Im}(i), \mathcal{E}(x_2) \overset{as}{=} \exists D'\text{ Im}(i)$ and there is $e \in E_R$ such that $e$ has type $D$ and any $e' \in E_R$ has a type different from $D'$

We comment each case of the definition below.

Case 1 states that two free variables of the post condition ($\mathcal{E}^{(1)}(x_1) \uparrow$ and $\mathcal{E}^{(1)}(x_2) \uparrow$) that are not assigned to the same node cannot guarantee $x_1 = x_2$. Definitions 19 and 20 exploit the fact that free variables of the post conditions are mapped to nodes in the RHS $R$ of $p$ (cf. Definition 14). Therefore, case 2 below states that it is impossible to guarantee $x_1 = x_2$ when $x_1$ is existentially quantified and assigned to a node not in $R$ and $x_2$ is a free variable of the post condition ($\mathcal{E}^{(1)}(x_2) \uparrow$) because, by construction such nodes would be different. For the same reason, in cases 3, 4, 5, and 6, $\bot$ is returned when $x_1$ is assigned to an internal node of $R$ while $x_2$ is either outside $R^\circ$ or it is different from the node assigned to $x_1$. Case 7 stipulates that $x_1 = x_2$ cannot hold when either (as before) $x_1$ is assigned to an internal node of $R$ while $x_2$ is assigned to a different node or because one node is assigned to (the image of) an interface node and the other to an internal. Cases 8 and 9 are similar: internal and external or (the image of) interface nodes of $R$ cannot be the same. Case 10 states that if $x_1$ is universally quantified, has type $D$, and is assigned to a node in the interface of $p$, then it cannot be identified with a variable $x_2$ assigned to an interface node of $p$ attached to an edge whose type is not in $R$. 

5.3. Extracting contracts for ADR productions
5.3.3 Weakest pre-condition algorithm

This section is a continuation of Sections 5.3.1 and 5.3.2; it provides the definitions for the weakest pre-condition algorithm as well as an example showing its usage.

A formula $\varphi \in \mathcal{L}$ is in negation normal form when negations occur only in front of equalities. Conventionally, in an equality $x_1 = x_2$ occurring in $\varphi$, the quantification on $x_1$ (if any) precedes the one of $x_2$ (if any). It is trivial to see that all formulae of $\mathcal{L}$ have an equivalent negation normal form. Given an environment $\mathcal{E}$ compatible with $h$, the weakest pre-condition of $p$ with post-condition $\varphi$ under $h$, $\bar{h}$, and $\mathcal{E}$, denoted by $W_{h,\mathcal{E}}^\bar{h}(p, \varphi)$, is the formula $wd_{\mathcal{E}}^p(\varphi) \land wp_{\mathcal{E},\bar{h}}^p(\varphi)$.

As also explained in Section 5.3.1, the weakest pre-condition is the conjunction of the predicates computed by the transformers $wd_{\mathcal{E}}^p$ and $wp_{\mathcal{E},\bar{h}}^p$ on the post condition $\varphi$. We anticipate that in Definition 19 for $wd$ and in Definition 20 $wp$ below, the following assumptions are made: $Z = \{z_1, \ldots, z_m\} \subseteq V$ where $m$ is the arity of $L$, $\varphi \in \mathcal{L}$ is in negation normal form, $h : \text{fv}(\varphi) \rightarrow V_R$ is injective, and $\bar{h} : Z \rightarrow V_L$ is a bijection. We also say that an environment $\mathcal{E}$ is compatible with $h$ iff, for each $x \in X$, $\mathcal{E}^{(3)}(x) = h(x)$.

Definition 19 checks whether the RHS of a production satisfies or violates the post-condition on any graph the production is applied to, as well as checks the cases where the post-condition requires that there does not exist any edge of a certain type in the graph the production is applied to. In this case the transformer $wd$ returns either $\top$, $\bot$, or $\text{noEdge}(\bot)$ (cf. Equation 5.1, page 62) depending on the post condition.

In simple terms $wd$ returns

- $\bot$ when given a post-condition $\phi$ and a production $p$, no graph can satisfy $\phi$ after the application of $p$.

5.3. Extracting contracts for ADR productions 79
• noEdge\langle D \rangle or noEdge\langle D, D' \rangle if a graph G must not have any edge of type D/D, D' for G' \models \phi where \phi is the post-condition and G'' is the graph obtained after applying the production to G.

• ⊤ for any graph G that should satisfy the post condition \phi in order for G' \models \phi where G' is the graph obtained after applying the production to G.

Given an environment \mathcal{E}, we let max denote the maximum index of quantifiers in \mathcal{E} (we tacitly assume that max = 0 when \mathcal{E} = 0); formally,

\[
\text{max} = \max \{ l' \in \mathcal{N} \mid \exists y \in V : \mathcal{E}^{(1)}(y) = \forall^{l'} \text{ or } \mathcal{E}^{(1)}(y) = \exists^{l'} \}
\]

**Definition 19** (Transformer \text{wd}). Let eq_{p,x_1=x_2,\psi_3}(\mathcal{E}) be as the auxiliary map given in Definition 17 but for the last case where instead of returning \( x_1 = x_2 \), \( \text{eq}_{p,x_1=x_2,\psi_3}(\mathcal{E}) \) returns
ψ₃.

\[
wd_{E}^{p,\psi}(x_1 = x_2) = eq_{p,x_1=x_2}^{noEdge(D_2),noEdge(D_1),\psi(E)}
\]

where \(D_1 = E^{(2)}(x_1)\) and \(D_2 = E^{(2)}(x_2)\)

\[
wd_{E}^{p,\psi}(x_1 \neq x_2) = \begin{cases} 
-eq_{p,x_1=x_2}^{T,T,\neg\psi}(E) & \text{if } E^{(1)}(x_1) \uparrow, E(x_2) \models \forall \cdot R^o \text{ and } E^{(3)}(x_1) = E^{(3)}(x_2) \\
-eq_{p,x_1=x_2}^{\perp,T,\neg\psi}(E) & \text{otherwise}
\end{cases}
\]

\[
wd_{E}^{p,\psi}(\top) = \top
\]

\[
wd_{E}^{p,\psi}(\bot) = \begin{cases} 
\bot & \text{if } \{x \mid E(x) \models \forall \cdot R^o\} \neq \emptyset \\
\top & \text{otherwise}
\end{cases}
\]

\[
wd_{E}^{p,\psi}(\phi \land \phi') = wd_{E}^{p,\psi}(\phi) \land wd_{E}^{p,\psi}(\phi')
\]

\[
wd_{E}^{p,\psi}(\phi \lor \phi') = wd_{E}^{p,\psi}(\phi) \lor wd_{E}^{p,\psi}(\phi')
\]

Let \(D\) have arity \(d\), \(\{v_1, \ldots, v_d\} \subseteq \bar{R}\) be a fixed set of \(d\) (representative) external nodes, \(\bar{u} \in (\bar{V}_R \cup \{v_1, \ldots, v_d\})^*, \bar{x} = x_1, \ldots, x_d\), and \(l = 1 + \text{max}, \bar{u} \in (\bar{V}_R \cup \{v_1, \ldots, v_d\})^*, \bar{x} = x_1, \ldots, x_d\), and \(l = 1 + \text{max}, \)

\[
wd_{E}^{p,\psi}(\forall D(\bar{x}).\phi) = \bigwedge_{\bar{u} \text{ on } R \cdot D} \text{wd}_{E}^{p,\psi}(\phi)
\]

where \(\mathcal{E}' = E[j \mapsto (\forall, \breve{D}, u_j) \mid j = 1, \ldots, d]\), \(\mathcal{E}' = E[j \mapsto (\forall, \breve{D}, u_j) \mid j = 1, \ldots, d]\), and condition \(\bar{u} \text{ on } R \cdot D\) holds iff \(\bar{u} \cap R^o = \emptyset\)

\[
wd_{E}^{p,\psi}(\exists D(\bar{x}).\phi) = \bigvee_{\bar{u} \text{ on } R \cdot D} \text{wd}_{E}^{p,\psi}(\phi)
\]

where \(\mathcal{E}' = E[j \mapsto (\exists, \breve{D}, u_j) \mid j = 1, \ldots, d]\), \(\mathcal{E}' = E[j \mapsto (\exists, \breve{D}, u_j) \mid j = 1, \ldots, d]\), and condition \(\bar{u} \text{ on } R \cdot D\) holds iff \(\bar{u} \cap R^o = \emptyset\)

Definition 20 considers the cases where the existence of certain types of edges in
the graph \( p \) will be applied to affects the validity of the application of \( p \) if the nodes attached to them are not as specified by \( \varphi \). Therefore, if \( wd \) does not return \( \perp \) or \( \text{noEdge}(\_\_\_) \) then the weakest pre-condition requires that we return \( \varphi \) (or part of \( \varphi \)) as the weakest pre-condition.

In simple terms \( wp \) returns \( \phi \) for any graph \( G \) that should satisfy the post-condition \( \phi \) in order for \( G' \models \phi \) where \( G' \) is the graph obtained after applying the production to \( G \).

**Definition 20** (\( wp \) predicate transformer). Similarly to Definition 19, let \( eq_{p,x_1=x_2}(E) \) be as the auxiliary map given in Definition 17 but for the last case where instead of returning \( x_1 = x_2 \), \( eq_{p,x_1=x_2}(E) \) returns \( \psi_3 \).

\[
\begin{align*}
wp_{E,h}^p(x_1 = x_2) &= eq_{p,x_1=x_2}^{\text{noEdge}(D_2),\text{noEdge}(D_1)}(E) \\
\text{where } D_1 &= E(2)(x_1) \text{ and } D_2 = E(2)(x_2) \\
wp_{E,h}^p(x_1 \neq x_2) &= \neg eq_{p,x_1=x_2}(E) \\
wp_{E,h}^p(\top) &= \top \\
wp_{E,h}^p(\bot) &= \bot \\
wp_{E,h}^p(\phi \land \phi') &= wp_{E,h}^p(\phi) \land wp_{E,h}^p(\phi') \\
wp_{E,h}^p(\phi \lor \phi') &= wp_{E,h}^p(\phi) \lor wp_{E,h}^p(\phi')
\end{align*}
\]

Let \( D \) have arity \( d \), \( \{v_1, \ldots, v_d\} \subseteq \bar{R} \) be a fixed set of \( d \) (representative) external nodes,
\[ \tilde{u} \in (V_R \cup \{v_1, \ldots, v_d\})^*, \tilde{x} = x_1, \ldots, x_d, \text{ and } l = 1 + \max, \]

\[
w_{p,h}^{\forall D}(\tilde{x}, \phi) = \bigwedge_{\tilde{u} \text{ on } R \cdot D} \forall D(\tilde{x}).w_{E',h}[\tilde{x} \mapsto \tilde{u}](\phi) \]

where \[ E' = E[x_j \mapsto \forall D, u_j] \mid j = 1, \ldots, d, \]

and condition \( \tilde{u} \) on \( R \cdot D \) holds iff \( \tilde{u} \cap R^c = \emptyset \)

\[
w_{p,h}^{\exists D}(\tilde{x}, \phi) = \bigvee_{\tilde{u} \text{ on } R \cdot D} (\exists D(\tilde{x}).w_{E',h}[\tilde{x} \mapsto \tilde{u}](\phi) \vee w_{E'}(\phi)) \]

where \[ E' = E[x_j \mapsto \exists D, u_j] \mid j = 1, \ldots, d, \]

and condition \( \tilde{u} \) on \( R \cdot D \) holds iff \( \tilde{u} \cap R^c = \emptyset \)

The most interesting cases in Definition 19, and Definition 20 are the ones for equality \( x_1 = x_2 \) dealt by the auxiliary map of Definition 17.

- If both \( x_1 \) and \( x_2 \) are existentially quantified and assigned to the same internal nodes of \( p \), the calculated weakest pre-condition is \( \top \); in fact, whatever graph the production is applied to, the post-condition would be guaranteed by the RHS of \( p \).

- Instead \( \bot \) is returned when say \( x_1 \) is universally quantified and either
  
  (i) \( x_2 \) is existentially quantified and assigned to an interface node, or
  
  (ii) it is assigned to an internal node of \( R \) different from the one assigned to \( x_2 \).

Note that in (i) if \( x_2 \) were universally quantified, there might be a chance to guarantee the equality if no edges of the type quantifying the variables were present in the graph \( p \) is applied to. In fact, \( eq_{p,x_1=x_2}^{\psi_1,\psi_2}(E) \) returns \( \bot \) if (i) \( x_1 \) is mapped to a fresh node in the RHS of \( p \) (i.e., an internal node of \( p \)) while \( x_2 \) is
mapped to a node outside \( p \) or (ii) if they are both mapped to two fresh nodes of
the RHS of \( p \), because the semantics of ADR does not allow such identifications
on the internal nodes of a production.

