ANALYSIS OF COGNITIVE MODELS IN CONSTRAINT HANDLING RULES

DANIEL GALL
aus Göppingen
Daniel Gall

*Analysis of Cognitive Models in Constraint Handling Rules*

**AMTIERENDER DEKAN:**
Prof. Dr.-Ing. Maurits Ortmanns

**GUTACHTER:**
Prof. Dr. Dr. Thom Frühwirth, Ulm University
Prof. Dr. Slim Abdennadher, German University in Cairo
Prof. Dr. Maurizio Gabbrielli, University of Bologna

**TAG DER PROMOTION:**
20. Januar 2021
Computational cognitive modeling explores cognition by building detailed computational models for cognitive processes, mechanisms and representations. Currently, computational cognitive modeling architectures as well as the implementations of cognitive models are typically ad-hoc constructs. There are many variants of architectures to support modeling for a certain domain. The architectures lack a formalization from the computer science point of view. This impedes analysis of the underlying languages and the programmed models.

In this work, we present how cognitive models in the popular cognitive architecture Adaptive Control of Thought – Rational (ACT-R) can be formalized and analyzed with the help of formal methods that come from the field of logic programming. In particular, analysis methods of the declarative programming language Constraint Handling Rules (CHR) are used and extended, such that ACT-R models can be analyzed.

Our work contains the following contributions:

- An abstract operational semantics of ACT-R has been formalized that allows for specifying different implementations by creating instances of the abstract formulation. For instance, the abstract semantics generalizes the conflict resolution mechanism, i.e. the rule choice of the procedural system of ACT-R, by introducing non-determinism on the rule choices.

- To allow for formal analysis, a sound and complete translation of ACT-R models to CHR has been developed. Due to the strong relation of CHR to formal logic, the translation is suitable for analysis.

- The analysis methods of CHR have to be extended to analyze ACT-R models. By introducing user-defined equivalence relations on states, the confluence condition can be relaxed such that translated ACT-R states can be considered. We have proposed a criterion for confluence modulo equivalence and operational equivalence modulo equivalence for CHR that can also be used in the context of ACT-R models.

- A necessary and sufficient confluence criterion for ACT-R models has been formulated that is based on our translation to CHR and the extended analysis methods.
Most of the content in this thesis previously appeared in the following publications:


Prof. Dr. Thom Frühwirth was the supervisor of this thesis and accompanied the whole work. He provided me with lots of suggestions and valuable feedback to make the above publications ready for submission. Hence, all of those papers are authored by myself and Prof. Dr. Frühwirth. Nevertheless, the contributions in the papers are the result of my own work and are therefore part of this thesis. A detailed attribution of the previously published contents to the chapters of this thesis is given in Section 1.6.
ACKNOWLEDGMENTS

First and foremost, I would like to thank my supervisor, Thom Frühwirth. He understood to give me the right amount of freedom and support both in teaching and research that motivated me to pursue my work. During my time working for him, I really learned a lot about academic research. He always motivated me to publish my results and his suggestions were most valuable to make the papers ready for publication. Finally, he gave me the chance to travel the world, which I enjoyed a lot.

I also want to thank the other members of the jury: Slim Abdennadher, Maurizio Gabbielli, Birte Glimm and Enno Ohlebusch.

My deepest thanks go to the whole Institute of Software Engineering and Programming Languages that was known as the Institute of Software Engineering and Compiler Construction when I started there. In particular, my gratitude goes to the two directors of the institute during my time there: Helmuth Partsch and Matthias Tichy. They both created the most wonderful and creative working atmosphere and I could not imagine a better place for a PhD project.

The credits for the nice working atmosphere also go to my dear colleagues, Alexander Breckel, Florian Ege, Raffaela Groner, Katharina Juhnke, Stefan Kögel, Jakob Pietron, Alexander Raschke, Michael Stegmaier, Thomas Witte, Marcel Dausend, Alexander Nassal and Tobias Weck. Our countless discussions both about our research, but also about everything else, were always a big pleasure.

Working in academia also consists of a lot of administrative chores. Therefore, I am glad that I could enjoy the support of our secretaries Ulrike Seiter, Eva Englert, Christina Knecht, Kathrin Deißler and Carolin Göhring, who allowed me to concentrate more on my other duties. In this regard, I also want to thank our technical staff Peter Nothelfer and Alexander Hinz.

During my work on a confluence test for ACT-R, the need of more powerful analysis methods became clear. Hence, I want to thank Henning Christiansen and Maja H. Kirkeby, who also worked on the topic of confluence modulo equivalence, for the fruitful discussions and exchange of ideas. Additionally, I am most grateful for their invitation to Roskilde and I really enjoyed the wonderful stay there.

I would like to thank Siegfried Schwill for proof reading the thesis and for his suggestions for improvement of this work.

I also want to thank my parents, who always supported me in pursuing my goals and whose support from the beginning of my life to this day made this work possible.
Finally, I would like to thank my family for just everything. In particular, I want to thank my wife, Stefanie Gall, who always encouraged me and kept my back free, when I had to work late because of an approaching deadline and who never questioned my scientific ambitions.
# CONTENTS

1 **INTRODUCTION** 1  
1.1 Context 1  
1.2 Focus and Scope 3  
1.3 Relevance and Importance 5  
1.4 Goals and Contributions 6  
1.5 Structure of the Thesis 7  
1.5.1 Part i – Preliminaries 7  
1.5.2 Part ii – Program Analysis of Constraint Handling Rules with User-Defined Equivalence Relations 8  
1.5.3 Part iii – Analysis of ACT-R Models 9  
1.6 Bibliographic Remarks 9

I **PRELIMINARIES**

2 **BASIC NOTIONS** 13  
2.1 Fundamental Definitions and Notations 13  
2.2 Program Analysis 15

3 **CONSTRAINT HANDLING RULES** 17  
3.1 Syntax 17  
3.1.1 Prerequisites from First-Order Logic 17  
3.1.2 Definition of the Syntax 18  
3.1.3 Notational Remarks 19  
3.2 Operational Semantics 19  
3.2.1 Preliminaries 20  
3.2.2 Very Abstract Semantics $\omega_{va}$ 23  
3.2.3 Equivalence-Based Semantics $\omega_{e}$ 28  
3.2.4 Other Operational Semantics 31  
3.3 Program Analysis 34  
3.3.1 Merge Operator 34  
3.3.2 Partial Order on States 36  
3.3.3 Monotonicity 37  
3.3.4 Rule States 38  
3.3.5 Confluence 41  
3.3.6 Operational Equivalence 43  
3.3.7 Invariant-Based Program Analysis 44

4 **ADAPTIVE CONTROL OF THOUGHT – RATIONAL** 55  
4.1 Overview of the ACT-R Architecture 55  
4.2 Syntax 57  
4.3 Informal Operational Semantics 59

II **PROGRAM ANALYSIS OF CONSTRAINT HANDLING RULES WITH USER-DEFINED EQUIVALENCE RELATIONS**

5 **USER-DEFINED EQUIVALENCE RELATIONS** 63
5.1 Object Language Level Reasoning 64
  5.1.1 Formalization 65
  5.1.2 State Transitions on the Meta-Level 68
  5.1.3 Case Splitting 70
5.2 Implications for Program Analysis 75
5.3 Compatibility of Equivalence Relations 76
  5.3.1 Definition 77
  5.3.2 Observations About Compatibility 79
  5.3.3 Class of \( p \)-State Equivalence Relations 82
5.4 Discussion 92
5.5 Related Work 93

6 INV.-BASED CONFLUENCE MODULO EQUIVALENCE 97
  6.1 Confluence Modulo Equivalence 99
  6.2 Extension of Confluence Modulo Equivalence by Invariants 100
  6.3 Decision Criterion 101
    6.3.1 Joinability Modulo Equivalence 101
    6.3.2 Criterion for the \( \alpha \)-Property 102
    6.3.3 Criterion for the \( \beta \)-Property 103
    6.3.4 Criterion for Invariant-Based Confluence Modulo Equivalence 106
  6.4 Criterion for \( p \)-State Equivalence 107
  6.5 Invariant-Maintaining Equivalence Relations 110
  6.6 Discussion 114
    6.6.1 Implications of the Results 114
    6.6.2 Relation to Prior Work 116
    6.6.3 Comparison of Confluence Definitions 117
  6.7 Related Work 117

7 OPERATIONAL EQUIVALENCE MOD. EQ. 121
  7.1 Definition 122
  7.2 Decision Criterion 123
  7.3 Discussion 128
  7.4 Related Work 130

III ANALYSIS OF ACT-R MODELS

8 A FORMAL OPERATIONAL SEMANTICS OF ACT-R 133
  8.1 Very Abstract Operational Semantics 134
    8.1.1 Chunk Stores 134
    8.1.2 States 138
    8.1.3 Operational Semantics 141
  8.2 Abstract Operational Semantics 148
  8.3 Discussion 153
    8.3.1 Summary 153
    8.3.2 Relation to Prior Work 155
  8.4 Related Work 156
    8.4.1 Semantics of Albrecht and Westphal 156
8.4.2 Other Work 158

9 EMBEDDING ACT-R MODELS INTO CHR 161
9.1 Translation to Constraint Handling Rules 161
   9.1.1 Set Normal Form 161
   9.1.2 Notational Aspects 163
   9.1.3 Translation of States 163
   9.1.4 Translation of Rules 164
   9.1.5 No-Rule Transition 168
9.2 Soundness and Completeness of the Translation 169
9.3 Discussion 179
   9.3.1 Summary 179
   9.3.2 Relation to Prior Work 180
9.4 Related Work 181

10 CONFLUENCE CRITERION FOR ACT-R 183
10.1 Generalization of the Embedding 184
10.2 ACT-R Invariant 186
10.3 Invariant-Based Confluence Test 189
   10.3.1 Criterion for the $\alpha$-Property 189
   10.3.2 Criterion for the $\beta$-Property 190
   10.3.3 Definition of the Confluence Criterion 191
10.4 Discussion 194
   10.4.1 Isomorphic Chunk Networks 194
   10.4.2 Module Requests 195
   10.4.3 Generality of Approach 196
   10.4.4 Relation to Prior Work 196
10.5 Related Work 197

IV CONCLUSION AND FUTURE WORK

11 CONCLUSION 201
11.1 Program Analysis of Constraint Handling Rules with
      User-Defined Equivalence Relations 201
11.2 Analysis of ACT-R Models 202
11.3 Summary of Contributions 204
11.4 Evaluation of the Approach 205

12 FUTURE WORK 207
12.1 Program Analysis of Constraint Handling Rules with
      User-Defined Equivalence Relations 207
12.2 Analysis of ACT-R Models 208

BIBLIOGRAPHY 211
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Hasse diagram of the set $2{a,b,c}$ and the partial order relation $\subseteq$.</td>
<td>14</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Confluence diagrams for confluence and local confluence.</td>
<td>16</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Hasse diagram for reasoning from rule states about larger states by monotonicity.</td>
<td>39</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>Hasse diagram with invariants.</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Two count facts with names $b$ and $c$ that model the counting chain $1, 2, 3$ ([GF15b, p. 115; GF18a, p. 224]).</td>
<td>55</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Modular architecture of ACT-R – inspired by ([TLa06; And+04]). Appeared in ([GF15b, p. 115; GF18a, 224]).</td>
<td>56</td>
</tr>
<tr>
<td>Figure 5.1</td>
<td>Confluence diagram with case distinctions for the built-in constraints.</td>
<td>74</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>Confluence diagram with case distinctions for the built-in constraints (cont.).</td>
<td>74</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Confluence diagram for confluence modulo a user-defined equivalence relation $\approx$ ([GF18b, p. 118]).</td>
<td>99</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Confluence diagrams for local confluence modulo equivalence.</td>
<td>100</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Hasse diagrams for $\mathcal{M}_e(\sigma_1, [\sigma_2])$.</td>
<td>104</td>
</tr>
<tr>
<td>Figure 8.1</td>
<td>Visual representation of the very abstract state defined in Example 42. The dashed arrow signifies that the chunk in the retrieval buffer is not yet visible (as indicated by the delay right of the buffer’s name) ([GF15b, p. 118; GF18a, 2211]).</td>
<td>140</td>
</tr>
<tr>
<td>Figure 8.2</td>
<td>Visual representation of state $\sigma_2$ from Example 46 ([GF15b, p. 120; GF18a, p. 2221]).</td>
<td>154</td>
</tr>
</tbody>
</table>
LIST OF TABLES

Table 8.1  Parameters of the very abstract semantics that must be defined by the architecture or the cognitive model respectively. 148
Table 8.2  Parameters of the abstract semantics that must be defined by the architecture. 152
LIST OF SYMBOLS

ACT-R SYNTAX AND SEMANTICS
\( \mathcal{C}_A \) set of constants 57
\( \mathcal{V}_A \) set of variables 57
\( B \) set of buffers 57
\( T \) set of types 57
\( \mathcal{R} \) set of types 57
\( \Rightarrow \) ACT-R production rule 57
\( \Lambda \) set of action symbols 58
\( = \) modification action and buffer test 58
\( + \) request action 58
\( - \) clearing action 58
\( \mathcal{A} \) set of actions 58
\( \tau \) typing function 134
\( \text{chunk} \) empty chunk type 134
\( \Delta \) chunk store 134
\( \Delta^\subset \) partial chunk store with reference to \( \Delta \) 136
\( D \) set of all chunk stores 135
\( D^\subset_{\Delta} \) set of all partial chunk stores with reference to \( \Delta \) 136
\( ::= \) chunk definition 134
\( \text{val} \) slot valuation function 134
\( id_\Delta \) chunk identifier function of chunk store \( \Delta \) 134
\( \text{nil} \) unique empty chunk 134
\( \circ \) chunk merging operator 136
\( \gamma \) cognitive state 138
\( \upsilon \) additional information 138
\( \Gamma \) set of cognitive states 138
\( \Gamma_\text{part} \) set of partial cognitive states 138
\( \Upsilon \) set of allowed predicates for additional information 139
\( \sqsubseteq \) a buffer test or a rule matches an ACT-R state 149

CHR STATES
\( G \) goal store in the CHR state tuple 21
**LIST OF SYMBOLS**

- **χ** built-in constraint store in the CHR state tuple  
  - 21
- **V** global variables in the CHR state tuple  
  - 21
- **σ** empty CHR state  
  - 21
- **σ** equivalence class of state  
  - 30
- **⋄** merge operator of CHR states  
  - 35
- **◁** partial order over CHR states  
  - 37

**CHR SYNTAX AND SEMANTICS**

- **⇔** simplification and simpagation rule  
  - 19
- **⇒** propagation rule  
  - 19
- **P** CHR program  
  - 18
- **ω**\textsubscript{va} very abstract operational semantics of CHR  
  - 23
- **ω**\textsubscript{e} equivalence-based operational semantics of CHR  
  - 28

**EQUIVALENCE RELATIONS**

- **=** algebraic/mathematical equality  
  - 14
- **::=** definition of an expression or symbol  
  - 14
- **≡** syntactic equality  
  - 14
- **≡** CHR state equivalence  
  - 21
- **≈** user-defined equivalence relation over states  
  - 14
- **≈**\textsubscript{S} user-defined equivalence relation for multi-sets  
  - 66, 83
- **≈**\textsubscript{A} user-defined equivalence relation for ACT-R state equivalence  
  - 185
- **≈** equivalence of the set representations of two chunk stores  
  - 185

**FIRST-ORDER LOGIC**

- **Φ** set of function symbols in the signature of a first-order language  
  - 17
- **Π** set of predicate symbols in the signature of a first-order language  
  - 17
- **⊥** logical falsity  
  - 17
- **⊤** logical truth  
  - 17
- **V** set of variables in the signature of a first-order language  
  - 17
- **x** sequence of variables  
  - 17
- **∀F** universal closure of formula F  
  - 13
xvi LIST OF SYMBOLS

∃F existential closure of formula F 14
θ substitution 18
θ† constraint representation of substitution θ 65
 êθ state representation of substitution θ 65
CT constraint theory 20

LIST SYMBOLS

+ list concatenation 14
[] empty list 14
[H|T] list with head element H and tail (remaining list) equivalent to [H] + T 14
[x : p(x)] list of elements from set M that satisfy p 163
[f'] sorted list representation of a function f with finite domain 163

PROGRAM ANALYSIS

I generic invariant 44
BW Blocks World invariant 45
A ACT-R invariant 187
P^I reduced state transition system of invariant I 44
Σ^I set of extensions that reestablish invariant I 47
M^I set of minimal extensions that reestablish invariant I 47
NF normal forms 52
I interface of CHR programs 51, 122

SET SYMBOLS

∅ empty set 14
∪ set union 14
∩ set intersection 14
⊎ multi-set union 14
× Cartesian product 14
2^S power set of set S 14
⊆ subset relation including equivalence 14
⊂ subset relation excluding equivalence 14
|S| cardinality of set S 14
{x | p(x)} set of all elements x, where p(x) holds 14
## Transition Systems

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\sigma$</td>
<td>state</td>
<td>14</td>
</tr>
<tr>
<td>$\tau$</td>
<td>state (usually after rule application)</td>
<td>14</td>
</tr>
<tr>
<td>$\rho$</td>
<td>state</td>
<td>14</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>generic state transition, usually abbreviation for $\rightarrow_e$</td>
<td>29</td>
</tr>
<tr>
<td>$\rightarrow^*$</td>
<td>reflexive transitive closure of a transition relation</td>
<td>15</td>
</tr>
<tr>
<td>$\rightarrow^+$</td>
<td>transitive closure of a transition relation</td>
<td>15</td>
</tr>
<tr>
<td>$\rightarrow_P$</td>
<td>state transition in program $\mathcal{P}$</td>
<td>123</td>
</tr>
<tr>
<td>$\rightarrow_{va}$</td>
<td>state transition relation of the very abstract operational semantics $\omega_{va}$ of CHR</td>
<td>24</td>
</tr>
<tr>
<td>$\rightarrow_e$</td>
<td>state transition relation of the equivalence-based operational semantics $\omega_e$ of CHR</td>
<td>28</td>
</tr>
<tr>
<td>$\rightarrow$</td>
<td>state transition relation in the (very) abstract semantics of ACT-R</td>
<td>145</td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>set of states</td>
<td>15</td>
</tr>
<tr>
<td>$\Sigma_{AvA}$</td>
<td>set of very abstract ACT-R states</td>
<td>138</td>
</tr>
<tr>
<td>$\Sigma_{CHR}$</td>
<td>set of CHR states</td>
<td>21</td>
</tr>
<tr>
<td>$\downarrow$</td>
<td>joinability of states</td>
<td>15</td>
</tr>
<tr>
<td>$\downarrow\approx$</td>
<td>joinability modulo equivalence relation</td>
<td>99</td>
</tr>
</tbody>
</table>
INTRODUCTION

Everything we do, every thought we’ve ever had, is produced by the human brain. But exactly how it operates remains one of the biggest unsolved mysteries, and it seems the more we probe its secrets, the more surprises we find.

— Neil deGrasse Tyson

1.1 CONTEXT

Understanding the functionality of the human mind is a question that at least goes back to the ancient Greek philosophers like Plato and Aristotle [Tha19]. Even though nowadays methods of artificial intelligence and neuroinformatics have progressed so far that for specific tasks computers outperform the best human agents in the respective fields like for instance Chess or Go [Cos+08; Så+17a; Så+18; Så+17b], those programs do not explain the representations and processes of the human mind, i.e. they do not focus on “faithfully modeling human intelligence” [Ta08, p. 170].

Cognitive science is an interdisciplinary research field examining the human mind by methods of philosophy, psychology, artificial intelligence, neuroscience, linguistics, and anthropology [Tha19]. The fundamental idea of cognitive science is to combine theoretical models of cognition with experiments performed on human subjects to validate the theories and its predictions. This follows the scientific consensus that “experiment without theory is blind, but theory without experiment is empty” [Tha07] arguably attributed to Immanuel Kant. The theories in cognitive science follow the hypothesis that an understanding of the human mind can best be achieved by understanding both the “representational structures in the mind and computational procedures that operate on those structures” [Tha19].

Usually, modeling in cognitive science can be roughly categorized into three types: verbal-conceptual, mathematical and computational modeling [Sun08, p. 4]. Whereas verbal-conceptual models use informal natural language descriptions of the representations and processes of the cognitive tasks considered, mathematical models describe the relations of those entities mathematically to make quantifiable predictions with mathematical equations [Sun08, p. 4].

Computational cognitive modeling uses algorithmic descriptions of the representations and processes of cognition. Following the idea of Turing that mental activity is computation [Tur50; Sun08, p. 3; TA10, p. 693], the methods of computational cognitive modeling are to build a “detailed, process-based understanding by specifying corre-
Cognitive Architectures

Examples of Cognitive Architectures

sponding computational models (in a broad sense) of representations, mechanisms, and processes” [Sun08, p. 3].

In the literature, cognitive models are arguably roughly categorized into symbolic or connectionist models. There are also hybrid models that embrace concepts of both categories [VMS07; Sun08, p. 5; LCF17]. Symbolic models apply methods from symbol manipulation languages, i.e. they use symbolic representations of knowledge that are typically manipulated by a set of if–then rules. Connectionist models try to understand cognition by artificial neural networks that are a simplified model of the biological structures of the brain. While connectionist models are lately very successful in the domains they have been trained for, their possibility of adaptation to similar problems is limited and it is unclear how this can be achieved in the future [LCF17, p. 1].

In contrast, symbolic models have proven to be adaptable to a variety of problems and their theory of cognition may serve as a basis for other models. For this reason, cognitive architectures emerged which support the modeling process by providing a formal unified theory of cognition that allows for building cognitive models of specific tasks and cognitive features.

There are numerous cognitive architectures. While there are purely symbolic architectures, most architectures try to more or less combine both approaches. The cognitive architecture States, Operators, And Reasoning (SOAR) [LNR87; New90] is a purely symbolic approach [TA08, p. 172] that “views all intelligent behavior as a form of problem solving” [TA08, p. 172], i.e. human intelligence is considered as an approximation of a knowledge system [TA08, pp. 171–172]. It has its origins in the knowledge system General Problem Solver (GPS) [NS63] developed by Newell and Simon [TA08, p. 171]. Thus, when SOAR is simulating human behavior, “a search process tries to accomplish a goal state through a series of operators” [TA08, p. 172]. Due to its symbolic character, all knowledge in SOAR is explicit and represented by symbolic structures. There are no implicit representations of knowledge like activation or utility values that are attached to the symbolic constructs [TA08, p. 172]. Every model in SOAR makes use of “a single long-term memory, a single learning mechanism, and only symbolic representations” [TA08, p. 172]. Although such a constrained architecture has its advantages in theoretical considerations, it is too strict to implement practical cognitive models. Hence, there are current endeavors of enhancing the purely symbolic representations of the traditional SOAR architecture [TA08, p. 172; NL05].