- The equality \( x_1 = x_2 \) may hold if \( x_1 \) and \( x_2 \) are mapped to the same internal node
  provided that no edge in the graph, to which \( p \) is applied, is typed as the type of
  the edges insisting on the variables, otherwise the universal quantification will be
  spoiled.

- Likewise, if both variables are universally quantified but one is internal and the
  other is external (not in \( p \)), then the weakest pre-condition returns \( \text{noEdge}\langle D \rangle \)
  where \( D \) is the type of the external variable. Intuitively, the graph resulting
  from the application of \( p \) to a graph with an \( e \) edge of type \( D \), would violate the
  quantification of \( x_1 \) and \( x_2 \) since \( e \) cannot insist on fresh nodes introduced by \( p \).

- In all other cases, \( wp_{\mathcal{C},\mathcal{h}}^{p,h}(x_1 = x_2) \) requires the initial graph to satisfy the same
  equality on the nodes corresponding to the variables of the post-condition; this
  requires that if either \( x_1 \) and \( x_2 \) are assigned to an interface node (that is \( h(x_j) \in
  Im(i) \)) it has a counterpart variable \( z \in \{z_1,\ldots,z_m\} \) mapped (through \( \mathcal{h} \)) on the
  node \( i^{-1}(x_1) \) or \( i^{-1}(x_2) \) in \( L \).

The remaining cases are trivial but for the quantifications \( \forall D(\bar{x}).\phi \) and \( \exists D(\bar{x}).\phi \)
where
the computed pre-conditions require \( \phi \) to be satisfied under any “reasonable” assign-
ment to \( \bar{x} \) for the universal quantification or one “reasonable” assignment to \( \bar{x} \) for the
existential quantification; this means that such variables are assigned in any possible
way either to nodes in \( R \) or to a fixed set of nodes \( v_1,\ldots,v_d \) outside \( R \). The identity of

5.3. Extracting contracts for ADR productions
v_1, \ldots, v_d$ is immaterial, in fact, the crucial point is that they refer to nodes outside $R$ (i.e., as many as the variables in $\tilde{x}$).

**Proposition 1.** If $\varphi$ and $\varphi'$ are logically equivalent $\mathcal{L}$-formulae, then $wd_{E,\bar{h}}^{p,\psi}(\varphi)$ (resp. $wp_{E,\bar{h}}^{p,\bar{h}}(\varphi)$) is logically equivalent to $wd_{E,\bar{h}}^{p,\psi}(\varphi')$ (resp. $wp_{E,\bar{h}}^{p,\bar{h}}(\varphi')$).

The next example shows how to compute weakest pre-conditions.

**Example 25.** Consider $\varphi \in \mathcal{L}$ and the production $p$ below:

\[
\varphi \overset{\text{def}}{=} \forall B(x, y). \forall C(z). y = z \quad \quad p \overset{\text{def}}{=} \begin{array}{c}
A \\
\circ \\
\circ \\
\circ \\
\circ \\
\circ \\
\circ \\
\circ \\
\end{array}
\]

Let $\bar{h} = w \mapsto v$. The first step (in finding the weakest pre-condition) (i) computes

\[
W_{\bar{h},0}^0(p, \varphi) \overset{\text{def}}{=} wd_{p,\top}^0(\varphi) \land wp_{p,\bar{h}}^{0,\bar{h}}(\varphi)
\]

and (ii) applies the quantification case in Definition 20 so to yield

\[
\left( \bigwedge_{j=1,2,3} wd_{E_j,\bar{h}}^{p,\psi}(\varphi') \right) \land \left( \bigwedge_{j=1,2,3} \forall B(x, y). wp_{E_j,\bar{h}}^{p,\bar{h}}(\varphi') \right)
\]

given that $E_1 = \{ x \mapsto (\forall, B, u_1), y \mapsto (\forall, B, u) \}$, $E_2 = \{ x \mapsto (\forall, B, u_1), y \mapsto (\forall, B, v_1) \}$ and $E_3 = \{ x \mapsto (\forall, B, v_1), y \mapsto (\forall, B, v_2) \}$ are the only assignments to consider (since $v_1$ and $v_2$ are representative nodes outside the RHS of $p$ while $u_1$ is the unique node on the interface, and $u$ is the unique internal node).

The second step (in finding the weakest pre-condition) applies again this case for $\forall C(z)$ (for both $wd_{E_j,\bar{h}}^{p,\psi}(\varphi')$ and $wp_{E_j,\bar{h}}^{p,\bar{h}}(\varphi')$) and yields

\[
\left( \bigwedge_{j,k=4,5} wd_{E_{j,k},\bar{h}}^{p,\psi}(\varphi'') \right) \land \left( \bigwedge_{j,k=4,5} \forall B(x, y). \forall C(z). wp_{E_{j,k},\bar{h}}^{p,\bar{h}}(\varphi'') \right)
\]

5.3. Extracting contracts for ADR productions
where $\mathcal{E}_4 = \{ z \mapsto (\forall, \mathcal{C}, u_1) \}$ and $\mathcal{E}_5 = \{ z \mapsto (\forall, \mathcal{C}, v_1) \}$; in fact there is no edge of type $\mathcal{C}$ in the RHS of $p$ (hence $v_1$ is representative external node and $u_1$ is its unique interface node).

Finally, applying the auxiliary map $eq_{\psi_1,\psi_2}(\mathcal{E})$ for node equality, we get

$$\bigwedge_{j,k} wd_{\mathcal{E}_j \cup \mathcal{E}_k}^{p,\psi}(\varphi'') = (\top \land \text{noEdge}(\mathcal{C})) \land (\top \land \top) \land (\top \land \top) = \text{noEdge}(\mathcal{C}) \quad (5.2)$$

$$\bigwedge_{j,k} Q_{\langle \mathcal{E}_j \cup \mathcal{E}_k, \emptyset \rangle}(\varphi'') = Q[\text{noEdge}(\mathcal{C})] \land Q[y = z] \quad (5.3)$$

where in $(5.3)$ $Q[\_]$ is the context $\forall B(x,y).\forall C(z),[\_]$. Note that, the weakest preconditions is the conjunction of $(5.2)$ and $(5.3)$, that is

$$W_{\emptyset,\mathcal{E}}(\varphi) = \text{noEdge}(\mathcal{C}) \land Q[\text{noEdge}(\mathcal{C})] \land Q[y = z]$$

This is consistent with the fact that $\varphi$ can only be satisfied by graphs that do not have any edges of type $\mathcal{C}$ due to the internal node $u$ introduced by $p$.

The correctness of the algorithm is established by showing the validity of the asserted production $\{ W_{h,\mathcal{E}}^{\tilde{h}}(p, \varphi), h \} \ p \{ \varphi, h \}$ (Theorem 1) and that the $W_{h,\mathcal{E}}^{\tilde{h}}(p, \varphi)$ is the weakest precondition for $p$ and $\varphi$ (Theorem 2).

Function $C$ takes $\varphi \in \mathcal{L}$ and an environment $\mathcal{E}$ and returns a new formula that takes into account the quantifications in $\mathcal{E}$ of the free variables of $\varphi$.

**Definition 21** (Formulae in context). Let $\varphi \in \mathcal{L}$ and $\mathcal{E}$ be an environment as in
Definition 16. Define

\[ C(\mathcal{E}, \varphi) = C(\mathcal{E} \{ y \mapsto \langle \hat{q}', D, \_ \rangle \mid y \in \mathcal{E}, qD(x'_1, \ldots, x'_d).\varphi \}) \] (5.4)

where \[ z = \max \{ z' \mid \exists y \in \mathcal{V}, q' \in \{ \forall, \exists \} : \mathcal{E}(y) = \langle z', q, D, \_ \rangle \} \]

and \[ x'_j = \begin{cases} z & \text{if } z \in \mathcal{E} \mid \mathcal{E}(z) = \langle z, q, D, \_ \rangle \text{ as}\; \mathcal{E} = \langle \varphi, h \rangle \; \text{otherwise} \end{cases} \]

and define \( C(\mathcal{E}, \varphi) = \varphi \), if \( \mathcal{E} = \emptyset \).

Notice that \( C \) is well defined as (5.4) in Definition 21 does not depend on the choice of the variables.

Theorem 1. Let \( p = \langle L, R, i \rangle \) be a production, \( \varphi \in L \), \( h : \text{fv}(\varphi) \to V_R \) be injective, \( \bar{h} : Z \to V_L \) be a bijection where \( Z \subseteq V \) disjoint from the variables in \( \varphi \), \( \mathcal{E} \) be an environment compatible with \( h \), and \( \pi \) be the asserted production \( \{ \mathcal{W}_{\bar{h}, \mathcal{E}}^h(p, \varphi), \bar{h} \} \; p \{ \varphi, h \} \). For any ADR graph \( G \) and the morphism \( \sigma \) from \( L \) to \( G \), if \( G \setminus \sigma(E_L) \models_{\mathcal{E}} \mathcal{W}_{\bar{h}, \mathcal{E}}^h(p, \varphi) \) then \( \pi(G, \sigma) \models_{\mathcal{E}} C(\mathcal{E}, \varphi) \).

Proof (sketch) The proof is by induction on the structure of \( \varphi \). The details are presented in Section 5.4.

Theorem 2. Given \( \varphi \in L \), let \( Z \subseteq V \) such that no variables in \( \varphi \) is in \( Z \) and \( \bar{h} : Z \to V_L \) be a bijection, then for any formula \( \psi \) such that \( \{ \psi, h \} \; p \{ \varphi, \bar{h} \} \) is a valid production for any graph \( G \) then \( \psi \) implies \( \mathcal{W}_{\bar{h}, \mathcal{E}}^h(p, \varphi) \).

Proof (sketch) The proof is by induction on the structure of \( \mathcal{W}_{\bar{h}, \mathcal{E}}^h(p, \varphi) \). The details are presented in Section 5.4.
5.4 Correctness of the algorithm

In the proofs of Theorems 1, and 2 we use the following observation.

Observation 1. By Definitions 19, and 20, the algorithm computes the pre-condition of a quantified formula by considering every “reasonable” assignment of the quantified variables. More precisely, if the post-condition is a quantification $\forall D(\tilde{x}).\phi$ (resp. $\exists D(\tilde{x}).\phi$), the computed weakest pre-condition requires $\varphi'$ to be satisfied under all (resp. some) assignments of each variable in $\tilde{x}$ to nodes in $R$ or to a fixed set of nodes $v_1, \ldots, v_n$ outside $R$. The identity of $v_1, \ldots, v_n$ is immaterial, in fact, the crucial point is that they refer to nodes outside $R$ (i.e., as many as the variables in $\tilde{x}$).

Lemma 1. Given a production $p$. If $G \models \text{noEdge}(D)$ and $G'$ is the application of $p$ to $G$, then the only edges of type $D$ are those introduced by $R$.

5.4.1 Proofs for Theorem 1

For the sake of readability, we repeat the statement of Theorem 1.

Theorem 1. Let $p = \langle L, R, i \rangle$ be a production, $\varphi \in L$, $h : \text{fv}(\varphi) \to V_R$ be injective, $\bar{h} : Z \to V_L$ be a bijection where $Z \subseteq V$ disjoint from the variables in $\varphi$, $\mathcal{E}$ be an environment compatible with $h$, and $\pi$ be the asserted production $\{W^h_{\mathcal{E}}(p, \varphi), \bar{h}\} P \{\varphi, h\}$. For any ADR graph $G$ and the morphism $\sigma$ from $L$ to $G$, if $G \setminus \sigma(E_L) \models_{\mathcal{E}} W^h_{\mathcal{E}}(p, \varphi)$ then $\pi(G, \sigma) \models_{\mathcal{E}} C(\mathcal{E}, \varphi)$.

Proof 1. The proof is by induction on the structure of the post-condition $\varphi$.

$\varphi$ is $\top$: we have $W^h_{\mathcal{E}}(p, \varphi) = \top$ and therefore the thesis follows trivially, because any graph satisfies $\top$.
\( \varphi \) is \( \bot \): we have \( \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi) = \bot \) and, since no graph satisfies \( \bot \), the implication of the statement vacuously hold.

\( \varphi \) is \( \varphi_1 \land \varphi_2 \): by definition, \( \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi) = \psi_1 \land \psi_2 \) where \( \psi_1 = \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi_1) \) and \( \psi_2 = \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi_2) \). By inductive hypothesis, \( G \setminus \sigma(E_L) \models \mathcal{E} \psi_i \) implies \( \pi(G, \sigma) \models \mathcal{E} C(\varphi_j) \) for \( j = 1, 2 \). The thesis follows by the definition of \( \models \) (cf. Definition 13).

\( \varphi \) is \( \forall D(\bar{x}).\varphi' \): let \( d(\bar{u}) \in \pi(G, \sigma) \) such that \( \tau_{\pi(G,\sigma)}(d) = D \) (i.e., the type of \( d \) is \( D \)) and \( \psi = \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi') \). By induction, \( G \setminus \sigma(E_L) \models \mathcal{E} \psi \) implies that \( \pi(G, \sigma) \models \mathcal{E} C(\varphi') \).

The thesis follows by observing that there are only finitely many edges \( d(\bar{u}) \) satisfying the property above and by the fact that, by Definitions 19, and 20, \( \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi) \) is obtained by the conjunction of

\[
\begin{align*}
wd^p_{\mathcal{E}}(\varphi) &= \bigwedge_{\bar{u} \in R \cdot D} wd^p_{\mathcal{E}}^\psi(\varphi') \\
wp^p_{\mathcal{E},h}(\varphi) &= \bigwedge_{\bar{u} \in R \cdot D} \forall D(\bar{x}).wp^p_{\mathcal{E}}[\bar{x} \mapsto (\forall, D, \bar{u})]_{h[\bar{x} \mapsto \bar{u}]}(\varphi')
\end{align*}
\]

for some \( \mathcal{E} \).

\( \varphi \) is \( \exists D(\bar{x}).\varphi' \): For a contradiction, suppose that \( \pi(G, \sigma) \not\models \mathcal{E} C(\varphi') \) for any tuple of nodes \( \bar{u} \) such that \( d(\bar{u}) \in \pi(G, \sigma) \) with \( \tau_{\pi(G,\sigma)}(d) = D \) (i.e., the type of \( d \) is \( D \)). Fix such a tuple \( \bar{u} \), assign \( \bar{u} \) in \( \mathcal{E} \) to \( \bar{x} \) and let \( \psi' = \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi') \). By induction, we have that \( G \setminus \sigma(E_L) \models \mathcal{E} \psi' \) implies that \( \pi(G, \sigma) \models \mathcal{E} C(\varphi') \) which yields a contradiction. Hence, there should exist an edge \( d(\bar{u}) \in \pi(G, \sigma) \) such that \( \pi(G, \sigma) \models \mathcal{E} C(\varphi') \). The thesis follows by observing that, by Definitions 19, and 20, in this case \( \mathcal{W}_{h,\mathcal{E}}^h(p, \varphi) \)
returns the conjunction of

\[
wd_{\mathcal{E}}^{p,\psi}(\varphi) = \bigvee_{\tilde{u} \text{ on } R \cdot D} wd_{\mathcal{E}',\tilde{u}}^{p,\psi}(\varphi') \quad \text{and}
\]
\[
wP_{\mathcal{E},h}^{p,h}(\varphi) = \bigvee_{\tilde{u} \text{ on } R \cdot D} (\exists D(\tilde{x}) \cdot wP_{\mathcal{E}',h[\tilde{x} \rightarrow (\exists, D, \tilde{u})]}(\varphi') \lor wd_{\mathcal{E}}^{p,\psi}(\varphi'))
\]

for some \( \mathcal{E} \). Observe that \( wd_{\mathcal{E}}^{p,\psi}(\varphi') \) returns \( \top \) when the RHS of \( p \) satisfies the post-condition \( \varphi \). For an assignment in which this is the case, the disjunct is just \( \top \) since, the graph resulting from the application of \( p \) would guaranteed the post-condition due to the instantiation of the RHS of \( p \) under such an assignment.

In all the other cases, the disjunct is \( \bigvee_{\tilde{u} \text{ on } R \cdot D} (\exists D(\tilde{x}) \cdot wp_{\mathcal{E}',h[\tilde{x} \rightarrow (\exists, D, \tilde{u})]}(\varphi') \lor wd_{\mathcal{E}}^{p,\psi}(\varphi')) \). Hence, the pre-condition for an existential quantification requires the existence of suitable edges in the graph to which \( p \) is applied to whenever an assignment does not make the RHS of \( p \) to satisfy the post-condition.

\( \varphi \) is \( x_1 = x_2 \): let \( \mathcal{E} \) be an environment as in Definitions 19, and 20 and recall that variables \( x_1 \) and \( x_2 \) are indexed according to the order in which they are quantified in \( \varphi \) (if at all). By definition the algorithm invokes the auxiliary map of Definition 17 and returns

\[
eq_{p,x_1=x_2}^{\text{noEdge}(D_2),\text{noEdge}(D_1,D_2)}(\mathcal{E})
\]

Hence it suffices to consider all the cases of the auxiliary map. In the following cases, recall also Observation 1 where, depending on the quantification, the algorithm will return the conjunction or disjunction of all the possible assignments of the quantified variables.
• If (5.5) is $\bot$ then the thesis follows trivially.

• If (5.5) is $\top$ then one of the following holds:
  
  – both $x_1$ and $x_2$ are existentially quantified and mapped to the same internal node of $R$
  
  – one of the variables is existentially quantified and mapped to an internal node of $R$ and the other is a free variable mapped to the same node
  
  – $x_1$ is universally quantified, $x_2$ is existentially quantified and both of them are mapped to the same internal node in $R$
  
  – both variables are free variables mapped to the same internal node of $R$.

Therefore, any graph obtained by applying the production will have edges of the appropriate type sharing the node assigned to $x_1$ and $x_2$ since such edges and nodes are introduced by the RHS $R$ of the production $p$.

• If (5.5) is $\text{noEdge}(D_2)$ then one of the following cases holds:

  – $E^{(1)}(x_1) \uparrow$, $E(x_2) \equiv \forall D_2 \ R^\circ$, and the two variables are mapped to the same internal node. If $G \setminus \sigma(E_L)$ has an edge of type $D_2$ then $G \setminus \sigma(E_L) \not\equiv \text{noEdge}(D_2)$ and the thesis follows vacuously. By Lemma 1, $\pi(G, \sigma)$ will satisfy $\varphi$ if all the edges of type $D_2$ in $R$ are attached to $E^{(3)}(x_2)$; that is, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_2$ has to hold. If one of such edges is not attached to $E^{(3)}(x_2)$ then by Observation 1 the computed pre-condition would be $\bot$ contrary to the hypothesis.

  – $E^{(1)}(x_1) \uparrow$ and $E(x_2) \equiv \forall D_2 \ R$. If $G \setminus \sigma(E_L)$ has an edge of type $D_2$, then as before $G \setminus \sigma(E_L) \not\equiv \text{noEdge}(D_2)$ and the thesis follows vacuously.
By Lemma 1, $\pi(G, \sigma) \pi(G, \sigma)$ satisfies $x_1 = x_2$ (in $\mathcal{E}$) if all the edges of type $D_2$ in $R$ are attached to $\mathcal{E}^{(3)}(x_1)$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, (5.5) would return $\perp$ instead of $\text{noEdge}(D_2)$.

- $\mathcal{E}(x_1) \models \forall D_1 R^c$ and $\mathcal{E}(x_2) \models \forall D_2 \tilde{R}$. In this case, $L$, the LHS of $p$, has a type different from $D_2$ (otherwise the morphism $\sigma$ could not exist). Hence, if $G$ has an edge of type $D_2$ then $G \setminus \sigma(E_L) \not\models \text{noEdge}(D_2)$ and the thesis follows vacuously. By Lemma 1, $\pi(G, \sigma)$ satisfies $x_1 = x_2$ (in $\mathcal{E}$) if all the edges of type $D_2$ in $R$ are attached to $\mathcal{E}^{(3)}(x_1)$; in fact, as in the previous case, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_2$ has to hold.

- $\mathcal{E}(x_1) \models \exists D_1 n, \mathcal{E}(x_2) \models \forall D_2 n$, and $n \in R^c$. If $G \setminus \sigma(E_L)$ has an edge of type $D_2$ then $G \setminus \sigma(E_L) \not\models \text{noEdge}(D_2)$ and the thesis follows vacuously. By Lemma 1, $\pi(G, \sigma)$ will satisfy $x_1 = x_2$ if there is no other edge of type $D_2$ in $R$ not attached to $n$ (recall Observation 1).

- (5.5) is $\text{noEdge}(D_1, D_2)$ when $\mathcal{E}(x_1) \models \forall D_1 n, \mathcal{E}(x_2) \models \forall D_2 n$ and $n \in R^c$. Since edges of type $D_1$ and $D_2$ are introduced by $R$ as specified by the mapping and $n$ is a fresh internal node then there can be no more edges of these types in $G \setminus \sigma(E_L)$. If $G \setminus \sigma(E_L)$ has an edge of type $D_1$ or $D_2$ then $G \setminus \sigma(E_L) \not\models \text{noEdge}(D_1, D_2)$ and the thesis follows vacuously. If $G \setminus \sigma(E_L) \models \text{noEdge}(D_1, D_2)$, then the only edges of type $D_1$ and $D_2$ in $\pi(G, \sigma)$ are those introduced by the RHS of $p$ and if all of them are attached to $n$ as checked by the conjunction on every possible mappings $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$.  

5.4. Correctness of the algorithm
Then, by Observation 1, $\pi(G, \sigma) \models \varphi$ otherwise the pre-condition would be $\bot$, contrary to the hypothesis.

- (5.5) is $x_1 = x_2$ in the following cases:
  
  - $\mathcal{E}(x_1) \equiv \exists D_1 n, \; \mathcal{E}(x_2) \equiv \exists D_2 n'$ and $n \notin R^c$ or $n' \notin R^c$. If $G \setminus \sigma(E_L) \not\models_{\mathcal{E}(3)} x_1 = x_2$ (i.e., $x_1$ and $x_2$ are assigned to different nodes in $\mathcal{E}$) then the thesis follows vacuously. If $G \setminus \sigma(E_L) \models_{\mathcal{E}(3)} x_1 = x_2$, then, due to the existential quantifications, $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 = x_2$ holds if there is an assignment mapping $x_1$ and $x_2$ to the same node (by Observation 1). Since $\pi(G, \sigma)$ is the graph obtained by the union of $G \setminus \sigma(E_L)$ and $R$ then $\pi(G, \sigma)$ ”inherits” from $G \setminus \sigma(E_L)$ the node assigned to $x_1$ and $x_2$ and hence satisfies the invariant.

  - $\mathcal{E}(x_1) \equiv \forall D_1 n, \; \mathcal{E}(x_2) \equiv \forall D_2 n'$, and $n, n' \notin R^c$. If there is no node assigned to $x_1$ in $G \setminus \sigma(E_L)$ such that every assignment on $x_2$ is equal to $x_1$ then $G \setminus \sigma(E_L) \not\models_{\mathcal{E}(3)} x_1 = x_2$ and the thesis follows vacuously. If $G \setminus \sigma(E_L) \models_{\mathcal{E}(3)} x_1 = x_2$, then $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 = x_2$ since there are no other edges of type $D_2$ in $R$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_2$ would not hold otherwise, by Observation 1 and therefore the computed pre-condition would have been $\text{noEdge}(D_2)$ and not $x_1 = x_2$ as by hypothesis.

  - $\mathcal{E}(x_1) \equiv \forall D_1 n, \; \mathcal{E}(x_2) \equiv \forall D_2 n'$, and $n, n' \notin R^c$. If $n \neq n'$ then $G \setminus \sigma(E_L) \not\models_{\mathcal{E}(3)} x_1 = x_2$ and the thesis follows vacuously. If $G \setminus \sigma(E_L) \models_{\mathcal{E}(3)} x_1 = x_2$, then $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 = x_2$ due to the fact that there are no other edges of type $D_1$ and $D_2$ in $R$; in fact, the conjunction on every possible
mapping $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\text{noEdge}(D_1, D_2)$ and not $x_1 = x_2$ as by hypothesis.

- $E(x_1) \overset{\text{as}}{=} \forall D_1 n, E(x_2) \overset{\text{as}}{=} \exists D_2 n'$ and $n \notin R^\circ$ If there is a node assigned to $x_1$ in $G \setminus \sigma(E_L)$ such that no assignment on $x_2$ is equal to $x_1$ then $G \setminus \sigma(E_L) \not\models_{E(3)} x_1 = x_2$ and the thesis follows vacuously. If $G \setminus \sigma(E_L) \models_{E(3)} x_1 = x_2$ then $\pi(G, \sigma) \models_{E(3)} x_1 = x_2$ if there in a node assigned to $x_1$ in $R$ not equal to any node assigned to $x_2$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $x_1 = x_2$ as by hypothesis.

$\varphi$ is $x_1 \neq x_2$: let $E$ be an environment as in Definitions 19, and 20. By definition the algorithm invokes the auxiliary map of Definition 17 and returns

$$-\neg eq_{p,x_1=x_2}^\top(E) \quad (5.6)$$

Hence it suffices to consider all the possible cases of the auxiliary map.

- If (5.6) is $\bot$ then the thesis follows trivially.