Adaptive Control of Thought – Rational (ACT-R) combines methods and techniques from both symbolic and connectionist architectures and can therefore be classified as a hybrid cognitive architecture. The two concepts are represented in ACT-R’s distinction of a symbolic and a sub-symbolic level. “The symbolic level […] is an abstract character-
ization of how brain structures encode knowledge” [Ando7, p. 33]. It uses symbolic representations of knowledge and its connections that can be manipulated by production rules. “The sub-symbolic level is an abstract characterization of the role of neural computation in making that knowledge available” [Ando7, p. 33]. It enhances those symbolic structures by quantities such as activation levels of declarative knowledge or utility values of production rules. Activation levels decide if and how fast a chunk of information is available to the system and utility values represent the success and effort of production rules. The values of those sub-symbolic quantities are adapted over time and therefore represent a form of learning.

The cognitive architecture Connectionist Learning with Adaptive Rule Induction Online (CLARION) [SST05] uses neural networks for many but not all of its representations. It is therefore considered a more truly hybrid architecture than ACT-R, since it features both true symbolic and connectionist characteristics [TAo8, p. 174].

1.2 Focus and Scope

Currently, cognitive architectures as well as the implementations of cognitive models are typically ad-hoc constructs. They lack a formalization from the computer science point of view. For instance, as mentioned before, Adaptive Control of Thought – Rational (ACT-R) [And+04] is a widely employed cognitive architecture that embraces both symbolic and connectionist concepts. It has a well-defined psychological theory, however, its computational system is not described formally leading to implementations that are full of technical artifacts as e.g. claimed in [AW14b; SW07]. Furthermore, there are various variants of the underlying theory of ACT-R [GF14; BR04] and several implementations of the architecture [Bot; SW07; Har08; Sal].

The ad-hoc fashion of cognitive architectures and the technical artifacts impede analysis of the underlying languages and the programmed models. They make it hard to compare different variants of the theories and language implementations. They make it hard to verify properties of the models. These issues call for a formal semantics of cognitive modeling languages together with proper analysis techniques.

The main goal of this thesis is to use and extend the program analysis results from computer science and combine them with psychological methods from computational cognitive modeling to eliminate ad-hoc cognitive modeling. For this purpose, a formal operational semantics for a popular cognitive modeling language is developed and an analysis framework is built upon it.

In more detail, this thesis is interested in the formalization of cognitive modeling languages to first of all enable a formal examination of the topic.
To better understand the results of cognitive models, they are analyzed for the confluence property which ensures that every possible computation yields the same result. Although cognition seems to be highly non-deterministic and non-confluent, typical large cognitive models usually consist of deterministic and non-deterministic parts. Confluence analysis helps to identify those parts and helps the modeler to decide whether an unveiled non-determinism is desired or not.

The question if two computational cognitive models are equivalent, i.e. yield the same results, is difficult to answer. However, such an analysis is of benefit to compare different solutions to the same problem and to formally analyze if changes of a model affect the result of a model. It can also be used to unveil redundant parts of the model.

This thesis concentrates on the popular cognitive architecture ACT-R. There are numerous cognitive models that use ACT-R demonstrating its popularity and applicability in various domains. For instance, there are models of problem solving and decision making [MM11; Belo6], language learning, e.g. of the past tense [TA02], learning and memory [And+98; Taag9; LWT98; BSC04], time perception [TRA07], or threaded cognition [STo8]. There are also applications in human computer interaction [Byro1] and tutoring systems [Sal17, p. 23]. Furthermore, ACT-R models can be used to predict brain activity that can be validated by functional magnetic resonance imaging (fMRI) studies on human subjects [BA15; And07].

The popularity of ACT-R and its wide dissemination make ACT-R a promising candidate for our work on analysis of cognitive models as the results are directly available to a large number of cognitive models and applications. The hybrid nature of ACT-R makes us confident that the results of this thesis can be transferred to other production rule centric architectures.

Since ACT-R is a production rule system, it seems likely that its analysis is related to analysis of other rule-based programming languages. Constraint Handling Rules (CHR) [Frü92; Frü95; Frü98; Frü09; Sne+10b; Frü15] is a rule-based programming language from the field of (constraint-) logic programming. In contrast to the ad-hoc nature of state of the art cognitive architectures, CHR is a clean and declarative production rule system. It has a formally defined operational semantics as well as a declarative semantics with corresponding soundness and completeness results. Due to its origins in logic programming, CHR has a strong theoretical background and is both a formalism and an executable programming language. In particular, CHR offers highly optimizing compilers [SS08; Sne08; Sne09; SS09; Van10a; Van10b] and allows for implementing every algorithm in optimal time and space complexity [SSD09].

There are many theoretical and practical results and tools for the analysis of CHR programs [Frü09]. For instance, there is a decidable
sufficient and necessary confluence criterion for terminating programs as well as a result on operational equivalence for terminating, confluent programs. In general, confluence analysis has a great history in CHR research [AFM96; Abd97; AFM99].

Due to its strengths in formal program analysis and its strong relation to first order [Frü09] and linear logic [Frü09; BF05], it has been used as a lingua franca that embeds many rule-based approaches [Frü09] like term rewriting systems [RFo8], graph transformation systems [Rai07; RF11] and business rules [MF07]. Such embeddings have been used successfully to make the analysis results of CHR available to other approaches, leading to a cross-fertilization between the different languages and formalisms.

This thesis aims to pursue the lingua franca approach for cognitive modeling languages. The main hypothesis of this approach is that CHR can be used to analyze cognitive models by developing a sound and complete embedding of ACT-R making the formal reasoning and analysis techniques of CHR available to cognitive modeling. For this purpose, existing CHR analysis techniques are extended by user-defined equivalence relations over states. This allows to use them to reason about the embedded ACT-R models.

### 1.3 Relevance and Importance

The semantics of ACT-R’s theory is informally defined by many different research and survey papers that differ in numerous details and also lack a definition of the operational semantics [e.g. And+04; TLA06; And07]. The behavior of ACT-R is defined by its reference LISP implementation that comes with a technical, but by no means formalized, description in a reference manual [Bot]. The lack of a formal description of its operational semantics has also been pointed out in the literature [AW14b; SW07]. Analysis of cognitive models usually assumes an implicit understanding of ACT-R’s semantics. There have been some efforts on formalizing ACT-R or cognitive modeling in general [Bri08; AW14b; AGW14] substantiating the necessity of a clean description of its functionality. However, our formalization is – to the best of our knowledge – the first formal description of the operational semantics. Independent from our work, another operational semantics has been developed [AW14b]. In this thesis, we show that our operational semantics can be expressed as an instance of an improved version of this related semantics, confirming that our semantics seems to correspond to the accepted understanding of ACT-R.

Since ACT-R is a rule-based language, the question of confluence arises naturally, as it is an important property in rule-based programming. To the best of our knowledge, there has been no research on confluence analysis for ACT-R or other cognitive architectures before. Furthermore, our results indicate that it is by no means trivial to con-
struct a confluence test for ACT-R due to the structure of its underlying concepts. This is supported by the fact that the canonical results on confluence in CHR have to be extended by reasoning on user-defined equivalence relations over states to capture the conditions of ACT-R. This thesis solves the beforehand open question of how a confluence criterion for ACT-R can be designed. On the practical side, identifying the parts of the program that may lead to non-deterministic behavior improves the understanding of cognitive models, in particular for larger models.

Our work on operational equivalence that is directly available for ACT-R enables to compare cognitive models, to understand how changes on the models affect its results, and to find redundant rules. Pursuing the spirit of the lingua franca approach, our results on program analysis with user-defined equivalence relations are directly available for other language embeddings of CHR and obviously for all other general purpose CHR programs as well.

1.4 GOALS AND CONTRIBUTIONS

Following the main goal of analyzing ACT-R models by extending CHR analysis techniques, the thesis has the following objectives:

FORMALIZATION OF ACT-R Since there is no formal description of ACT-R’s behavior, an operational semantics is developed. There are many different ACT-R variants, and hence the semantics is formulated in two degrees of abstraction:

1. A very abstract semantics that formalizes the overall architecture of ACT-R as a production rule system. This semantics only defines the components of which typical ACT-R systems are composed of, but leaves those components undefined. This allows to describe as many different variants of ACT-R in one semantics and might be capable of capturing other production rule centric cognitive architectures related to ACT-R as well.

2. An abstract semantics is an instance of the very abstract semantics that defines the central part of ACT-R: the procedural system. It defines the general recognize-act-cycle by defining how rules are matched and what effects they have. However, the sub-symbolic effects like rule selection (conflict resolution) are not defined in detail, but captured in a non-deterministic fashion. This allows for a general reasoning about the procedural behavior of ACT-R which we are focusing on in this thesis. Hence, the abstract semantics is a central contribution and is used throughout the rest of this thesis.
1.5 Structure of the thesis

The thesis is structured into three main parts that are described in this section. Part iv concludes the thesis by summarizing its contribution and giving a prospect of future work.

1.5.1 Part i – Preliminaries

To make this thesis self-contained to readers that are not familiar with ACT-R or CHR, the foundations of those two systems necessary to understand the main part of the thesis are briefly summarized in this part.
Chapter 2 introduces some basic notations and definitions used throughout the thesis, Chapter 3 introduces syntax and semantics of CHR as well as the program analysis methods used in this thesis. Chapter 4 briefly summarizes the functionality of ACT-R.

1.5.2 Part ii – Program Analysis of Constraint Handling Rules with User-Defined Equivalence Relations

Chapter 5 first motivates reasoning with user-defined equivalence relations and then discusses how CHR program analysis can be used for this purpose. User-defined equivalence relations summarize a set of states to equivalence classes and hence reasoning about such relations often includes reasoning on a more abstract meta-level that allows for reasoning about sets of states. Therefore, the chapter starts with a general discussion of how CHR states can be used directly to describe such sets of states without introducing a meta-level. Then, the implications and problems of user-defined equivalence relations for program analysis are discussed. Since such equivalence relations may introduce many difficulties for program analysis due to their generality, a restricted class of equivalence relations, to which the proposed analysis techniques are applicable, is defined: the so-called compatible equivalence relations. It is shown that this restriction still leads to a meaningful class of equivalence relations by proposing a general pattern of building such relations that can be extended by arbitrary first-order predicates qualifying the equivalence of states. Then some observations on the proposed class of compatible equivalence relations are discussed.

Chapter 6 gradually develops a sufficient and necessary criterion for invariant-based confluence modulo equivalence. This extends the existing results on confluence as well as confluence with an invariant on states and allows to combine them with user-defined equivalence relations. Such invariants allow for restricting (confluence) analysis to only those states that can be observed in practical applications. The proposed criterion is then simplified for the special case of equivalence-maintaining invariants, i.e. in an equivalence class of states the invariant either holds or does not hold for all states in that class. Then, some example programs are discussed demonstrating the application of the proposed confluence criterion. Eventually, the results are discussed and compared to related work.

In Chapter 7, the ideas of the confluence modulo equivalence result to develop a criterion for operational equivalence modulo equivalence. After defining the notion of invariant-based operational equivalence modulo equivalence, a criterion for the latter is gradually developed, discussed and compared to related work.
1.5.3 Part iii – Analysis of ACT-R Models

In Chapter 8, the formalization of ACT-R is presented in two degrees of abstraction. The very abstract semantics defines the basic architecture as a modular production rule system. The abstract operational semantics is an instance of the very abstract semantics and for example defines how the matching of rules operates in typical ACT-R systems.

Chapter 9 defines the embedding of ACT-R in CHR. It starts with some formal preliminaries before defining the translation scheme for both states and rules. Then, the embedding of the so-called no-rule transition from the operational semantics is described that handles the behavior when no rule is applicable. The embedding is shown to be sound and complete with respect to the abstract operational semantics from Section 8.2.