- If (5.6) is $\top$ when one of the following cases occur:

  - $E(x_1) \overset{\text{as}}{=} \exists D_1 R^\circ, E^{(1)}(x_2) \uparrow$ and $E^{(3)}(x_1) \neq E^{(3)}(x_2)$ If the node assigned to $x_1$ by $E^{(3)}$ is not equal to the free node of the post-condition mapped to $x_2$ then the invariant is satisfied regardless of $G \setminus \sigma(E_L)$ due to the

5.4. Correctness of the algorithm
existential quantification on $x_1$ and the algorithm returns $\top$. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ since the instance of $R$ that replaces $L$ in $G$ guarantees the existential quantification of the inequality.

- If $\mathcal{E}(x_2) \overset{\alpha}{=} \forall D R^\circ$, $\mathcal{E}^{(1)}(x_2) \uparrow$ and $\mathcal{E}^{(3)}(x_1) \neq \mathcal{E}^{(1)}(x_2) \uparrow$. If every node in $R$ assigned to $x_1$ by $\mathcal{E}^{(3)}$ is not equal to the free node of the post-condition mapped to $x_2$ then the invariant is satisfied as long as there is no other edge of type $D$ in $G \setminus \sigma(E_L)$. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if there are no other edges of type $D$ in $R$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\mathsf{noEdge}(D)$ and not $\top$ as by hypothesis.

- As in the previous case if variable $x_1$ is universally quantified and this time mapped to either an interface node or an external node while $\mathcal{E}^{(1)}(x_2) \uparrow$ then $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if there is no other edge of type $D$ in $R$ attached to $x_2$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $\top$ as by hypothesis.

- $\mathcal{E}(x_1) \overset{\alpha}{=} \exists D_1 R^\circ$, $\mathcal{E}(x_2) \overset{\alpha}{=} \forall D_2 n$ and $n \notin R^\circ$. Since the node assigned by $\mathcal{E}^{(3)}$ to $x_1$ is internal and to $x_2$ is external then they are disjoint and the algorithm returns $\top$ for this case. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if there is no node in $R$ assigned to $x_2$ by $\mathcal{E}^{(3)}$ equal to the node assigned to $x_1$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1,
the computed pre-condition would have been $x_1 \neq x_2$ and not $\top$ as by hypothesis.

- $\mathcal{E}(x_1) \overset{\text{as}}{=} \forall \_ \cdot R^1$, $\mathcal{E}(x_2) \overset{\text{as}}{=} \exists \_ \cdot \neg n$ and $n \notin R$ then like the previous case $x_1$ and $x_2$ are distinct nodes and the algorithm returns $\top$ for this case. $G \setminus \sigma(EL) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if there is no node assigned to $x_1$ by $\mathcal{E}(3)$ equal to the node assigned to $x_2$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $x_1 \neq x_2$ and not $\top$ as by hypothesis.

- $\mathcal{E}(x_1) \overset{\text{as}}{=} \forall \_ \cdot R^0$, $\mathcal{E}(x_2) \overset{\text{as}}{=} \forall \_ \cdot R^0$ and $\mathcal{E}(3)(x_1) \neq \mathcal{E}(3)(x_2)$ then $x_1$ and $x_2$ are mapped by $\mathcal{E}(3)$ to distinct nodes in this case and the algorithm returns $\top$. $G \setminus \sigma(EL) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if there are no other nodes assigned to $x_1$ and $x_2$ such that they are equal; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $\top$ as by hypothesis.

- $\mathcal{E}(x_1) \overset{\text{as}}{=} \forall \_ \cdot R^0$, $\mathcal{E}(x_2) \overset{\text{as}}{=} \exists \_ \cdot R^0$ and $\mathcal{E}(3)(x_1) \neq \mathcal{E}(3)(x_2)$ then $x_1$ and $x_2$ are mapped by $\mathcal{E}(3)$ to distinct nodes in this case and the algorithm returns $\top$. $G \setminus \sigma(EL) \models \top$ and the obtained $\pi(G, \sigma) \models_{\mathcal{E}(3)} x_1 \neq x_2$ if for every node assigned to $x_1$ there is a different node assigned to $x_2$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $\top$ as by hypothesis.
If $E(x_1) \equiv \forall \ D \text{Im}(i)$, $E(x_2) \equiv \exists \ R$ and $D \in R$ this means that an edge of type $D$ is in $R$ and the mapped node to $x_1$ is not equal to $x_2$ and in this case the algorithm returns $\top$. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{E(3)} x_1 \neq x_2$ if for every node assigned to $x_1$ there is a different node assigned to $x_2$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $\top$ as by hypothesis.

- $E(x_1) \equiv \exists \ R$ and $E(x_2) \equiv \forall \ R^\circ$ then since $x_2$ is a fresh internal node this means that it cannot be equal to the external node mapped to $x_1$ and in this case the algorithm returns $\top$. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{E(3)} x_1 \neq x_2$ if there is a node assigned to $x_1$ distinct to every node assigned to $x_2$ by $E^{(3)}$; in fact, the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D_1$ and $\tilde{u}$ on $R \cdot D_2$ has to hold otherwise, by Observation 1, the computed pre-condition would have been $\bot$ and not $\top$ as by hypothesis.

- If $E(x_1) \equiv \exists \ R^\circ$, $E(x_2) \equiv \exists \ R^\circ$ and $E^{(3)}(x_1) \neq E^{(3)}(x_2)$. If the node assigned to $x_1$ by $E^{(3)}$ is not equal to the node assigned to $x_2$ then the invariant is satisfied regardless of $G \setminus \sigma(E_L)$ due to the existential quantification on both $x_1$ and $x_2$ and therefore algorithm returns $\top$. $G \setminus \sigma(E_L) \models \top$ and the obtained $\pi(G, \sigma) \models_{E(3)} x_1 \neq x_2$ since the instance of $R$ that replaces $L$ in $G$ guarantees the existential quantification of the inequality.

5.4. Correctness of the algorithm
5.4.2 Proofs for Theorem 2

For the sake of readability, we repeat the statement of Theorem 2.

**Theorem 2.** Given $\varphi \in \mathcal{L}$, let $Z \subseteq V$ such that no variables in $\varphi$ is in $Z$ and $\bar{h} : Z \to V_L$ be a bijection, then for any formula $\psi$ such that $\{\psi, h\} \models \{\varphi, \bar{h}\}$ is a valid production for any graph $G$ then $\psi$ implies $W^{\bar{h}}_{h, \mathcal{E}}(p, \varphi)$.

**Proof 2.** $W^{\bar{h}}_{h, \mathcal{E}}(p, \varphi)$ is $\top$: This case is trivial.

$W^{\bar{h}}_{h, \mathcal{E}}(p, \varphi)$ is $\bot$: when either $\varphi$ is $\bot$, or when $\varphi$ is $x_1 = x_2$ s.t. cond$_\bot$ (Definition 17) holds. Recall that $\sigma$ is the morphism from the LHS of a production to the graph $G$ and that $\pi(G, \sigma)$ refers to the graph obtained after the application of the production.

$\varphi$ is $\bot$: we show that $\psi$ is equivalent to $\bot$; assume this is not the case. Then there is $G \setminus \sigma(E_L) \models_h \psi$ and, by the validity of $\{\psi, h\} \models \{\varphi, \bar{h}\}$ it follows that $\pi(G, \sigma) \models_h \varphi$ that is $G \setminus \sigma(E_L) \models_h \bot$, which is absurd.

$\varphi$ is $x_1 = x_2$: for any of the cases cond$_\bot$ in Definition 17 we show that $\psi$ is equivalent to $\bot$; assume this is not the case. By the following case analysis we observe that for every case in cond$_\bot$ since $W^{\bar{h}}_{h, \mathcal{E}}(p, \varphi) = \bot$ there is a graph $G$ s.t. $\pi(G, \sigma) \not\models_h \varphi$ and hence the production would not be valid if $G \setminus \sigma(E_L) \models_h \psi$. The first four cases bellow show that since the environment $\mathcal{E}$ is compatible to $h$ and $\pi(G, \sigma) \not\models_h x_1 = x_2$ then the production would not be valid if $G \setminus \sigma(E_L) \models_h \psi$. For the rest of the cases we assume the conjunction on every possible mapping $\tilde{u}$ on $R \cdot D$ returns $\bot$ and therefore as in the other cases the fact that $\mathcal{E}$ is...
compatible to \( h \) and that \( \pi(G, \sigma) \not\models_h x_1 = x_2 \) implies that if \( G \setminus \sigma(E_L) \models_h \psi \) then the production cannot be valid.

1. If \( E(x_1) \equiv D_1 R^\circ, E^{(1)}(x_2) \uparrow \) and \( E^{(3)}(x_1) \neq E^{(3)}(x_2) \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \)

2. If \( E(x_1) \equiv \forall D R^\circ, E^{(1)}(x_2) \uparrow \) and \( E^{(3)}(x_1) \neq E^{(1)}(x_2) \uparrow \) then \( \pi(G, \sigma) \not\models_E x_1 = x_2 \)

3. If \( E(x_1) \equiv \exists D \text{Im}(i), E^{(1)}(x_2) \uparrow \) then \( \pi(G, \sigma) \not\models_E x_1 = x_2 \)

4. If \( E(x_1) \equiv \exists \neg n, E^{(1)}(x_2) \uparrow \) and \( n \notin R^\circ \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \).

5. If \( E(x_1) \equiv \forall \neg R^\circ, E(x_2) \equiv \exists \neg n \) and \( n \notin R^\circ \) then since \( x_1 \) is an internal node and \( x_2 \) an external node they are distinct nodes and so \( \pi(G, \sigma) \not\models_h x_1 = x_2 \)

6. If \( E(x_1) \equiv \forall \neg R^\circ, E(x_2) \equiv \exists \neg R^\circ \) and \( E^{(3)}(x_1) \neq E^{(3)}(x_2) \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \)

7. If \( E(x_1) \equiv \forall D \text{Im}(i), E(x_2) \equiv \exists D \bar{R} \) and \( D_1 \in R \) this means that an edge of type \( D_1 \) is in \( R \) and the mapped node to \( x_1 \) is not equal to \( x_2 \) and therefore \( \pi(G, \sigma) \not\models_h x_1 = x_2 \)

8. If \( E(x_1) \equiv \exists \neg \bar{R} \) and \( E(x_2) \equiv \forall \neg R^\circ \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \) since \( x_2 \) is mapped to an internal node and \( x_1 \) to an external node.

9. If \( E(x_1) \equiv \exists \neg R^\circ, E(x_2) \equiv \exists \neg R^\circ \) and \( E^{(3)}(x_1) \neq E^{(3)}(x_2) \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \)

10. If \( E(x_1) \equiv \exists D_1 R^\circ, E(x_2) \equiv \forall D_2 n \) and \( n \notin R^\circ \) then \( \pi(G, \sigma) \not\models_h x_1 = x_2 \).

\( \mathcal{W}^h_{h, E}(p, \varphi) \text{ is noEdge} (D_1, D_2) \): If \( \psi \) does not imply \( \text{noEdge} (D_1, D_2) \), there should be a
graph $G$ such that $G \setminus \sigma(E_L) \models_h \psi \land \neg \text{noEdge}\langle D_1, D_2 \rangle$. This yields a contradiction since by the case analysis provided in the proofs for Theorem 1 when $G \setminus \sigma(E_L) \not\models_h \text{noEdge}\langle D_1, D_2 \rangle$ then $\pi(G, \sigma) \not\models_h \varphi$ under this hypothesis and hence the production would not be valid if $G \setminus \sigma(E_L) \models \psi$.

$W_{h,E}^\varphi(p, \varphi) \text{ is } \text{noEdge}\langle D_2 \rangle$: If $\psi$ does not imply $\text{noEdge}\langle D_2 \rangle$, there should be a graph $G$ such that $G \setminus \sigma(E_L) \models_h \psi \land \neg \text{noEdge}\langle D_2 \rangle$. This yields a contradiction since by the case analysis provided in the proofs for Theorem 1 when $G \setminus \sigma(E_L) \not\models_h \text{noEdge}\langle D_2 \rangle$ then $\pi(G, \sigma) \not\models_h \varphi$ under this hypothesis and hence the production would not be valid if $G \setminus \sigma(E_L) \models \psi$.

$W_{h,E}^\varphi(p, \varphi) \text{ is } x_1 = x_2$: This case is analogous to the previous one. If $\psi$ does not imply $x_1 = x_2$, there should be a graph $G$ such that $G \setminus \sigma(E_L) \models_h \psi \land \neg x_1 = x_2$. This yields a contradiction since by the case analysis provided in the proofs for Theorem 1 when $G \setminus \sigma(E_L) \not\models_h x_1 = x_2$ then $\pi(G, \sigma) \not\models_h \varphi$ under this hypothesis and hence the production would not be valid if $G \setminus \sigma(E_L) \models \psi$.

$W_{h,E}^\varphi(p, \varphi) \text{ is } x_1 \neq x_2$: This case is analogous to the previous one.

$W_{h,E}^\varphi(p, \varphi) \text{ is } \forall D(\tilde{x}).\psi'$: By the inductive hypothesis, $\varphi$ implies both $\varphi_1$ and $\varphi_2$. Hence, $\varphi$ implies $\varphi_1 \land \varphi_2 = W_{h,E}^\varphi(p, \varphi_1) \land W_{h,E}^\varphi(p, \varphi_2) = W_{h,E}^\varphi(p, \varphi)$, where the last equality holds by definition of our algorithm.

$W_{h,E}^\varphi(p, \varphi) \text{ is } \forall D(\tilde{x}).\varphi'$: By contradiction, let $\varphi = \forall D(\tilde{x}).\varphi'$ and recall that in this case it should be $\varphi = \forall D(\tilde{x}).\varphi''$ for some $\varphi'' \in \mathcal{L}$. If $\psi$ does not imply $\forall D(\tilde{x}).\varphi'$, there should be a graph $G$ and a mapping $h' : \text{fv}(\psi \land \varphi) \to V_G$ such that $G \models_{h'} \psi \land
\[ \neg \forall D(\bar{x}).\varphi', \text{ namely there is } e(\bar{u}) \in G \text{ such that } e \text{ is of type } D \text{ and } G \models_{h'[\bar{x}\rightarrow\bar{u}]} \psi \land \neg \varphi'. \] This yields a contradiction since by inductive hypothesis if \( G \setminus \sigma(EL) \not\models_{h'[\bar{x}\rightarrow\bar{u}]} \varphi' \) then \( \pi(G,\sigma) \not\models_{E} \varphi \) and if \( G \setminus \sigma(EL) \models_{h'[\bar{x}\rightarrow\bar{u}]} \varphi' \) then \( \psi \rightarrow \varphi' \).

\[ \mathcal{W}_{h,E}(p,\varphi) \text{ is } \exists D(\bar{x}).\varphi': \text{ This case is analogous to the previous one.} \]

### 5.5 AcDR in Action

We show how asserted productions can guarantee design properties by considering a simplified version of Java’s remote method invocation (JRMI) mechanism [44] and show how semantic conditions of JRMI can be modelled by asserted ADR productions. In short, JRMI is used to enable programmers to create distributed applications, in which the methods of remote Java objects can be invoked from other Java virtual machines, in a distributed environment. The asserted productions are obtained by using the algorithm in Section 5.3.

Figure 5.3 shows the type graph for JRMI which includes types as follows:
The intuition is that Java objects are either in non-remote classes \(O\) or in remote ones \(RO\). In the latter case, we want them to provide specific methods (i.e., \texttt{equals} and \texttt{hashCode}) to allow objects to be exported. Recall that in JRMI a remote object has to be \textit{exported} in order to make its methods remotely invokable. This can be done in two ways: (\(i\)) by extending the class \texttt{UnicastRemoteObject} or (\(ii\)) by explicitly implementing some of the \texttt{java.lang.Object} methods. To account for this, we use \(OM\) and the node type \(\bullet\) that explicitly models the overriding.

The productions of our model are in Figure 5.4. Production \texttt{obj2RUsingURO} is used to represent the Java capability to make local invocations to remote methods when those are not exported. Moreover, we provide productions to capture (\(i\)) and (\(ii\)) above. These possibilities are covered by three productions in Figure 5.4; \texttt{obj2RUsingOM} which con-
verts a local object to a remote one by explicitly implementing the `java.lang.Object`
methods, and `obj2RUUsingURO` and `extUnicast` which extend `UnicastRemoteObject`.
The last two productions `obj2R` and `addMethod` are for remote objects design.

The following invariant establishes that every remote object either extends `URO` or
implements an `OM`

\[
\phi_{inv} \overset{\text{def}}{=} \forall RO(x, y, z, w). (\exists URO(a, b). x = b \lor \exists OM(c). w = c )
\]

which intuitively imposes designers to avoid reusing remote objects for non-remote ones.

We now apply our algorithm which requires the computation of the weakest pre-
condition for every production in the system. We initially consider productions `extUnicast`
and \texttt{obj2RU}Using\texttt{URO}. By Definitions 19, and 20, the algorithm has to compute

\[ \mathcal{W}_{0,0}^{\tilde{h}}(p, \phi_{inv}) = w d_{0}^{p', \top} (\phi_{inv}) \land w p_{0,0}^{\tilde{h}} (\phi_{inv}) \]

Therefore, the case of universal quantification is used first:

\[ ( \bigwedge_{j=1,...,4} w d_{j}^{p_{j}^{\top}} (\phi'_{inv}) ) \land ( \bigwedge_{j=1,...,4} \forall \text{RO}(x, y, z, w). w p_{j}^{\tilde{h}} (\phi'_{inv}) ) \]

where \( \phi'_{inv} = \exists \text{URO}(a, b). x = b \lor \exists \text{OM}(c). w = c \). Let \( v_1 \) and \( v_2 \) be representative nodes outside the RHS of the productions above and let \( u_1, u_2, \) and \( u \) be as in such production in Figure 5.4 (that is \( u_1 \) and \( u_2 \) are interface nodes and \( u \) the internal node).

The assignments

\[
\begin{align*}
\mathcal{E}_1 &= \{ x \mapsto (\forall, \text{RO}, u), \ y \mapsto (\forall, \text{RO}, u_1), \ z \mapsto (\forall, \text{RO}, u_2), \ w \mapsto (\forall, \text{RO}, u_3) \} \\
\mathcal{E}_2 &= \{ x \mapsto (\forall, \text{RO}, u_1), \ y \mapsto (\forall, \text{RO}, u_2), \ z \mapsto (\forall, \text{RO}, u_3), \ w \mapsto (\forall, \text{RO}, u_4) \} \\
\mathcal{E}_3 &= \{ x \mapsto (\forall, \text{RO}, v_1), \ y \mapsto (\forall, \text{RO}, v_2), \ z \mapsto (\forall, \text{RO}, v_3), \ w \mapsto (\forall, \text{RO}, v_4) \} \\
\mathcal{E}_4 &= \{ x \mapsto (\forall, \text{RO}, u_1), \ y \mapsto (\forall, \text{RO}, v_1), \ z \mapsto (\forall, \text{RO}, v_2), \ w \mapsto (\forall, \text{RO}, v_3) \}
\end{align*}
\]

cover all the cases (all the non listed cases are captured by \( \mathcal{E}_3 \)) for the computation of \( \mathcal{W}_{0,\mathcal{E}}^{\tilde{h}}(p, \phi_{inv}) \).

The next step requires the use of the disjunction rule returning

\[ w d_{j}^{p_{j}^{\top}} (\phi'_{inv}) = w d_{j}^{p_{j}^{\top}} (\phi''_{inv}) \lor w d_{j}^{p_{j}^{\top}} (\phi'''_{inv}) \]
where $\phi''_{inv} = \exists \text{URO}(a, b). x = b$ and $\phi'''_{inv} = \exists \text{OM}(c). w = c$.

By applying now the existential quantification steps for $\phi''_{inv}$ and $\phi'''_{inv}$ we get

\[
wp_{E_j,0}^p(\phi''_{inv}) = wp_{E_j,0}^p(\phi''_{inv}) \lor wp_{E_j,0}^p(\phi'''_{inv})
\]

where $k = 5, 6, 7$ and $l = 8, 9$ and the assignments for $E_5, \ldots, E_9$ are:

\[
E_5 = \{ a \mapsto (\exists, \text{URO}, u_1), \ b \mapsto (\exists, \text{URO}, u) \}
\]
\[
E_6 = \{ a \mapsto (\exists, \text{URO}, v_1), \ b \mapsto (\exists, \text{URO}, u_1) \}
\]
\[
E_7 = \{ a \mapsto (\exists, \text{URO}, v_1), \ b \mapsto (\exists, \text{URO}, v_2) \}
\]
\[
E_8 = \{ c \mapsto (\exists, \text{OM}, u_1) \}
\]
\[
E_9 = \{ c \mapsto (\exists, \text{OM}, v_1) \}
\]

Finally, applying the node equality cases in the auxiliary map $eq_{p,p_1=p_2}(E)$ of Defi-
nition 20, we get

\[ \bigwedge_j w_d^{p,T}(\phi''_{\text{inv}}) = (\top \lor \bot \lor \bot) \land (\top \lor \top \lor \bot) \land \ldots = \top \] (5.7)

\[ \bigwedge_j w_d^{p,T}(\phi''_{\text{inv}}) = (\top \lor \bot \lor \top) \land (\top \lor \bot \lor \top) \land \ldots = \top \] (5.8)

\[ \bigwedge_j \forall RO(x,y,z,w).wp_{E_j,0}(\phi''_{\text{inv}}) = \phi \lor \phi' \] (5.9)

where, letting \( Q[\_] \) (resp. \( Q'[\_] \)) be the context \( \exists URO(a,b).[\_] \) (resp. \( \exists OM(c).[\_] \)),

\[ \phi = (Q[x = b] \lor Q[\bot] \lor Q[\bot]) \land (Q[x = b] \lor \ldots) \land \ldots = Q[x = b] \]

\[ \phi' = (Q'[w = c] \lor Q'[\bot] \lor Q'[w = c]) \land (Q'[w = c] \lor \ldots) \land \ldots = Q'[w = c] \]

Using the computations above

\[ \phi_{\text{inv}} = \mathcal{W}_{0,0}^h(p, \phi_{\text{inv}}) = \top \land \forall RO(x,y,z,w).(Q[x = b] \lor Q'[w = c]) \]

The weakest pre-condition for \( \text{obj2R} \) using \( OM \) and \( \text{addMethod} \) is \( \phi_{\text{inv}} \) itself. Indeed, the productions do not violate the invariant. For a production to violate \( \phi_{\text{inv}} \) it has to create a fresh edge of type \( RO \) that is not attached to any edge of either type \( URO \) or \( OM \).

By applying though the algorithm to production \( \text{obj2R} \) we get \( \mathcal{W}_{0,0}^h(\text{obj2R}, \phi_{\text{inv}}) = \bot \) as the production introduces a \( RO \) with no \( URO \) or \( OM \) attached to it. Cases 8 and 9 of the auxiliary mapping in Definition 18 forces the algorithm to return \( \bot \). We omit the workings as \( \mathcal{W}_{0,0}^h(p, \phi_{\text{inv}}) \) is computed in a similar fashion as in Example 25 and above.
for productions \texttt{extUnicast} and \texttt{obj2RUsingURO}.

By checking the weakest pre-condition against a graph before applying a production we guarantee that after the application the obtained graph will satisfy the architectural style specified by the invariant. This way we can use a more general design of a system like in this case and using the invariant, stop the system from evolving independently without removing or adding new productions.
In this chapter, we envisage architectural styles as formalised by a set of AcDR productions combined with a formula of our logic specifying a global invariant of the system as illustrated in Example 26 below.

**Example 26.** Consider the following run-time reconfiguration of a functioning server $S$, a non-functioning server $F$, and a client $C$

$$
\text{badServer()} \\
\text{where } S \text{ changes as illustrated to } F. \text{ By imposing an invariant requiring every client } C \text{ to be connected to a functioning server } S, \text{ the invalid configuration can be identified and recovered.}
$$

Based on what we defined so far in Chapter 4 and Chapter 5, we give a methodology in Section 6.1 for recovering a system to a valid state when run-time configurations
compromise it. In Section 6.2 we show how the methodology is applied to a simple flight booking system and finally in Section 6.3 we apply the methodology to our running example showing how the architectural style can be preserved when ships get detached from the network.

6.1 Methodology for recovering the architectural style

In Chapter 5, we introduced DbC for ADR which allowed us to specify pre- and post-conditions to production in an attempt to allow the generation of graphs to happen in a style preserving manner. Such an approach allows us to be more expressive when defining the style of the architecture making sure that by the correct application of AcDR productions we obtain a graph that is guaranteed to be style preserving. What happens though once run-time reconfigurations come to place? One cannot guarantee that every run-time reconfiguration of a system will not compromise its architectural style. What is required is a framework that can adapt once an unexpected configuration violates the system.

This section provides a methodology for recovering a system, compromised by run-time reconfigurations. The fact that we can record the evolution of our graphs (cf. Chapter 4) provides many advantages. The monitoring mechanism introduced can help us to reconstruct a graph matching the system’s state during execution and the new rewriting mechanism gives us the potential to identify which part of the graph has been re-written. Once the violated part of the graph is identified, one can use the weakest
pre-condition algorithm provided in Section 5.3 to identify a sequence of productions that can enforce the appropriate architectural style to the system.