Using the formalization of ACT-R and its embedding into CHR, a decidable sufficient and necessary confluence criterion is developed in Chapter 10. It uses the methods of invariant-based confluence modulo equivalence described in Chapter 6. This includes a result that allows to simplify the representation of ACT-R states in CHR as it has been done informally in prior work.

1.6 Bibliographic Remarks

Most of the work presented in this thesis has already been formally published in conference proceedings or journals. Some chapters in this thesis are adapted or extended versions of the published material. In this section, the chapters in this thesis are mapped to the prior publications by discussing their relation. Where necessary, a more detailed enumeration of the differences between the published version and the version in this thesis is given in the corresponding chapters. On Page v, a compilation of the publications incorporated into this thesis can be found.

• Chapter 5 contains mostly new, unpublished work. Only parts of Sections 5.2 and 5.3 appeared in [GF18b], but have been extended by new insights. The parts about meta-level reasoning, defining compatible equivalence relations by using an equivalence relation pattern and the observations on compatible equivalence relations are completely new and create a deeper understanding of the prior published results.

• Chapter 6 is a heavily extended version of [GF18b]. The results have been generalized such that they are available for more invariants and equivalence relations and hence are widening the scope of the published results. The examples have not been published before.

• Chapter 7 is completely new.
• Chapter 8 is a slightly adapted version of the corresponding sections in [GF18a]. It combines the results from [GF15a; GF15c; GF15b].

• Chapter 9 is a slightly revised version of the corresponding section in [GF18a] that itself is based on [GF16]. The first ideas of the translation are loosely based on [GF14].

• Chapter 10 is an extended version of [GF17]. The extensions formalize the formerly simplified parts of the published confluence test using user-defined equivalence relations. This comes with a novel unpublished result that allows to use set notation for chunk stores in the CHR translation as informally done in [GF17] instead of assuming an order on chunks as proposed by the translation in Chapter 9 and [GF18a].
Part I

PRELIMINARIES
2 BASIC NOTIONS

2.1 FUNDAMENTAL DEFINITIONS AND NOTATIONS

The concepts and symbols presented in this section are mostly standard in the literature. As sometimes differing terms and notations are used, this section clarifies those notions. However, it is not meant as a complete and self-contained introduction to the fundamental topics affected by those notions. Therefore, for a better introduction on those topics, we refer to the commonly accepted textbooks.

As usual, a (n-ary) relation $R$ is defined as $R \subseteq A_1 \times \cdots \times A_n$ for sets $A_i$ ($1 \leq i \leq n$). For binary relations, we usually use infix notation. An equivalence relation is a reflexive, symmetric and transitive relation. A partial order is a reflexive, antisymmetric and transitive relation. We refer to the transitive closure of a relation $R$ as $R^+$ and to its reflexive transitive closure as $R^*$. A function is a left-total and right-unique binary relation $f \subseteq A \times B$ where $A$ is called the domain and $B$ is called the co-domain of $f$. We also write $f : A \to B$ for a function with domain $A$ and co-domain $B$. Note that the domain $A$ and co-domain $B$ might be a Cartesian product of sets. We write $\text{dom}(f)$ to denote the domain of the function $f$. The set of functions from $A$ to $B$ (i.e. the set of all functions with domain $A$ and co-domain $B$) is denoted as $B^A$. A partial function is only required to be a right-unique binary relation.

Finite partially ordered sets can be graphically represented by a Hasse diagram by representing its transitive reduction as a directed acyclic graph. This means that for a partial order $\leq$ over some set $S$, each element of $S$ is represented by a vertex in the corresponding Hasse diagram. There is an edge from element $a \in S$ to element $b \in S$, whenever $a < b$, but there is no $c \in S$ such that $a < c < b$. Thereby, $x < y$ means that $x \leq y$, but $x \neq y$. Typically, the direction of the edges is expressed by drawing $b$ above $a$ if $a < b$.

**Example 1** (Hasse Diagram). Let $S = \{a, b, c\}$. The Hasse diagram of the power set $2^S$ and the subset relation $\subseteq$ is depicted in Fig. 2.1.

In this thesis, Hasse diagrams are sometimes used to illustrate the relations of states in a state transition system (even for infinite sets of states) using a partial order over those states. The notion of a state transition system is defined in the following Section 2.2 about program analysis. The partial order over Constraint Handling Rules (CHR) states is derived in Lemma 6.

This work makes use of the following symbols and notations:
Figure 2.1: Hasse diagram of the set \(2^{\{a,b,c\}}\) and the partial order relation \(\subseteq\).

- \(\forall F\) universal closure of formula \(F\)
- \(\exists F\) existential closure of formula \(F\)
- \(\emptyset\) empty (multi-) set
- \(\cup\) infix operator for set union
- \(\cap\) infix operator for set intersection
- \(\uplus\) infix operator for multi-set union
- \(\times\) Cartesian product of two sets
- \(2^S\) the power set of set \(S\)
- \(\subseteq\) subset relation including equivalence and \(\subset\) for actual subsets
- \(|S|\) cardinality of set \(S\)
- \(\{x \mid p(x)\}\) set of all elements \(x\), where \(p(x)\) holds
- \(\:+\) infix operator for list concatenation
- \([\,\,]\) empty list
- \([H|T]\) list with head element \(H\) and tail (remaining list) \(T\) equivalent to \([H]\:+\:T\)
- \(\approx\) denotes equality in the algebraic/mathematical sense, i.e. the two related mathematical expressions refer to the same mathematical object.
- \(S := e\) is the defining variant of algebraic/mathematical equality where the symbol \(S\) is defined as expression \(e\).
- \(\doteq\) denotes syntactic equality, usually in the sense of Clark’s Equality Theory [Cla78; JLM84, pp. 218–220; FA03, p. 20].
- \(\approx\) and its occurrences with indices etc. usually denote an (arbitrary or user-defined) equivalence relation over states.
- Usually, $\sigma, \sigma', \sigma_1, \ldots, \tau, \tau', \tau_1, \ldots$ and $\rho, \rho', \rho_1, \ldots$ denote states in a state transition system, where the $\tau$-variants are usually states resulting from the application of a transition rule.

2.2 PROGRAM ANALYSIS

A central concept of program semantics and program analysis is the notion of a state transition system.

**Definition 1** (State Transition System [Frü09, pp. 51–52]). A state transition system is a tuple $(\Sigma, \rightarrow)$ where $\Sigma$ is an arbitrary (possibly infinitely large) set of states and $\rightarrow \subseteq \Sigma \times \Sigma$ is a transition relation over the states.

A transition system is deterministic if there is at most one transition from every state, i.e. it is right-unique and therefore a partial function. Otherwise, the transition system is non-deterministic.

A state $\sigma \in \Sigma$ is a final state, if there is no $\tau \in \Sigma$ such that $\sigma \rightarrow \tau$. A derivation or computation is a sequence of states $\sigma_0, \sigma_1, \ldots$, often written as $\sigma_0 \rightarrow \sigma_1 \rightarrow \ldots$, such that $\sigma_0 \rightarrow \sigma_1 \wedge \sigma_1 \rightarrow \ldots \wedge \ldots$. A computation is finite or terminating, if its sequence is finite. Otherwise it is infinite or non-terminating.

All operational semantics that are subject of this thesis are defined as state transition systems. Recall that by $\rightarrow^+$ we denote the reflexive transitive closure of $\rightarrow$ and by $\rightarrow^*$ we denote the transitive closure of $\rightarrow$.

The transition relation of a state transition system may have certain properties of interest. One important property is confluence. Intuitively, a program is confluent if every computation started in some state leads to the same result state, i.e. the program is deterministic considering its observable results (but not necessarily in the way those results are obtained). In particular, for rule-based programming languages, this means that the rule selection in confluent programs does not play a role for the outcome, since every computation leads to the same result state.

**Definition 2** (Confluence [Hue80, p. 799; Frü09, p. 102]). Two states $\sigma_1$ and $\sigma_2$ are joinable, if there exists a state $\tau$ such that $\sigma_1 \rightarrow^* \tau$ and $\sigma_2 \rightarrow^* \tau$. We then write $\sigma_1 \downarrow \sigma_2$.

A state transition system is confluent, if for all states $\sigma, \sigma_1, \sigma_2$ it holds that if $\sigma \rightarrow^* \sigma_1$ and $\sigma \rightarrow^* \sigma_2$, then $\sigma_1 \downarrow \sigma_2$.

The notion of confluence is illustrated in Fig. 2.2a that shows a so-called confluence diagram for the state $\sigma$. In such diagrams, states are depicted as nodes of a directed graph, whereas (transitive) transitions are depicted as arrows (labeled with $\ast$). The dashed arrows denote the existentially quantified transitions on the right hand side of the implication in Definition 2 that stem from joinability.
Since the state spaces of typical transition systems are infinitely large, the confluence property is not trivial to prove in general. The weaker definition of \textit{local confluence} plays an important role for program analysis:

\textbf{Definition 3} (Local Confluence [Hue80, p. 800; Frü09, p. 104]). A state transition system is \textit{locally confluent}, if for all states \(\sigma, \sigma_1, \sigma_2\) it holds that if \(\sigma \rightarrow \sigma_1\) and \(\sigma \rightarrow \sigma_2\), then \(\sigma_1 \downarrow \sigma_2\).

The difference to the confluence definition is that the transitions from \(\sigma\) to \(\sigma_1\) and \(\sigma_2\) are required to only take one transition step. The confluence diagram for local confluence is depicted in Fig. 2.2b.

It has been shown in \textit{Newman’s Lemma} that confluence and local confluence coincide for terminating programs.

\textbf{Lemma 1} (Newman’s Lemma [New42; Hue80, p. 800; Frü09, p. 104]). A terminating reduction system is confluent if and only if it is locally confluent.

This means that for terminating programs, confluence analysis can be reduced to proving local confluence.

A different subject of interest in program analysis is the notion of program equivalence. The notion of equivalence of two programs is ambiguous: Should the two programs perform the same execution steps? Should they share the exact same source code? There are many other possible understandings of the notion of program equivalence. In this thesis, we are interested in the notion of \textit{operational equivalence} that considers two state transition systems equivalent, if for any given start state both programs lead to the same answer.

\textbf{Definition 4} (Operational Equivalence [Frü09, p. 128]). Let \(\rightarrow_{P_1}\) and \(\rightarrow_{P_2}\) be the transition relations of two state transition systems \(P_1\) and \(P_2\).

\(P_1\) and \(P_2\) are operationally equivalent, if and only if for all states \(\sigma\) there are computations such that \(\sigma \rightarrow_{P_1}^* \tau\) and \(\sigma \rightarrow_{P_2}^* \tau\) or \(\sigma\) is a final state in both programs.
In this chapter, we recall syntax and semantics of Constraint Handling Rules (CHR) mainly based on [Frü09] (c.f. Sections 3.1 and 3.2). Then, we summarize some established program analysis techniques that are used and extended in the main parts of the thesis in Section 3.3.

3.1 SYNTAX

This section introduces the syntax of CHR. It starts with some prerequisites from first-order logic in Section 3.1.1. Then the actual syntax of CHR rules is defined in Section 3.1.2. Eventually, some differences in notations found in the literature are discussed in Section 3.1.3.

3.1.1 Prerequisites from First-Order Logic

Since CHR is a first-order language, we begin with some basic definitions from first-order logic. This section is mainly based on the standard definitions of [Frü09, pp. 49–50] and [FA03, pp. 125–130].

3.1.1.1 Signature of a First-Order Language

The signature of a first-order language consists of a set of variables $\mathcal{V}$, a set of function symbols $\Phi$ and a set of predicate symbols $\Pi$. Function and predicate symbols are associated with an arity for the number of arguments they take. We write $c/n$ for a function or predicate symbol $c$ with arity $n$. Function symbols with arity 0 are called constants, predicate symbols with arity 0 are called propositions.

A term is either a variable or a function term of the form $f(t_1, \ldots, t_n)$ where $f/n \in \Phi$ and all $t_i(1 \leq i \leq n)$ are terms. An atomic formula or atom has the form $p(t_1, \ldots, t_n)$ where $p/n \in \Pi$ and all $t_i(1 \leq i \leq n)$ are terms. The symbol $\bot$ denotes logical falsity, whereas $\top$ denotes logical truth.

Terms and formulas are called (logical) expressions. We say that an expression is ground, if it does not contain any variables. The notation $\overline{x}$ denotes a sequence of variables.

For all expressions, the function $\text{vars}(E)$ maps the expression $E$ to the set of variables that appear in $E$. We use the notation $E[X]$ to denote that the variable $X$ appears in expression $E$, i.e. $X \in \text{vars}(E)$. 


3.1.1.2 Substitutions and Variants

A substitution is a finite function \( \theta \) from variables to terms written as \( \theta = \{ X_1/t_1, \ldots, X_n/t_n \} \) where \( X_i \) are variables and \( t_i \) are terms \((1 \leq i \leq n)\) and each \( X_i \neq t_i \). For an expression \( E \) and a substitution \( \theta \) the expression \( E\theta \) is obtained by replacing each occurrence of a variable \( X_i \) in \( E \) by the corresponding term \( t_i \) for each \( X_i/t_i \in \theta \). \( E\theta \) is called an instance of \( E \). Two expressions \( E \) and \( F \) are variants of each other, if one is an instance of the other. A variant of an expression is called fresh, if it only contains variables that do not occur elsewhere.

Two expressions \( E_1, E_2 \) are unifiable, if there is a substitution \( \theta \) such that \( E_1\theta = E_2\theta \). The substitution is then called a unifier of \( E_1 \) and \( E_2 \). The problem of finding a unifier of two expressions is called unification.

The relation between substitution and syntactic equality \( = \) can be expressed by the following classical theorem as formulated by Frühwirth:

**Theorem 1 (Substitutions and Syntactic Equality [Frü09, p. 50]).** Given two expressions \( A \) and \( B \) and a substitution \( \theta := \{ X_1/t_1, \ldots, X_n/t_n \} \), then

\[
\forall \theta. A = B\theta \leftrightarrow X_1=t_1 \land \ldots \land X_n=t_n.
\]

3.1.2 Definition of the Syntax

Constraints are distinguished predicates. They are the atomic data structure of CHR. The first-order language CHR divides the set of predicate symbols \( \Pi \) into two disjoint sets of CHR (user-defined) constraint symbols \( \Pi_u \) and built-in constraint symbols \( \Pi_b \). For a constraint symbol \( c \in \Pi \) and terms \( t_i (1 \leq i \leq n) \), the predicate \( c(t_1, \ldots, t_n) \) is a CHR constraint, if \( c \) is a CHR constraint symbol, and a built-in constraint otherwise. The set of all user-defined constraints in all rules of a CHR program \( P \) is denoted as \( C(P) \). The set of built-in constraint symbols contains at least the logical truth \( \top /0 \), logical falsity \( \bot /0 \) and syntactic equality \( \doteq /2 \). [Rai10, p. 8]

**Definition 5 (CHR Programs and Rules [Rai10, p. 8]).** A CHR program \( P \) is a finite set of CHR rules of the form

\[
r : H_k \setminus H_r \leftrightarrow G | B,
\]

where

- \( r \) is an optional rule name,
- the head \( H := H_k \cup H_r \) consists of two multi-sets of CHR constraints (the kept head \( H_k \) and the removed head \( H_r \)),
- the guard \( G \) is a conjunction of built-in constraints, and
- the body \( B := B_c, B_b \) consists of a multi-set of CHR constraints \( B_c \) and a conjunction of built-in constraints \( B_b \).
Depending on the head, there are three types of CHR rules with different notations:

**Simplification rules** If \( H_k \neq \emptyset \land H_r \neq \emptyset \), we write \( H_k \setminus H_r \Leftrightarrow G \mid B \).

**Simplification rules** If \( H_k = \emptyset \), we write \( H_r \Leftrightarrow G \mid B \).

**Propagation rules** If \( H_r = \emptyset \), we write \( H_k \Rightarrow G \mid B \).

The guard \( G \) and the \( \mid \) character may be omitted, if \( G = \top \).

Variables that occur in the rule, but not in its head, are called local variables. A rule without local variables is called range-restricted and a CHR program is called range-restricted if all its rules are range-restricted.

### 3.1.3 Notational Remarks

The syntax of CHR rules in Definition 5 is defined over multi-sets of CHR constraints and conjunctions of built-in constraints. Furthermore the constraints in the body of a rule are sorted by built-in constraints and user-defined constraints. We made these choices to simplify the notational overhead in many definitions by making use of (multi-)set operations.

However, in the literature, a different notation is found, where the heads are defined as conjunctions of CHR constraints and the body is a conjunction of CHR constraints and built-in constraints that can be mixed freely. Frühwirth argues that the conjunction of constraints comes from the close relation of CHR to logic and may be understood purely syntactically as it is “interpreted as logical operator, multi-set, or sequence forming operator” [Früh9, p. 55].

Therefore, we simplify notation by liberally omitting the curly brackets for multi-sets and instead write all elements separated by commas. We also mix built-in constraints and user-defined constraints in the body where it does not make a semantic difference, but simplifies the understanding of the rule. We omit the built-in constraints in the body of the rule, if they are the empty conjunction \( \top \).

### 3.2 Operational Semantics

In this section, we define the operational semantics of CHR as a state transition system. The operational semantics describes formally how a program is executed. For this purpose, we first define the notions of constraint theories, states and state equivalence in Section 3.2.1.

There exist various definitions of the operational semantics of CHR and each has its own purpose. In this thesis, we only present the semantics that are necessary to understand the main contributions. Additionally, CHR also has a declarative semantics. It relates a program with a logical theory, i.e. defines the logical reading of a CHR program.
An overview of the history and development of CHR’s different (operational and declarative) semantics can be found in [Frü09, pp. 55–82; Sne+10b].

The original definition of the operational semantics was presented by Frühwirth in 1995 [Frü95]. The most recent form of this semantics is the so-called very abstract operational semantics $\omega_{va}$ that is summarized in Section 3.2.2. It has the highest degree of abstraction and plays an important role in program analysis. By introducing the equivalence-based operational semantics $\omega_e$ described in Section 3.2.3, Raiser et al. have simplified the representation of $\omega_{va}$ and made it more suitable to analysis using the state equivalence relation [RBF09; Rai10, pp. 41–44; Bet14, pp. 37–46]. This operational semantics $\omega_e$ will be used throughout the thesis because of its advantages in program analysis.

There are some other important operational semantics for CHR that are described in Section 3.2.4. The two most influential semantics reflect the behavior of actual CHR implementations with increasing level of detail: the abstract operational semantics $\omega_l$ and the refined operational semantics $\omega_r$.

The most important declarative semantics relates CHR to first-order logic [AFM99; Frü09, pp. 69–75]. There also is a more recent declarative semantics based on linear logic [BF05; Bet14, pp. 73–85; Frü09, pp. 75–81]. We refer the interested reader to those references, since the declarative semantics are not necessary for the understanding of the thesis.

3.2.1 Preliminaries

In this section we give a brief introduction into three concepts that are required to understand the definitions of operational semantics of CHR: constraint theories, states and state equivalence.

3.2.1.1 Constraint Systems and Constraint Theories

The operational semantics of CHR is based on a constraint system. We summarize the basic definitions from [Frü09, p. 51]. A constraint system consists of constraint symbols, a set of values called the domain, and a constraint theory $\mathcal{CT}$ that describes the constraints. $\mathcal{CT}$ must be nonempty, consistent and complete. The latter means that for every constraint $c$ either $\mathcal{CT} \models c$ or $\mathcal{CT} \models \neg c$ holds. A more detailed introduction to constraint systems, their properties and capabilities can be found in [FA03, pp. 53–62]. Following the definition of the set of built-in constraints $\Pi_b$ in Section 3.1.2, $\mathcal{CT}$ must include an axiomatization of $\top$, $\bot$ and syntactic equality $\models$. 
3.2.1.2 CHR States

As the operational semantics of CHR is defined as a state transition system, the notion of states is defined in this section. There are many different notations to describe CHR states in the literature. The simplest ones describe them as goals, i.e. conjunctions of user-defined and built-in constraints [Frü09, p. 55]. Throughout the thesis, we use the following recent notation. Note that its components can be found in most other state notations.

**Definition 6 (CHR State [RBF09, p. 3; Rai10, pp. 33–34; Bet14, pp. 38–39]).** A CHR state is a tuple \( \langle G; B; V \rangle \) where the goal \( G \) is a multi-set of constraints, the built-in constraint store \( B \) is a conjunction of built-in constraints and \( V \) is a set of global variables. All variables occurring in a state that are not global are called local variables and the local variables that only appear in the built-in constraint store \( B \) are called strictly local variables.

The set of CHR states is denoted by \( \Sigma_{\text{CHR}} \). The empty state is defined as \( \sigma_\emptyset := \langle \emptyset; \top; \emptyset \rangle \).

3.2.1.3 State Equivalence

The semantics of CHR relies on a definition of state equivalence. Although this is such a crucial part of the language, there have been various (mostly ad-hoc) definitions of state equivalence that all differ in some regards [e.g. AFM99; FDW02; DSS07; Frü09, pp. 71–72].

In [RBF09; Rai10, pp. 33–41; Bet14, pp. 40–44] the differences of the various definitions are discussed in detail and the problems of each definition are pointed out. As a result, an axiomatic definition of state equivalence is given that satisfies the desired properties for CHR state equivalence. For this reason, we use the definition of [RBF09; Rai10; Bet14] throughout the thesis and give a summary of the definition in this section.

**Definition 7 (State Equivalence of CHR States).** Equivalence between CHR states is the smallest equivalence relation \( \equiv \) over CHR states that satisfies the following conditions:

1. **Equality as Substitution**

   \[
   \langle G; X \leftarrow t \land B; V \rangle \equiv \langle G[X/t]; X \leftarrow t \land B; V \rangle. 
   \]

2. **Transformation of the Constraint Store**

   \[
   \langle G; B; V \rangle \equiv \langle G; B'; V \rangle, 
   \]

   if \( CT \models \exists \bar{s}.B \leftrightarrow \exists \bar{s}'.B' \), where \( \bar{s}, \bar{s}' \) are the strictly local variables of \( B, B' \), respectively.
3. OMISSION OF NON-OCCURRING GLOBAL VARIABLES

\[ \langle G; B; \{X\} \cup V \rangle \equiv \langle G; B; V \rangle, \]

if \(X\) is a variable that does not occur in \(G\) or \(B\).

4. EQUIVALENCE OF FAILED STATES

\[ \langle G; \bot; V \rangle \equiv \langle G'; \bot; V' \rangle. \]

The following lemma summarizes some properties of the state equivalence relation.

**Lemma 2** (Properties of State Equivalence [Rai10, p. 37]). The state equivalence relation \(\equiv\) from Definition 7 has the following properties:

1. **Renaming of Local Variables** Let \(X, Y\) be variables such that \(X, Y \notin V\) and \(Y\) does not occur in \(G\) or \(B\):

\[ \langle G; B; V \rangle \equiv \langle G[X/Y]; B[X/Y]; V \rangle \]

2. **Partial Substitution** Let \(G[X \bowtie t]\) be a multi-set where some occurrences of \(X\) are substituted with \(t\):

\[ \langle G; X \bowtie t \land B; V \rangle \equiv \langle G[X \bowtie t]; X \bowtie t \land B; V \rangle \]

3. **Logical Equivalence** If

\[ \langle G; B; V \rangle \equiv \langle G'; B'; V' \rangle \]

then \(C^T \models \exists \bar{y}. G \land B \iff \exists \bar{y}'. G' \land B'\), where \(\bar{y}, \bar{y}'\) are the local variables of \(\langle G; B; V \rangle\) and \(\langle G'; B'; V' \rangle\), respectively.