More precisely, we are interested in computations that start from a system configuration, say \( s_0 \), that corresponds to an initial graph, say \( G_0 \), supposed to satisfy an invariant, say \( \phi_{inv} \). The system may evolve at run-time through a series of reconfigurations \( (r_i) \) that are reflected at the architectural level as schematically represented in the diagram (6.1) below (where \( G_i \vdash s_i \) stands for ‘\( s_i \) “matches” \( G_i \)’):

\[
\begin{align*}
G_0 & \rightarrow G_1 \rightarrow \cdots \rightarrow G_{k-1} \rightarrow G_k \rightarrow \cdots \\
T & \quad T \quad \cdots \quad T \quad T \quad \cdots \\
s_0 \xrightarrow{r_1} & \ xrightarrow{r_2} \cdots \xrightarrow{r_{k-1}} \ xrightarrow{r_k} \ xrightarrow{r_{k+1}} \cdots \\
\end{align*}
\]

We assume that most of the run-time reconfigurations produce graphs that do not violate \( \phi_{inv} \). Occasionally, the graph obtained by a run-time reconfiguration, say \( G_i \), may violate \( \phi_{inv} \). Our approach essentially computes how to rewrite graph \( G_i \) to a graph \( G_{i+1} \) satisfying \( \phi_{inv} \) and then reflect this into \( s_i \) by means of reconfigurations leading to a state \( s_{i+1} \) with architecture \( G_{i+1} \).

We propose a methodology that can select a number of productions that when applied to \( G_i \) induce the rewriting of the violating system into a state whose style satisfies \( \phi_{inv} \). Typically, \( \phi_{inv} \) states a global property of the system as opposed to the local properties captured by pre- and post-conditions of productions. We assume a monitoring mechanism that triggers our methodology whenever a reconfiguration yields an invalid system.
Our methodology consists of the steps 1 ÷ 5 below. 1

1. The architecture (say $G$) corresponding to the configuration of the current system is computed using the tracking tree $T$ and the corresponding environment $\mathcal{T}$ of the system.

2. Identify the sub-tree $T_R$ of $T (t' \bowtie T_R)$ that corresponds to the RHS term $t'$ of the reconfiguration applied.

3. Check whether $G$ satisfies $\phi_{inv}$ ($G \models \phi_{inv}$).
   (a) If $G \models \phi_{inv}$ then the style is not violated.
   (b) If $G \not\models \phi_{inv}$ then go to step 4

4. For each production $p$, compute the weakest pre-condition $\psi$ with respect to $\phi_{inv}$.
   (a) Select a production $p : L \rightarrow R$ and let $\sigma : L \rightarrow G$ be the morphism from $L$ to $G$ such that $G \setminus \sigma(E_L) \models \psi$ (if any); apply $p$ to $G$ to determine the reconfiguration needed for the system to reach a valid state.
   (b) If the designer considers the reconfigured system obtained in the previous stage to be not satisfactory or if there is no production $p$ such that $G \setminus \sigma(E_L) \models \psi$, then the designer may either repeat step 4a replacing $G$ with $G \setminus \sigma(E_L)$ and $\phi_{inv}$ with $\psi$, or try step 5.

5. Given $T_R$, computed in step 2, the designer can select a 2-tier subtree $T'_R$ of $T_R$ that contains a parent vertex and all its leaf children correspond to replaceable

---

1 Recall that the designer has to specify productions and the architectural invariant $\phi_{inv}$ so to establish the architectural style of interest (as done in Example 26).
edges through the environment $\mathcal{T}$. Using $T'_R$ in Definition 22 we parse the graph as defined below and repeat step 4.

**Definition 22** (Parsing). Given a graph $G$, a corresponding 2-tier sub-tree $T$ with root $\diamond$ and an environment $\mathcal{T}$. $p : L \xrightarrow{\iota} R$ is the production returned from $\mathcal{T}^{(2)}(\diamond)$. Given an instance $L'$ of $L$ through the isomorphism $\iota : L \to L'$ and let $\sigma' : R \to G$ a graph $G' = G \setminus (E_G \cup \sigma'(R^e)) \cup L''$ ($R^e$ refer to the internal nodes of $R$) is the graph obtained by parsing $G$ with $p$ under the morphism $\sigma'$ iff $L'' = L'[\iota(l) \mapsto \sigma'(i(l)) \mid l \in \text{Im}(\iota)]$ and there exist no edge in $G$ that is non-replaceable.

Identifying the architecture of a system is non-deterministic in the original specification of ADR (cf. [18]). In AcDR we can identify and retrieve the architecture of the system with all the information of how it has been adapted so far using our monitoring mechanism. This mechanism is used in step 1 for retrieving graph $G$ and in step 2 for identifying the subtree of the tracking tree that has been reconfigured. In step 3, we assume that an underlying monitoring mechanism uses the $\models$ relation of our logic to determine if the graph $G$ identified in step 1 violates the invariant. In such case, step 4 uses the weakest pre-condition algorithm (c.f. Section 5.3) on each production to compute their weakest pre-conditions (this step does not need to be re-iterated at each reconfiguration but instead all the weakest pre-conditions can be computed offline and reused accordingly). In step 4a, if the graph representing the violated system satisfies one of the computed weakest pre-conditions then the corresponding production is a candidate to re-establish the architectural style and trigger the appropriate reconfigurations on the invalid system. In step 4b the designer has to decide whether to stop or continue the process. In the latter case, the idea is to repeat steps 4 and 4a replacing $G$. 

6.1. Methodology for recovering the architectural style
with $G \setminus \sigma(E_L)$ and $\phi_{inv}$ with $\psi$ so to compute the weakest pre-condition of the weakest pre-condition computed in the previous iteration. This, allows us to exploit every possible sequence of productions that can be applied in order to enforce the architectural style. Note that the morphism that invalidates $G \models \phi_{inv}$ indicates which part of the system has to be rewritten, while the production $p$ suggests plausible reconfigurations. If the above steps do not prove to be satisfactory then in step 5 the designer uses the tree computed in step 2 and selects which branch of the tree can be abstracted using the parsing mechanism in Definition 22. After parsing the graph we repeat steps 4 ÷ 5 until either, a possible sequence of productions is identified that can potentially fix the architectural style, or, the designer decides to stop the process.

In Sections 6.2 and 6.3 we apply the methodology above to a simple flight booking system and to a more complex scenario related to our running ship example in order to give a better understanding of how our methodology works in a dynamic environment.

### 6.2 Flight Booking System

We consider a scenario where a flight-search engine allows users to book flights. We use the type graph in Figure 6.2 where, to simplify the type graph, we use $e \in E \setminus \{c\}$ (instead of drawing an edge for each edge of $\Gamma$ with arity 2). In addition, there is only one type of node $\bullet$ and the types of edges are $C$ (for clients), $BF$ (for the booking
flights), FF (for the broker service finding flights), Fls (for the different flights available), Fl (for the flight to be booked), and P and PF (for completed or failed payment services, respectively). Consider the following productions:

where findFlights establishes a broker service FF, bookFlight and retryBooking yields a flight (Fl) connected to a payment service (P), browseFlights generates as many flights as necessary, deleteFlight and noFlights respectively remove and stop adding flights to the design, and finally removeFailedPaymentFlag removes a failed payment PF from the design.

Services can either be composed with other services using findFlights and bookFlight like for instance when one chooses a specific flight and the system needs to “invoke” another service (payment service) to complete the request, or branch using the production browseFlights to represent the different flights a customer can choose from.
CHAPTER 6. ENFORCING ARCHITECTURAL STYLES

Graph $G_a$, in Figure 6.2, shows the architectural style of a system where a client books a flight and successfully pays for it. Initially, the client searches for a flight by invoking the $\text{findFlight}$ service which, in turn, invokes different airlines about their flights. Once a flight is selected a payment service is used to complete the transaction.

In a realistic environment though, a payment may fail to go through. This will force graph $G_a$ to be rewritten to $G_b$ (Figure 6.2) where $P$ now is $PF$. We show how to apply our methodology in this scenario.

6.2.1 Enforcing the Style with a Simple Invariant

The style we consider consists of the productions presented in Section 6.2 and the invariant

$$\phi_{F1} = \exists F1(x_1, x'_1). \exists P(x'_2, x_2) \cdot x_1 = x_2$$

specifying that some flight $F1$ has to be connected to a successful payment $P$.

Following the methodology presented in Section 6.1, we need to check if graph $G_b$ given in Figure 6.2 satisfies the invariant $\phi_{F1}$ and find that $G_b \not\models \phi_{F1}$. In fact, there is no edge of type $P$ in $G_b$ so we invoke $W^{\bar{h}}_{h,0}(p, \phi_{F1})$ on every production $p$ where $h$ is $\emptyset$ (since $\phi_{F1}$ is a closed formula) and $\bar{h}$ maps the interface nodes of $p$. We have $W^{\bar{h}}_{\emptyset,0}(p, \phi_{F1}) = \phi_{F1}$ for all $p \neq \text{bookFlight}$ whereas, for $p = \text{bookFlight}$, $W^{\bar{h}}_{\emptyset,0}(p, \phi_{F1}) = \top$.

We show that $W^{\bar{h}}_{\emptyset,0}(p, \phi_{F1})$ yields $\phi_{F1}$ for any $p \neq \text{bookFlight}$ since such productions do not have edges of type $F1$ or $P$ in their RHS. We have to compute $wd^{\mu'}_{0}(\phi_{F1}) \land wp^{\nu',\bar{h}}_{0,0}(\phi_{F1})$ by first applying the case for existential quantification (cf. Definition 19,
and Definition 20, page 80):

\[
\left( \bigvee_{j=1,\ldots,5} \text{wd}_{E_j}(\phi'_{F_1}) \right) \land \left( \bigvee_{j=1,\ldots,5} \exists \text{Fl}(x_1, x'_1). \text{wp}_{E_j, \emptyset}(\phi'_{F_1}) \lor \text{wd}_{E_j}(\phi'_{F_1}) \right)
\]

where \( \phi'_{F_1} = \exists \text{P}(x'_2, x_2). \ x_1 = x_2 \). Let \( v_1 \) and \( v_2 \) be representative nodes outside the RHS of the productions above, \( u_1 \) and \( u_2 \) be interface nodes of the productions. The assignments

\[
\mathcal{E}_1 = \{ x_1 \mapsto (\exists, \text{Fl}, u_1), \ x'_1 \mapsto (\exists, \text{Fl}, v_1) \}
\]
\[
\mathcal{E}_2 = \{ x_1 \mapsto (\exists, \text{Fl}, u_2), \ x'_1 \mapsto (\exists, \text{Fl}, v_1) \}
\]
\[
\mathcal{E}_3 = \{ x_1 \mapsto (\exists, \text{Fl}, u_1), \ x'_1 \mapsto (\exists, \text{Fl}, u_1) \}
\]
\[
\mathcal{E}_4 = \{ x_1 \mapsto (\exists, \text{Fl}, v_1), \ x'_1 \mapsto (\exists, \text{Fl}, u_2) \}
\]
\[
\mathcal{E}_5 = \{ x_1 \mapsto (\exists, \text{Fl}, v_1), \ x'_1 \mapsto (\exists, \text{Fl}, v_2) \}
\]

are the only ones to consider for the first quantification. Instead, for the other existential quantification \( \exists \text{P}(x'_2, x_2) \) yields

\[
\left( \bigvee_{j,k=7,\ldots,11} \text{wd}_{E_j \cup E_k}(x_1 = x_2) \right) \land \left( \bigvee_{j,k=7,\ldots,11} \text{Q}[\text{wp}_{E_j \cup E_k}(x_1 = x_2), \emptyset] \lor \text{wd}_{E_j \cup E_k}(x_1 = x_2) \right)
\]
where \( Q[\cdot] \) is the context \( \exists F_1(x_1, x'_1).\exists P(x'_2, x_2).[\cdot] \) and the assignments \( \mathcal{E}_7, \ldots, \mathcal{E}_{11} \) are:

\[
\begin{align*}
\mathcal{E}_7 & = \{ \ x_2 \mapsto (\exists, P, u_1), \ x'_2 \mapsto (\exists, P, v_1) \ \} \\
\mathcal{E}_8 & = \{ \ x_2 \mapsto (\exists, P, u_2), \ x'_2 \mapsto (\exists, P, v_1) \ \} \\
\mathcal{E}_9 & = \{ \ x_2 \mapsto (\exists, P, v_1), \ x'_2 \mapsto (\exists, P, u_1) \ \} \\
\mathcal{E}_{10} & = \{ \ x_2 \mapsto (\exists, P, v_1), \ x'_2 \mapsto (\exists, P, u_2) \ \} \\
\mathcal{E}_{11} & = \{ \ x_2 \mapsto (\exists, P, v_1), \ x'_2 \mapsto (\exists, P, v_2) \ \}
\end{align*}
\]

Finally, applying the case for node equality in the auxiliary map \( eq_p^{\psi_1, \psi_2, \psi_3}(\mathcal{E}) \) of Definition 17, page 74 for \( 1 \leq j \leq 5 \) and \( 7 \leq k \leq 11 \), we get

\[
\begin{align*}
wd_{\mathcal{E}_j \cup \mathcal{E}_k}^p (x_1 = x_2) & = \top \quad (6.2) \\
Q[wp_{\mathcal{E}_j \cup \mathcal{E}_k}^p (x_1 = x_2), \emptyset] & = \phi_{F_1} \vee \bot = \phi_{F_1} \quad (6.3)
\end{align*}
\]

therefore, the conjunction of (6.2) and (6.3) respectively yield \( \top \) and \( \phi_{F_1} \) hence, \( \mathcal{W}_{\emptyset, 0}^h (p, \phi_{F_1}) \) returns \( \phi_{F_1} \).

We now consider \( p = \text{bookFlight} \) and show that \( \mathcal{W}_{\emptyset, 0}^h (p, \phi_{F_1}) = \top \). As in the previous case, we consider the quantifications for which we have to consider the extra mappings due to \( F_1 \) and \( P \):

\[
\begin{align*}
\mathcal{E}_6 & = \{ x_1 \mapsto (\exists, F_1, u), x'_1 \mapsto (\exists, F_1, u_2) \} \\
\mathcal{E}_{12} & = \{ x'_2 \mapsto (\exists, P, u_1), x_2 \mapsto (\exists, P, u) \}
\end{align*}
\]
where $u_1$ and $u_2$ are the production’s interface nodes as before and $u$ is its unique internal node. By the quantification cases we have

$$
\left( \bigvee_{j,k} wd^p_{E_j \cup E_k}(x_1 = x_2) \right) \land \left( \bigvee_{j,k} Q[wp^p_{E_j \cup E_k, \emptyset}(x_1 = x_2)] \lor wd^p_{E_j \cup E_k}(x_1 = x_2) \right)
$$

for $1 \leq j \leq 6$ and $7 \leq k \leq 12$.

Finally, applying the case for node equality in the auxiliary map $eq^{\psi_1, \psi_2, \psi_3}(E)$ of Definition 17 for $1 \leq j \leq 6$ and $7 \leq k \leq 12$, we get

$$
wd^p_{E_j \cup E_k}(x_1 = x_2) = \top \quad (6.4)
$$

$$
Q[wp^p_{E_j \cup E_k}(x_1 = x_2), \emptyset] = Q[\top] \lor \top = \top \quad (6.5)
$$

Note that the weakest pre-conditions is the conjunction of (6.4) and (6.5), that is

$$(wd^p_0(\phi_{F1}) \land wp^p_{0, \emptyset}(\phi_{F1})) = \top$$

The next step requires that we check whether $G_b$ satisfies any of the weakest pre-conditions computed.

$$
G_b \not\models \exists F1(x_1, x_1'). \exists P(x_2', x_2). x_1 = x_2 \quad \text{but instead,} \quad G_b \models \top
$$

and therefore we know that by applying the production bookFlight we get a graph $G'_b$ that satisfies the invariant $\phi_{F1}$.
6.2.2 A more Complicated Invariant

Let us now consider a more strict invariant

\[ \phi_{PF} = \forall F1(x_1, x'_1). \exists P(x'_2, x_2). x_1 = x_2 \land \forall PF(x_3, x'_3). \perp \]

that specifies that every flight F1 has to be connected to a valid payment P and also there shouldn’t exist any edge of type PF in the graph.

Following the methodology in Section 6.1, we need to check if graph \( G_b \) given in Figure 6.2 satisfies the invariant \( \phi_{PF} \) to find that \( G_b \not\models \phi_{PF} \). In fact, although there are edges of type F1, there is no edge of type P in \( G_b \) and there is an edge of type PF in the graph. Therefore, we invoke \( W_{h,0}^\bar{h}(p, \phi_{PF}) \) on every production \( p \) where \( h = \emptyset \) (since \( \phi_{PF} \) is a closed formula) and \( \bar{h} \) maps the interface nodes of \( p \). Unlike in Section 6.2.1, \( W_{\emptyset,0}^\bar{h}(p, \phi_{PF}) = \phi_{PF} \) for each production \( p \). We omit the workings for \( W_{\emptyset,0}^\bar{h}(p, \phi_{PF}) \) since it is computed in a similar fashion as in Example 25 and Section 6.2.1.

Recall that \( \sigma(E_L) \) refers to the edge in \( G_b \) that corresponds to the LHS of the production \( p \) we are applying. The next step requires that we check whether the graph \( G_b \setminus \sigma(E_L) \) satisfies any of the weakest pre-conditions computed. Note that, even though \( G_b \not\models \phi_{PF} \) we are now checking whether \( G_b \setminus \sigma(E_L) \models \phi_{PF} \). \( G_b \setminus \sigma(E_L) \) is different depending on the production we are applying as well as the edge we are applying the production to.

For this specific example it is possible to fix the architectural style of \( G_b \) by applying initially the production \texttt{retryBooking} on the edge \( f_1 : F1 \) and then apply \texttt{paymentFailure}.
on the edge $pf : \text{PF}$. Using the methodology we know that

$$W_{0,0}^+(p, W_{0,0}^+(p, \phi_{\text{PF}})) = \phi_{\text{PF}}$$

Therefore we iterate step 4 on $G_b \setminus \sigma(E_L)$ where $E_L$ refers to the edge we are applying the production to. In this case applying $\text{retryBooking}$ to $f_1 : \text{F1}$ returns $G'_b$ and latter applying $\text{paymentFailure}$ to $pf : \text{PF}$ returns $G''_b$. Note that we had to repeat step 4 because $G'_b \not\models \phi_{\text{PF}}$ and we stop the iteration once we reach the point where $G''_b \models \phi_{\text{PF}}$.

### 6.3 Adaptive Network Connectivity on Ships

We consider a scenario given in Section 3.1, where ships can detach from their access point, or other connected ships they were connected to. We use the type graph in Figure 3.3, page 32 where there is only one type of node $\bullet$ and the various types of edges are describe in Figure 3.2, page 29. Hereafter, we will refer to ships being online or offline when they are connected to a broadcasting access point. A ship is also online when its second tentacle is attached to the first tentacle of another online ship. Consider
the following productions:

- `init`
- `detachedFleetON`
- `fleetON2Ship`
- `clusterChain`
- `detachedFleetOFF`
- `fleetON2OFF`
- `shipChain`
- `shipOFF2ON`
- `shipON2OFF`
- `cluster`
- `manyShips`
- `fleetOFF2Ship`
where \texttt{init} refers to the main initial graph of our system, \texttt{detachedFleetON} and \texttt{detachedFleetOFF} generates ships that are detached from the "network", \texttt{cluster} yields an access point and a number of ships attached to it, \texttt{clusterChain} generates a series of clusters, \texttt{shipChain} yields an online ship and a number of ships attached to it, \texttt{manyShips} is used to generate more ships attached to the same port and \texttt{fleetON2Ship}, \texttt{fleetON2OFF}, \texttt{shipOFF2ON}, \texttt{shipON2OFF} and \texttt{fleetOFF2Ship} are productions used for changing the state of ships from offline to online and vice-versa. The production \texttt{noFleet} below takes an edge of type \texttt{FleetON} and removes it from the graph.

\texttt{Reconfiguration detach} presented below is invoked when a ship disconnects from its access point.

\[
\text{detach} : \quad \texttt{init(clusterChain(x, cluster(y,z)),w)} \\
\quad \rightarrow \quad \texttt{init(clusterChain(x, cluster(y,noFleet())),detachedFleetON(z,w))}
\]

This is a very common when dealing with such dynamic systems since ships travel all the time while access points are fixed in different locations.

Graph \( G_1 \) in Figure 6.3 shows a sub-graph of the system where ship \( s_1 \) is online through its connection with the access point \( a_1 \) and ships \( s_2 \) and \( s_3 \) are also online through their connection to \( s_1 \). The graph \( G_2 \) represented in both Figure 6.3, and
Figure 6.3: Ships disconnecting scenario

Figure 6.4 represents what happens to the graph once $G_1$ is reconfigured using the detach reconfiguration. When one ship is detached from an access point all the other ships attached to it go offline as well. In Graph $G_2$ though, despite the fact that all ships are detached from the network, they are still of type $\text{ShipON}$ indicating that they are online. The next section shows how to apply our methodology by using the parsing mechanism (cf. Definition 22, page 112) to change the type of these edges and make our graph consistent.
6.3.1 Enforcing the Style Using the Tracking Tree

The style we consider consists of the productions given in Section 6.3 and the invariant \( \phi_{\text{ON}} \) below.

\[
\begin{align*}
\phi_{\text{ON}} &= \phi_{\text{ON}}^1 \land \phi_{\text{ON}}^2 \\
\phi_{\text{ON}}^1 &= \forall \text{ShipON}(x,y).((\exists \text{AP}(a,b,c).y = b) \lor (\exists \text{ShipON}(d,e).(y = d \land x \neq e))) \\
\phi_{\text{ON}}^2 &= \forall \text{FleetON}(x,y).\exists \text{ShipON}(a,b).y = a
\end{align*}
\]
\( \phi^1_{\text{ON}} \) specifies that each ship has to be connected to either an access point or another online ship in order to be considered connected. More precisely, every edge of type \texttt{ShipON} should be connected to either an edge of type \texttt{AP} or another edge of type \texttt{ShipON}. \( \phi^2_{\text{ON}} \) states that for every edge of type \texttt{FleetON} there should exist an edge of type \texttt{ShipON} attached to it.

We follow the methodology presented in Section 6.1, and check whether the graph \( G_2 \) given in Figure 6.3, and Figure 6.4 satisfies the invariant \( \phi_{\text{ON}} \) to find that \( G_2 \not\models \phi_{\text{ON}} \). In fact, after the reconfiguration

\[
G_1 \xrightarrow{\text{detach}} G_2
\]

the edge \( s_1 \) of type \texttt{ShipON} in \( G_2 \) is no longer attached to either an \texttt{AP} or another \texttt{ShipON} on its second tentacle. Therefore, we invoke the weakest pre-condition \( \mathcal{W}^h_{\bar{h},0}(p, \phi_{\text{ON}}) \) on every production \( p \) where \( h = \emptyset \) (since \( \phi_{\text{ON}} \) is a closed formula) and \( \bar{h} \) maps the interface nodes of \( p \). In this scenario \( \mathcal{W}^{\bar{h}}_{\emptyset,0}(p, \phi_{\text{ON}}) = \phi_{\text{ON}} \) for all \( p \) except \texttt{shipOFF2ON}, \texttt{manyShips}, \texttt{shipChain} and \texttt{fleetON2Ship}. In those cases the algorithm would return \( \bot \) as they all satisfy Conditions 9 and 10 from Definition 18, page 77 and hence, will not considered in the rest of the example. We will use \( p \) to denote all the other productions presented in Section 6.3. We skip the computation of \( \mathcal{W}^{\bar{h}}_{\emptyset,0}(p, \phi_{\text{ON}}) \) as it is done in a similar fashion as in Section 6.2.1 and focus more on the application of the methodology.

After computing the weakest pre-condition for every production we check whether the graph \( G_2 \backslash \sigma(E_L) \) satisfies any of the weakest pre-conditions computed. Recall that

\[
\sigma : L \to G_2
\]

6.3. Adaptive Network Connectivity on Ships 125
refers to the morphism from $L$ (LHS of $p$) to $G_2$ and therefore, $\sigma(E_L)$ is the edge in $G_2$ that corresponds to the LHS of the production we are applying.

The tracking forest $T_2$ and its corresponding environment $T_2$ corresponding to $G_2$ are illustrated in Figure 6.5. It shows which part of the graph has been reconfigured and therefore we only have to consider productions that can be applied to our graph. More precisely we are only interested in productions whose type exist in $G_2$. By following the methodology steps 1 ÷ 4, we can fix the architectural style of $G_2$ by applying production $\text{shipON2OFF}$ several times for edges $s_1$, $s_2$ and $s_3$. This way we obtain graph $G'_2$ (see Figure 6.4) that satisfies $\phi_{\text{ON}}$ since all the ships are now in the offline state. The way $G'_2$ is computed is similar to the way used in Section 6.2.2 which required the computations of the weakest pre-condition $\mathcal{W}_{\emptyset,0}^h(p, \mathcal{W}_{\emptyset,0}^h(p, \phi_{\text{Fl}}))$ to be done in several repetitions. This can become even more complicated when dealing with more complex reconfigurations and when there exist more productions of type $\text{ShipON}$ affected by the
reconfiguration.

The tracking forest $T_2$ and its corresponding environment $T_2$ in Figure 6.5 correspond to $G_2$ after the reconfiguration has been applied. In the rest of the section we will use of step 5 in the methodology and try to enforce the architectural style of $G_2$ in a more efficient way.