The first property allows to rename local variables with fresh names. The second property allows to apply some of the variable bindings to the goal store and the third property denotes that equivalent states have an equivalent logical reading.

**Example 2** (State Equivalence [Rai10, p. 34]). By the above definition of state equivalence, the following states are equivalent:

- \(\langle c(X); \top; \emptyset \rangle \equiv \langle c(Y); \top; \emptyset \rangle\), i.e. local variables can be renamed.
- \(\langle c(X); X \bowtie 0; \{X\} \rangle \equiv \langle c(0); X \bowtie 0; \{X\} \rangle\), i.e. variable bindings from the built-in store can be applied to the goal store.
- \(\langle \emptyset; X \bowtie Y \land Y \bowtie 0; \emptyset \rangle \equiv \langle \emptyset; X \bowtie 0 \land Y \bowtie 0; \emptyset \rangle\), i.e. equivalent built-in stores can be interchanged.

This includes that \(\langle \emptyset; X \bowtie 0; \emptyset \rangle \equiv \langle \emptyset; \top; \emptyset \rangle\), since \(\exists X. X \bowtie 0\) is a tautology.
• \(\langle c(0); \top; \{X\} \rangle \equiv \langle c(0); \top; \emptyset \rangle\), i.e. unused global variables can be omitted.

However, \(\langle c(0); \top; \emptyset \rangle \not\equiv \langle c(0); X^=1; \emptyset \rangle\), since \(X\) appears in the built-in constraint store as a strictly local variable. By renaming the local variable, it is possible to introduce the unnecessary global variable: \(\langle c(0); X^=1; \emptyset \rangle \equiv \langle c(0); Y^=1; \emptyset \rangle \equiv \langle c(0); Y^=1; \{X\} \rangle\).

• \(\langle c(X); \top; \{X\} \rangle \not\equiv \langle c(Y); \top; \{Y\} \rangle\), i.e. \(X\) and \(Y\) are free variables and therefore the logical readings of the states are different.

Given the axiomatic definition of state equivalence (c.f. Definition 7) that defines the desired properties of CHR state equivalence, it is of interest how state equivalence of two states can be decided. Raiser et al. propose a sufficient and necessary criterion for deciding state equivalence according to Definition 7 that is suitable for implementation in the following theorem:

**Theorem 2** (State Equivalence Criterion [RBF09, pp. 8–9; Rai10, pp. 38–40]). Let \(\sigma := \langle G; B; V \rangle\) and \(\sigma' := \langle G'; B'; V \rangle\) be two CHR states with local variables \(\bar{y}_1, \bar{y}_2\) that have been renamed apart.

\[\sigma \equiv \sigma'\]

if and only if

\[\mathcal{C}T \models \forall(B \rightarrow \exists \bar{y}_2.((G^=G') \land B')) \land \forall(B' \rightarrow \exists \bar{y}_1.((G^=G') \land B)),\]

where \(\forall F\) is the universal closure of formula \(F\) and \(\equiv\) is syntactic equivalence.

Note that the criterion requires the two states to have equivalent global variables \(V\). For states with different global variables, the **Omission of Non-Occurring Global Variables** axiom from Definition 7 can be used to equate the global variables of both states.

### 3.2.2 Very Abstract Semantics \(\omega_{va}\)

The most concise and simple operational semantics of CHR is the **very abstract operational semantics** \(\omega_{va}\). It goes back to the earliest presentations of CHR [Frü91; Frü92]. The very abstract semantics also is the most general formulation of the semantics in the sense that every computation in the more concrete semantics and implementations are also possible in the very abstract semantics [Frü09, pp. 81–82].

**Definition 8** (Very Abstract Operational Semantics \(\omega_{va}\) [RBF09, p. 10]). For a rule \(r\), the variables appearing in the rule are called local variables. A variant of a rule is a copy of a rule where a subset of its local variables has been renamed.
For a CHR program $P$, the state transition system over CHR states and the rule transition relation $\mapsto_{\text{va}}$ is defined as the following transition scheme:

$$r : H_k \setminus H_t \leftrightarrow G \mid B_c, B_b \in P \text{ with fresh variables } \bar{x}$$

$$C^T \models \forall (B \rightarrow \exists \bar{x} . (H_k \setminus H'_k \land H_t \setminus H'_t \land G))$$

$$\mapsto_{\text{va}} (H'_k \cup H'_t \cup G ; G \land B ; V)$$

$$\mapsto_{\text{va}} (H'_k \cup B \cup G ; H_k \setminus H'_k \land H_t \setminus H'_t \land G \land B_b \land B ; V)$$

We also write $\mapsto_{\text{va}}$, if the rule $r$ is clear from the context.

The global variables $V$ do not play a role in the operational semantics, but only for state equivalence. Hence, in the literature, they often are omitted in the state tuple. The original description of $\omega_{\text{va}}$ even syntactically represents states as goals, i.e. conjunctions of user-defined and built-in constraints $G \land B$ [Fru09, p. 55].

**Example 3 (\(\omega_{\text{va}}\) Transitions).** In CHR, arrays can be represented as a sequence of constraints of the form $a(\text{Index}, \text{Value})$, i.e. an array of length 3 is represented as

$$a(0, A_0), a(1, A_1), a(2, A_2),$$

where $A_i (1 \leq i \leq 3)$ are the values of the array. The following program sorts an array by exchanging pairs of values that are in the wrong order:

$$\text{ex} : a(I, V), a(J, W) \leftrightarrow I > J, V < W \land a(I, W), a(J, V).$$

Consider the following example derivation:

$$\langle a(0, 1), a(1, 7), a(2, 5), a(3, 6); \top; \emptyset \rangle$$

$$\mapsto_{\text{va}}^{\text{ex}} \langle a(0, 1), a(J, V), a(I, W), a(3, 6);$$

$$I \equiv 2 \land J \equiv 1 \land V \equiv 5 \land W \equiv 7 \land I > J \land V < W; \emptyset \rangle$$

$$\mapsto_{\text{va}}^{\text{ex}} \langle a(0, 1), a(J, V), a(J', V'), a(I', W');$$

$$I \equiv 2 \land J \equiv 1 \land V \equiv 5 \land W \equiv 7 \land I > J \land V < W \land$$

$$I' \equiv 3 \land J' \equiv I \land V' \equiv 6 \land W' \equiv W \land I' > J' \land V' < W'; \emptyset \rangle.$$
Note that this is not the only possible derivation:

\[ \langle a(0,1), a(1,7), a(2,5), a(3,6); \top; \emptyset \rangle \xrightarrow{ex} \langle a(0,1), a(1,6), a(2,5), a(3,7); \top; \emptyset \rangle \]  
\[ \xrightarrow{ex} \langle a(0,1), a(1,5), a(2,6), a(3,7); \top; \emptyset \rangle. \]

Note that in this derivation we have implicitly applied the state equivalence relation to simplify the store, although technically \( \omega_{va} \) does not allow for such a simplification. Although two different derivations have been chosen, the final states are equivalent.

**MATCHING AND UNIFICATION** Instead of unification, which is e.g. used in Prolog, the very abstract semantics uses matching to find applicable rules.

**Definition 9** (Matching [Bür89, p. 523]). Matching can be defined on top of substitutions in two ways:

1. A substitution \( \theta \) matches a term \( t \) to a term \( s \), if and only if \( s = t \theta \).

2. A substitution \( \theta \) matches a term \( t \) to a term \( s \), if and only if \( s = s \theta = t \theta \).

We also say that the instance \( s \) matches the pattern \( t \) with matching substitution \( \theta \).

The definition means that \( t \) serves as a pattern for a term \( s \), if there is a substitution \( \theta \) that replaces the variables in \( t \) such that \( s \) is obtained. This means that only the variables of the pattern are allowed to be replaced to unify the terms, not the variables of the (more specific) term \( s \).

The second definition makes a further restriction on which variables are allowed to substitute into. For instance, the substitution \( \{ X / g(X) \} \) is a matcher of \( f(X) \) to \( f(g(X)) \) in the first definition, whereas it is not in the second [Bür89, p. 523]. We will see that for the operational semantics of CHR it does not play a role which of the different definitions of matching is chosen and hence the first way is typically found in the literature [Frü09, p. 50].

According to [Bür89, p. 525], matching can be expressed by unification in the following way: In an equational theory \( E \), matching the terms \( t_1, \ldots, t_n \) to the terms \( s_1, \ldots, s_n \) is equivalent to finding assignments for the existentially quantified variables in

\[ E \models \forall \bar{y} \exists \bar{x}. (s_1 = t_1 \land \ldots \land s_n = t_n), \]

where \( \bar{g} \) denotes the sequence of all variables (or some subset of the variables) in \( s_1, \ldots, s_n \) and \( \bar{x} \) denotes the sequence of the remaining variables of \( t_1, \ldots, t_n \). Due to this definition, matching is sometimes referred to as **one-sided unification**. Note that here the symbol \( = \) refers
The applicability condition in the operational semantics of CHR reflects matching.

Variable bindings from the matching and other built-in constraints are accumulated in the state.

to the respective equality relation of the equational theory $E$ (usually syntactic equality $\equiv$ of Clark’s Equality Theory [Cl78; JLM84, pp. 218–220; FA03, p. 20]).

This is exactly what is expressed in the applicability condition in Definition 8: We are looking for assignments of the variables of the rule $\bar{x} \in \text{vars}(r)$, such that the guard together with equality constraints of the form $X \equiv X'$ is implied by the built-in store. This means that only the variables of the rule can be bound, but not the variables of the state. I.e. there are constraints in the store that match the head of the rule (and satisfy the guard). Hence, the rule serves as a pattern for the constraints in the store and the constraints in the store are instances of the rule head. Since the variables in the rule are replaced by fresh variants that do not appear in the state, it does not matter which one of the two definitions of matching in Definition 9 is chosen.

Note that the bindings of the matching are added to the built-in store (as the rest of the guard) when the rule is applied. While user-defined constraints can be added and removed from the store, built-in constraints are always added to the state and remain there. Hence, the built-in constraints monotonically accumulate information [Frü09, p. 57].

**Example 4 (Matching).** Consider the following expression as a pattern:

$$c(X, 5)$$

The pattern is matched by the following instances:

- $c(1, 5)$ with substitution $\{X/1\}$,
- $c(Y, 5)$ with substitution $\{X/Y\}$.

However, the following expressions do not match the pattern $c(X, 5)$, i.e. they are no instances:

- $c(1, Y)$,
- $c(Y, Z)$.

The reason is that here the variables $Y$ and $Z$ would have to be substituted by 5, which is not allowed (one-sided unification).

**Trivial Non-Termination** The very abstract semantics is the most general formulation of CHR’s behavior. The selection of rules $\alpha$ and the choice of matching constraints from the goal store is non-deterministic. This high level approach leads to trivial non-termination with propagation rules.

**Example 5 (Trivial Non-Termination of $\omega_{\alpha}$).** Consider the rule $a \Rightarrow b$. The following derivation is possible:

$$\langle a; \top; \emptyset \rangle \mapsto \langle a, b; \top; \emptyset \rangle \mapsto \langle a, b, b; \top; \emptyset \rangle \mapsto \ldots$$
Due to the non-determinism of ωva this is no problem in theory, since one can always apply the rule only as often as needed. However, in practice (and also for program analysis), trivial non-termination usually is undesired. Section 3.2.4 discusses some extensions that try to avoid trivial non-termination of propagation rules.