Since $AP$ in a non-replaceable edge in our graph we can turn the focus to the rest of the graph. Using $T_2$ one can backtrack the forest starting from the leaves and parse (cf. Definition 22, page 112) edges $s_1$, $s_2$ and $s_3$ until the vertex of the subtree with label $s$. The parsing (bottom-up application) happens in the 4 steps below where

1. Bottom up application of $fleetON2Ship$ on edge $s_2$ to obtain edge $ms_2$;
2. Bottom up application of $fleetON2Ship$ on edge $s_3$ to obtain edge $ms_3$;
3. Bottom up application of $manyShips$ on edges $ms_2$ and $ms_3$ to obtain $ms_1$;
4. Bottom up application of $shipChain$ on edges $s_1$ and $ms_1$ to obtain edge $s$ of type $FleetON$.

Graph $G_2^p$ shown in Figure 6.4 represents the graph obtained after steps 1 ÷ 4 are invoked. $G_2^p$ does not satisfy $\phi_{ON}$ though as there exists an edge of type $FleetON (s)$ that is not attached to either an access point ($AP$) nor to another online ship ($ShipON$). Using $G_2^p$ we can apply the methodology step 4 and this time consider only productions of type $FleetON$ since it is the only replaceable edge of our graph. As we already observed earlier productions $shipChain$, $manyShips$ and $fleetON2Ship$ all return $\perp$. Therefore, the only production of type $FleetON$ left is production $fleetON2OFF$. By replacing $G_2^p$ with $G_2^p \setminus \sigma'(E_L)$ where $E_L$ refers to the edge we are applying the production to.

6.3. Adaptive Network Connectivity on Ships
(in this case edge $s$) we observe that $G^p_h \setminus \sigma'(E_L) \models W^h_{0,0}(p, \phi_{ON})$. Therefore applying \texttt{fleetON2OFF} to $G^p_2$ returns a graph $G_3$ (shown in Figure 6.4) that is guaranteed to satisfy $\phi_{ON}$.

This time we did not have to repeat step 4 because using the parsing mechanism we managed to bring graph $G_2$ to a state ($G^p_2$) that could be fixed with a single iteration of the methodology. It is important to remind the reader that parsing would not be possible if either $s_1$, $s_2$ or $s_3$ were non-replaceable edges. One sets an edge to be non-replaceable if its value is essential for the correct functioning of the system (like in the case of AP) and the fact that ships can migrate from one access point to another makes them a perfect candidate to represent how replaceable edges can be handled when the system is violated.
In this chapter, we give a summary of our contributions in Section 7.1 and then in Section 7.3 we list some future work.

7.1 Contributions

AcDR defines a framework that allows us to exploit the “hierarchical nature” of ADR graphs. In particular, we used the functional reading of ADR productions as well as its reconfigurations that preserve the identity of components throughout the rewriting. Our framework permits to tackle certain architectural aspects of the design and allows the designer to identify and address problems at the architectural level.

A variant of ADR. In Chapter 4, we focused on the development of the technical presentation of our framework. We proposed a monitoring mechanism through which
CHAPTER 7. CONCLUSION

the evolution of a computation is recorded and maintained in a tree-like structure reflecting the hierarchical nature of ADR graphs. We then exploit this mechanism to formally define more efficient parsing algorithms as well as more efficient ways of applying reconfigurations. Interestingly, this approach brings forth the definition of a new rewriting mechanism for ADR reconfiguration rules. More precisely, instead of parsing an ADR graph searching for a sub-graph matching the RHS of a reconfiguration rule, we propose a rewriting mechanism that visits the trees describing the graph evolution to find the match. We argue that this is more efficient than parsing the graph at the negligible cost of recording the evolution of the system through the monitoring. One could argue that the run-time monitoring of the system could be inefficient. In this respect, we note that a form of monitoring is necessary when dealing with self-configuring or self-healing systems; quoting [50] “autonomic system might continually monitor its own use”. Since autonomic systems are the reference systems of our work, and, as a matter of fact, a form of monitoring is indispensable in order to identify run-time violation of the style, we argue that our approach simply adds to the necessary monitoring activities the cost of tracking the evolution of systems.

AcDR: Introducing Architectural Contract-based Design Rewriting. In Chapter 5 we introduced a methodology inspired by Design-by-Contract (DbC) [65] to guarantee properties of architectural designs. Technically this is achieved by

1. equipping ADR with a logic tailored to express such properties;

2. using the logic to decorate design productions with pre- and post-conditions;

3. devising an algorithm to compute weakest pre-conditions for ADR productions.
Albeit very simple, our logic can express rather interesting properties (cf. Example 18, page 62). It allows us to improve the expressiveness of ADR and to specify interesting properties exploiting the 'hierarchical nature' of ADR graphs. The work presented in Chapter 5 is a first step in the exploration of the use of DbC in architectural style reconfigurations.

**Enforcing Architectural Styles.** In Chapter 6 we show how we make use of AcDR to provide strategies for recovering system violations of the architectural style when reconfigurations (possibly unexpected) compromise it. The methodology we present fixes architecturally our graphs, provided that we have the appropriate productions to do this. Using the monitoring mechanism one can obtain the architectural representation of the system and in conjunction with the new rewriting mechanism identify which part of the graph has been compromised. We then recursively compute the weakest pre-condition computed by the algorithm until a possible sequence of reconfigurations is identified to re-establish the architectural style of the system. More precisely, one can compute a sequence of productions by iterating the methodology in Chapter 6 on the weakest pre-condition obtained at every “round” (starting from the invariant) until either “false” or a valid style is reached. Furthermore, if no sequence of productions is identified to enforce the architectural style at a specific level of abstraction we can follow the tracking tree (cf. Chapter 4) in a bottom-up fashion and bring the graph to a more abstract level before reapplying the weakest pre-condition methodology. One could argue that by abstracting the graph of the system we lose information about certain components of the system. The introduction of replaceable and non-replaceable edges in Chapter 4 can help us preserve the components that should not be abstracted.
away. By setting an edge as non-replaceable we prevent the parsing mechanism from abstracting its information.

7.2 Evaluation of Contribution

**Thesis Roadmap**  The properties highlighted in Section 2.2, page 15 have been used as roadmap throughout the thesis and this section capitalises on Section 7.1 to briefly highlight which goals have been addressed.

Chapter 4, is a step towards more understandable and scalable systems, that offer through both the more simplified rewriting mechanism and the proposed monitoring mechanism a better link between design time evolution (in terms of the way various components interact) and run-time configuration. Due to the hierarchical style and persistence of the monitoring mechanism, from design to run-time, one can view the system at various levels of abstraction, which is of paramount importance when dealing with large, scalable systems.

Chapters 5 and 6 mainly focus on controlling system evolution and adaptability (see Section 2.2, page 15). As also discussed in Section 2.2, being able to model dynamic changes of a system does not guarantee style preservation. One has to able to enforce the architectural style during the evolution even when unexpected misbehaviours occur. By equipping ADR with DbC (Chapter 5), we automatically provide a mechanism to the ADL that not only enhances the architectural style specifications but also provides control of how and when changes are allowed. One of the goals of the thesis was to present an ADL that is able to provide strategies for recovering the
architectural style of a system when unexpected and unwanted changes occur. By taking advantage of the weakest pre-condition algorithm, we introduced a methodology (see Chapter 6) that as explained earlier allows our system to adapt to such changes.

**Quality of Methodology** The methodology in Chapter 6 is employed for recovering a system once its architectural style is compromised. Our methodology partially relies on the designer to choose which of the proposed solutions should be employed to reinforce the style. Once such solutions are proposed, the designer is certain that applying one will fix the architectural style. This is due to the fact that these solutions were computed using the design productions that define the architectural style. The choice of the solution is therefore immaterial if the criteria lies on fixing the architectural style but as discussed also in Section 7.3 a possible future direction would be to automate the selection and application of such solutions by setting criteria in terms of components that should be preserved or maximum number of productions that can be used to enforce the architectural style.

### 7.3 Future Work

**Logic Extensions.** We expect our research to lead to extensions of the logic. We would like to make use of the fact that design productions can be applied in a bottom-up fashion and link this characteristic to our logic. So far our logic quantifies over types of edges and predicates on (in)equality of nodes of a given graph. We believe that including the diamond modality to our logic and checking whether some $\varphi$ is satisfied at a more abstract level will enhance the expressiveness of our logic and open new routes
in terms of style enforcement in AcDR.

\[ G \models \diamond \varphi \iff \text{exists a } G' \text{ and productions } \tilde{\pi} \text{ s.t. } G' \models \varphi \text{ for some } G' \xrightarrow{\tilde{\pi}} G \] (7.1)

In (7.1), given a graph \( G \) and a series of productions \( \tilde{\pi} \), \( G \models \diamond \varphi \) if the bottom-up application of \( \tilde{\pi} \) on \( G \) returns a graph \( G' \) such that \( G' \models \varphi \).

**Link** We contemplate a new research direction to devise autonomic systems where component managers use architectural elements in the reconfigurations they distill. From this point of view, ADR is particularly suitable due to the fact that it can represent not only architectural level aspects of systems, but it can also be used to represent operational semantics by e.g. encodings of process calculi or modelling languages [11, 15]. This would immediately establish a connection between DbC approaches of abstractions levels close to the implementation of systems (for instance, see [9, 55]) with the DbC approach for ADR suggested in this thesis.

**Expressiveness of reconfigurations** We start by commenting about the linearity conditions imposed on the reconfigurations rules (cf. Equation (3.2), page 38 and Definition 11, page 52). The linearity of the LHS of a reconfiguration rule can be relaxed at the cost of making the semantic of the matching more complex since multiple occurrences of the same variable would account for checking the existence of an isomorphism among different sub-graphs. Instead, the linearity condition on the RHS of a production can be relaxed by simply using the counterpart semantic mechanism described in [36] to keep track of one copy of the variable. Finally, we note that in [18], reconfigurations
rules of the form \( r(x) : p(y) \rightarrow q(x, y) \) are considered, where \( x \) act as a parameter of the rule that can be used in its LHS or RHS. Such rules can be easily added to our framework using Definition 11 by mapping \( \eta(x) \) to the fresh input graph.

As also discussed in Chapter 1 openness magnifies the complexity of software. The fact that open systems are subject to run-time reconfigurations that can lead to erroneous states opens many interesting ideas and questions.

**Software Architecture Design** When designing dynamic software, designers specify the normal (expected) behaviour of a system and then try to enumerate the (relatively few) erroneous states (that are most likely to happen and for which plausible reconfigurations can be designed). It is desirable to have mechanisms to support designers in this phase. For instance, having executable models or abstract representations of the computations may provide a rich framework into which applications can be probed under different conditions.

**Enforcing Architectural Styles.** Once the architectural style of the system is compromised the methodology in Chapter 6 is employed for recovering the system. Our methodology partially relies on the designer to choose which of the proposed solutions should be employed to re-enforce the style. When different sequences of productions are found, one could devise criteria to order them, or else to try to find criteria for good or best strategies. Generalising our idea for computing ‘strategies’ based on many productions to recover failures could be a very interesting future direction. More importantly, one would benefit from a framework capable to suggest possible ways to bring executions back to an acceptable state (which is of course dependent on the application).
Completeness of Methodology  In addition to the previous paragraph our proposed methodology relies on the design productions that define the architectural style and tries to compute patterns and sequences of productions that when applied to the system will bring it to an acceptable state. This is not always the case and a possible solution does not always exist. Our approach is non-deterministic which means that one could end up infinitely applying the methodology trying to identify patterns when no solution might exist.

In a practical environment one would have to set a limit on the number of production sequences used in the attempt to find a solution. In addition, taking inspiration from SAP’s Process Integration monitoring mechanism, which stores versions of their monitoring data one could store the various versions of our tracking tree. This way if no solution is found then one could devise ways of bringing the system back to a previous state.

Tool support and visual patterns.  In [16] a Maude implementation of ADR has been provided. A very interesting extension to our work would be to incorporate the DbC approach to the current implementation and allow users to specify contracts for ADR productions. This could lead to various interesting uses like for instance a simulator where one can give the specification of a system to the Maude tool and then apply a number of reconfigurations that may compromise the architectural style and check how the methodology in Chapter 6 reacts to these violations in an automated fashion.

In addition, we started developing a visual editor in Microsoft C# following ADRs original specifications [18]. The editor allows the specification of a number of ADRs ingredients like for instance the type graph and the design productions in a graphical
format. It offers a user friendly graphical interface in an attempt to give the designer and developer an environment that is easy to use and understand. The textual representation of our DbC approach contradicts the graphical environment of our editor. Such a visual tools open routes to new practises like for instance the diagrammatic constraints presented in [33]. Providing visual notations for specifying our pre- and post-conditions and also specifying recurring patterns for contracts and productions could provide automatic and efficient ways of modelling dynamic systems.