**Remarks on differing formulations**

The formulation of the very abstract operational semantics in Definition 8 differs slightly from the most recent definition in [Früo9, p. 56], where rules are required to be in the so-called head normal form (HNF). The HNF only allows variables to appear as arguments in the head that are then bound in the guard.

**Definition 10 (Head Normal Form of a CHR Rule [Früo9, p. 56]).** A CHR rule r : Hk \ Hr ⇔ G | Bc, Bb ∈ P with fresh variables x

\[ CT \models \forall (B \rightarrow \exists x.G) \]

\[ \langle H_k \cup H_r \cup G; G \wedge B_c \wedge \exists V \rangle \rightarrow^*_{va} \langle H_k \cup B_c \cup G; G \wedge B_b \wedge B_c \wedge \exists V \rangle \]

However, this definition in [Früo9, p. 56] is not entirely correct: If there appear constraints with non-variable arguments in the goal store, the rules in the transition scheme do not match and are therefore not applicable, which was certainly not intended.

**Example 6.** Consider the exchange sort example from Example 3:

\[ ex : a(I, V), a(J, W) \Leftrightarrow I>J, V<W | a(I, W), a(J, V) \].

In the example state \( \langle a(0, 1), a(1, 7), a(2, 5), a(3, 6); \top; \emptyset \rangle \) no derivation is possible, since there is no exact representation of a variant of the head in the goal store. We first have to obtain the equivalent state

\[ \langle a(0, A_0), a(1, A_1), a(2, A_2), a(3, A_3); A_0=1 \wedge A_1=7 \wedge A_2=5 \wedge A_3=6; \top; \emptyset \rangle \]

\[ \equiv \langle a(0, 1), a(1, 7), a(2, 5), a(3, 6); \top; \emptyset \rangle. \]

Then the same derivations as in Example 3 are possible.
28 CONSTRAINT HANDLING RULES

We therefore use the formulation from [RBF09, p. 10] in Definition 8. It is a mix of the very similar definitions in [Frü09, p. 56] and [FA03, p. 43] and thoroughly captures the intention of [Frü09].

The problem can be fixed by requiring states to be in a corresponding normal form, where variable bindings are moved to the built-in store. The state equivalence relation from Definition 7 allows to do so. Another solution is to introduce a transition that allows to apply rules in equivalent states. Those considerations ultimately lead to the definition of the equivalence-based operational semantics $\omega_e$ in Section 3.2.3.

3.2.3 Equivalence-Based Semantics $\omega_e$

The very abstract semantics in its pure form as proposed in [Frü09, p. 56] and in Definition 8 requires the constraints to appear in the state exactly as they are in the rule head; variable bindings cannot be applied directly to the user-defined constraints. In general, the semantics does not define a transition that simplifies the built-in store, although such definitions may be found in the literature [e.g. FA03, p. 42]. Nevertheless, this is done very often in practice to simplify the visual representation of the states (c.f. e.g. Examples 3 and 6).

In this section, it is shown that equivalent states are indistinguishable for the transition system $\omega_{va}$, i.e. that we can always use equivalent states to proceed our computations. This leads to the definition of the equivalence-based operational semantics $\omega_e$ from recent work [RBF09; Rai10, pp. 41–44; Bet14, pp. 38–44] that simplifies $\omega_{va}$. It is then shown that the formulations of $\omega_e$ and $\omega_{va}$ are equivalent and can therefore be interchanged. Because of its advantages in program analysis, the equivalence-based operational semantics $\omega_e$ is used in the main parts of this thesis.

We start with the definition of the transition system.

**Definition 12** (Operational Semantics $\omega_e$ [RBF09, p. 10; Rai10, pp. 41–42; Bet14, pp. 43–44]). For a CHR program $P$ we define the state transition system $(\Sigma_{CHR}, \mapsto_e)$ referred to as $\omega_e$ as follows:

$$r : H_k \setminus H_r \Leftrightarrow G \mid B_c, B_b$$

$$\langle H_k \cup H_r \cup G; G \land C; V \rangle \mapsto_e \langle H_k \cup B_c \uplus G; G \land B_b \land C; V \rangle$$

$$\sigma' \equiv \sigma \quad \sigma \mapsto_e \tau \quad \tau \equiv \tau'$$

$$\sigma' \mapsto_e \tau'$$

Thereby, $r$ is a variant of a rule in the program such that its local variables are disjoint from the variables occurring in the pre-transition state. We may just write $\mapsto_e$ instead of $\mapsto_e$ if the rule $r$ is clear from the context. Since $\omega_e$ is the operational semantics of CHR used throughout the rest of this thesis, we may also just write $\mapsto$. For a CHR program $P$, $\mapsto_P$ denotes the state
transition relation of \( \mathcal{P} \). If the rule or program are clear from the context, we simply write \( \mapsto \).

**Example 7** (\( \omega_e \) Transitions). Reconsider the exchange sort rule from Example 3:

\[
\text{ex : } a(I, V), a(J, W) \leftrightarrow I > J, V < W \ | \ a(I, W), a(J, V).
\]

The example derivation from Example 3 is also possible in \( \omega_e \):

\[
\begin{align*}
\langle a(0, 1), a(1, 7), a(2, 5), a(3, 6); \top; \emptyset \rangle \\
\equiv \langle a(0, 1), a(J, W), a(I, V), a(3, 6); \\
I \models 2 \land J \models 1 \land V \models 5 \land W \models 7; \emptyset \rangle \\
\mapsto_{\text{ex}}^e \langle a(0, 1), a(J, V), a(I, W), a(3, 6); \\
I \models 2 \land J \models 1 \land V \models 5 \land W \models 7 \land I > J \land V < W; \emptyset \rangle \\
\equiv \langle a(0, 1), a(1, 5), a(2, 7), a(3, 6); \top; \emptyset \rangle \\
\equiv \langle a(0, 1), a(1, 5), a(J, W), a(I, V); \\
I \models 3 \land J \models 2 \land V \models 6 \land W \models 7; \emptyset \rangle \\
\mapsto_{\text{ex}}^e \langle a(0, 1), a(1, 5), a(J, V), a(I, W); \\
I \models 3 \land J \models 2 \land V \models 6 \land W \models 7 \land I > J \land V < W; \emptyset \rangle \\
\equiv \langle a(0, 1), a(1, 5), a(2, 6), a(3, 7); \top; \emptyset \rangle.
\end{align*}
\]

To obtain the equivalent states, the axiom Equality as Substitution from Definition 7 is used to apply the variable bindings in the goal store, i.e. replace the variables by their actual values. This makes all the variables strictly local, since they only appear in the built-in store now. The axiom Transformation of the Constraint Store allows to reduce tautologies of the form \( \exists I, J, (I \models 2 \land J \models 1 \land I > J) \) to \( \top \) (since obviously there exist values for \( I \) and \( J \) that satisfy the properties – namely 2 and 1).

In the example, it can be seen that the state first has to be transformed to an equivalent state that includes the rule head exactly as is, in contrast to finding a matching substitution for head and state directly in the rule application (as in Definition 8 of \( \omega_{va} \)). Furthermore, the syntactic representation of the state can be simplified by solving the built-in store according to Definition 7. This is an advantage compared to the canonical definition of \( \omega_{va} \).

In the next steps, we will see that \( \omega_{va} \) and \( \omega_e \) are equivalent modulo state equivalence and that state equivalence is semantically compliant with both semantics. This lifts the habit of regarding CHR states modulo state equivalence that is often found in practical CHR examples, where built-in stores are usually simplified in an ad-hoc manner, on a formal basis and allows for a thorough simplification of program analysis.

**Theorem 3** (Equivalence of \( \omega_{va} \) and \( \omega_e \) [RBFO9, p. 11; RA10, p. 43; BET14, p. 44]). For any CHR state \( \sigma \in \Sigma_{\text{CHR}} \) the following two propositions hold:`
State equivalence is semantically compliant with $\omega_{va}$ and $\omega_{e}$.

The semantics can be defined over equivalence classes of states.

Theorem 3 shows that $\omega_{e}$ and $\omega_{va}$ are equivalent modulo state equivalence, i.e., transitions from equivalent states in both semantics lead to equivalent states. This also means that trivial non-termination with propagation rules (c.f. Section 3.2.2 and Example 5 on Page 26) is inherited from $\omega_{va}$.

The following theorem ensures semantic compliance of state equivalence with $\omega_{e}$ and also $\omega_{va}$. It follows from Theorem 3.

**Theorem 4** (Semantic Compliance [Bet14, p. 44]). The state equivalence relation $\equiv$ is semantically compliant with the operational semantics $\omega_{e}$ and $\omega_{va}$. In particular, this means that the following properties hold:

1. For states $\sigma, \sigma', \tau$ such that $\sigma \equiv \sigma'$ and $\sigma \rightsquigarrow_{va} \tau$, there exists a state $\tau'$ such that $\sigma' \rightsquigarrow_{va} \tau'$ and $\tau \equiv \tau'$.
2. For states $\sigma, \sigma', \tau$ such that $\sigma \equiv \sigma'$ and $\sigma \rightsquigarrow_{e} \tau$, there exists a state $\tau'$ such that $\sigma' \rightsquigarrow_{e} \tau'$ and $\tau \equiv \tau'$.

The semantic compliance theorem allows us to consider both operational semantics modulo the state equivalence relation, i.e., all equivalent states behave alike with respect to the operational semantics. Hence, it does not matter which syntactical representation of two equivalent states is chosen, they both lead to equivalent results (in both semantics). This allows us to consider the operational semantics over equivalence classes of states. An equivalence class of a CHR state is defined as

$$\left[\sigma\right] := \{\sigma' \mid \sigma' \equiv \sigma\}.$$

Due to semantic compliance, equivalence classes and states can be interchanged freely. Note that to signify state equivalence of two states we use $\equiv$. To signify that two representatives denote the same equivalence class (i.e., are equivalent), we use $=\equiv$.

From semantic compliance it follows that the state transition system can be defined over the set of equivalence classes $\Sigma_{CHR}/\equiv$ (also called the quotient set of the set of CHR states $\Sigma_{CHR}$ by state equivalence $\equiv$ or $\Sigma_{CHR}$ modulo state equivalence $\equiv$).

**Definition 13** (Operational Semantics $\omega_{e}$ Modulo State Equivalence [RBF09, p. 12; Rai10, pp. 43–44; Bet14, p. 44]). For a CHR program $P$ we define the state transition system $(\Sigma_{CHR}/\equiv, \rightsquigarrow_{e})$, referred to as $\omega_{e}$ as follows:

$$r : H_k \setminus H_r \leftrightarrow G \mid B_c, B_b \quad \left[\langle H_k \cup H_r \cup G; G \land C; V \rangle\right] \rightsquigarrow_{e} \left[\langle H_k \cup B_c \cup G; G \land B_b \land C; V \rangle\right]$$
Thereby, \( r \) is a variant of a rule in the program such that its local variables are disjoint from the variables occurring in the representative of the pre-transition state. Again as in Definition 12, we may just write \( \rightarrow \omega \) instead of \( \rightarrow^\prime \omega \) if the rule \( r \) is clear from the context or \( \rightarrow^\prime \) and \( \rightarrow \), respectively.

**Example 8** (\( \omega_\epsilon \) Transitions Modulo State Equivalence). The derivations from Example 7 can be applied directly to equivalence classes:

\[
([\langle 0, 1 \rangle, \langle 1, 7 \rangle, \langle 2, 5 \rangle, \langle 3, 6 \rangle; \top; \emptyset])
\]

\[
= ([\langle 0, 1 \rangle, \langle I, W \rangle, \langle I, V \rangle, \langle 3, 6 \rangle; I \triangleq 2 \land J \triangleq 1 \land V \triangleq 5 \land W \triangleq 7; \emptyset])
\]

\[
\rightarrow^\epsilon \omega \; ([\langle 0, 1 \rangle, \langle I, V \rangle, \langle I, W \rangle, \langle 3, 6 \rangle; I \triangleq 2 \land J \triangleq 1 \land V \triangleq 5 \land W \triangleq 7 \land I > J \land V < W; \emptyset])
\]

\[
= ([\langle 0, 1 \rangle, \langle 1, 5 \rangle, \langle 2, 7 \rangle, \langle 3, 6 \rangle; \top; \emptyset])
\]

\[
= ([\langle 0, 1 \rangle, \langle 1, 5 \rangle, \langle J, W \rangle, \langle I, V \rangle; I \triangleq 3 \land J \triangleq 2 \land V \triangleq 6 \land W \triangleq 7; \emptyset])
\]

\[
\rightarrow^\epsilon \omega \; ([\langle 0, 1 \rangle, \langle 1, 5 \rangle, \langle J, V \rangle, \langle I, W \rangle; I \triangleq 3 \land J \triangleq 2 \land V \triangleq 6 \land W \triangleq 7 \land I > J \land V < W; \emptyset])
\]

\[
= ([\langle 0, 1 \rangle, \langle 1, 5 \rangle, \langle 2, 6 \rangle, \langle 3, 7 \rangle; \top; \emptyset]).
\]


The semantic compliance of state equivalence and the operational semantics shows that state equivalence is an integral part of CHR. The formulation of the transition system over equivalence classes makes this fact explicit. It takes care of grouping different representations of semantically equivalent states together to an equivalence class. Hence, one does not have to care about the particular syntactic representation of a state. This makes \( \omega_\epsilon \) modulo state equivalence suitable for program analysis of CHR without losing expressiveness compared to \( \omega_\va \).

### 3.2.4 Other Operational Semantics

There are some other operational semantics for CHR that are all related to some degree to the original semantics \( \omega_\va \). The purpose of this section is to give a brief overview of how the basic CHR semantics can be extended or adapted. We are confident that our work can be transferred to some of those semantics.

**Theoretical Semantics** (\( \omega_\theta \)) This semantics is also called the abstract operational semantics. It is an extension of \( \omega_\va \) that is still suitable for analysis, but covers the behavior of actual CHR implementations more thoroughly.

In \( \omega_\va \) and also \( \omega_\epsilon \), the introduction of a propagation rule to a program leads to trivial non-termination (c.f. Section 3.2.2 and Example 5 on Page 26). The theoretical operational semantics
\( \omega_t \) extends \( \omega_{va} \) by a propagation history (also called a token store) to fix trivial non-termination caused by propagation rules. Basically, the constraints in the goal store are indexed by a unique identifier. The propagation history keeps track of which has been applied to which combination of constraints by using the unique identifiers. The semantics ensures that every (propagation) rule is only applied once for the same combination of constraints. A detailed description of the theoretical operational semantics can be found in \cite{Abd97, Frü09, pp. 59–64].

\( \omega_t \) is sound, i.e., every derivation is also possible in \( \omega_{va} \) (and therefore in \( \omega_e \)). However, it lacks completeness, since not every \( \omega_{va} \) or \( \omega_e \) computation is possible in \( \omega_t \) \cite{Rai10, p. 48].

**Refined Semantics (\( \omega_r \))** The theoretical operational semantics \( \omega_t \) allows for non-deterministic derivations since it does not define the order in which constraints of a goal are processed and the order in which rules are applied. To formally define the behavior of implementations, the refined operational semantics has been introduced as an extension of \( \omega_t \) \cite{Duc+04, Frü09, pp. 64–69]. It also uses a propagation history to avoid trivial non-termination and additionally defines that rules are applied top-down in textual order and goals are executed from left to right.

For this purpose, the traditional goal store or user-defined constraint store is divided into a goal store that contains the so-called active constraints and a CHR store that contains all other constraints. The goal store behaves like a stack, where the constraint on top is called the active constraint. When the top constraint of the goal is activated, a matching rule is searched. When the rule is applied, the active constraint is either moved to the CHR store (if it appears in the removed head of the rule) or kept active (if it appears in the kept head of the rule, e.g., in propagation rules). The constraints that are introduced by the body are added on top of the goal stack, which leads to a depth-first execution model.

Note that the refined semantics does not define the matching order in which partner constraints for the active constraints are selected. This behavior may differ from implementation to implementation. The reason for this source of non-determinism is that program flow would be almost unpredictable or at least difficult to follow if developers actually rely on a specific matching order.

The refined semantics is sound with respect to \( \omega_t \), i.e., every computation in \( \omega_r \) is also possible in \( \omega_t \). However, by eliminating the two sources of non-determinism in \( \omega_t \), \( \omega_r \) is not complete with respect to \( \omega_t \).

**Persistent Constraints (\( \omega! \))** Introducing a propagation history to eliminate trivial non-termination as in \( \omega_t \) comes at the cost of
losing completeness with respect to $\omega_{va}$ and $\omega_t$. This gap is filled by the introduction of persistent constraints to the equivalence based semantics [BRF10b; BRF10a; Rai10, pp. 44–49]. The idea of persistent constraints comes from linear logic, where formulas represent resources that can be consumed rather than logical truths. To reestablish the power of classical logic, the $!$-operator marks a resource as *stable* or *unlimited*. The name $\omega_t$ for the operational semantics with persistent constraints is inspired by the $!$-operator.

Transferred to CHR, traditional user-defined constraints correspond to consumable resources, since constraints that appear in the removed head are actually removed from the constraint store. However, constraints that appear in the body of a propagation rule can be reproduced infinitely once the propagation rule was applicable. I.e., $\omega_t$ considers constraints that have been introduced by a propagation rule as unlimited resources – called *persistent constraints*. Such persistent constraints are not added to the classical goal store, but to the persistent constraint store. Constraints in the persistent constraint store cannot be removed.

Persistent constraints are a finite representation of an infinite amount of constraints. Intuitively, they represent the idea that propagation rules produce just as many instances of its constraints in the body (by repeated application) as are needed for further processing.

$\omega_t$ closes the gap between a semantics that avoids trivial non-termination of propagation rules (like $\omega_t$) without losing completeness to $\omega_{va}$ and $\omega_e$ (which is not the case for $\omega_t$). While $\omega_t$ only applies each propagation rule once (on identical constraints) and therefore misses some derivations possible in $\omega_e$, $\omega_e$ itself also allows for an infinite application of the propagation rules, i.e. non-termination. $\omega_t$ combines the best of both worlds, where all derivations of $\omega_e$ that do not include infinite application of a propagation rule (since this is represented by the unlimited resource of persistent constraints in the state, i.e. the state does not change by a repeated application of a propagation rule). Hence, $\omega_t$ is sound and complete with respect to $\omega_e$.

**Set-based semantics ($\omega_{\text{set}}$)** Another way of solving the trivial non-termination problem without the introduction of a propagation history is the set-based semantics $\omega_{\text{set}}$ [SR07]. The difference to typical CHR semantics is that multi-set rewriting is exchanged by set rewriting. Furthermore, the semantics includes efficient tabling.

**Equivalence-based token store semantics ($\omega_\tau$)** The idea of $\omega_\tau$ is to reduce the complexity of the theoretical semantics $\omega_t$. 

---

3.2 Operational Semantics 33
Therefore, the approach of $\omega_e$ to use state equivalence and equivalence classes is applied analogously. The semantics $\omega_\tau$ has been introduced in [Bet14, pp. 50–57]. It is shown to be equivalent to $\omega_1$ [Bet14, p. 56]. Hence, in the same way as $\omega_1$ is an extension of the very abstract semantics $\omega_{va}$, $\omega_\tau$ can be considered as the analogous extension of $\omega_e$.

In contrast to the set-based semantics or the semantics with persistent constraints $\omega_!$ that try to avoid non-termination of propagation rule without introducing a propagation history, $\omega_\tau$ keeps the popular and prevalent token store, but improves its formal representation and therefore improves its suitability for program analysis.

**Rule Priorities ($\omega_p$)** This semantics introduces rule priorities to manipulate execution control [DSD07; DSD08]. In contrast to the procedural approach of $\omega_r$, it still offers a high-level and declarative reading. Execution control is only affected by priority annotations leading to separation of logic and control that is desired in declarative programming languages. It is shown that the semantics can be implemented efficiently by optimizing compilers [DSD08]. Hence, $\omega_p$ can be considered a more declarative but still efficient alternative to $\omega_r$.

**Semantics with Disjunction (CHR $\lor$)** Traditional CHR semantics have a committed-choice nature that does not include any backtracking, i.e. a rule application is never made undone. CHR $\lor$ is an extension that allows for disjunctive rule bodies and therefore introduces backtracking over those alternatives [AS98].

**Probabilistic Semantics** The first approach to combine CHR with probabilistic rule selection has been introduced in [FDW02]. A more advanced probabilistic semantics has been obtained in CHRiSM [SMV09; Sne+10a] that combines CHR with the probabilistic Prolog dialect PRISM as host language.

### 3.3 Program Analysis

This section discusses some established program analysis techniques for CHR. The formalisms and methods introduced in this section are the foundation for our advanced analysis techniques introduced in Part ii.

**3.3.1 Merge Operator**

In program analysis, there is often the necessity to create a state by merging two states or splitting a state into two smaller states. Both can be achieved by the following definition of the merge operator.
Definition 14 (Merge Operator $\odot$ [Rai10, p. 50]). Let $\sigma_i = \langle G_i; B_i; V_i \rangle$ for $i = 1, 2$ be two CHR states such that local variables of one state are disjoint from all variables in the other state. Then for a set $\mathcal{V}$ of variables

$$\sigma_1 \odot_{\mathcal{V}} \sigma_2 := \langle G_1 \uplus G_2; B_1 \land B_2; (V_1 \cup \mathcal{V}) \setminus \mathcal{V} \rangle.$$ 

For equivalence classes of CHR states, the merging is defined as

$$[\sigma_1] \odot_{\mathcal{V}} [\sigma_2] := [\sigma_1 \odot_{\mathcal{V}} \sigma_2]$$

for two representatives of the equivalence class that have disjoint local variables. For $\mathcal{V} = \emptyset$ we write $[\sigma_1] \odot [\sigma_2]$.

The definition of the merge operator over equivalence classes is justified, since Lemma 2.1 guarantees that renaming local variables maintains state equivalence. Hence, it is guaranteed that the representatives with disjoint local variables needed for merging exists [Rai10, p. 50]. Additionally, $\odot_{\mathcal{V}}$ also maintains state equivalence, which further justifies the use of the merge operator over equivalence classes.

Lemma 3 ($\odot_{\mathcal{V}}$ Maintains State Equivalence [Rai10, p. 50]). Let $\sigma_1 \equiv \sigma_2$, then $(\sigma_1 \odot_{\mathcal{V}} \tau) \equiv (\sigma_2 \odot_{\mathcal{V}} \tau)$ for all $\mathcal{V}$.

As we will learn in Chapter 2, equivalence relations that are maintained by $\odot_{\mathcal{V}}$ are also called congruence relations with respect to $\odot_{\mathcal{V}}$ (c.f. Definitions 35 and 36).

The merge operator is often used in program analysis to extract contextual information from the state to divide it from its essential parts. However, this becomes a problem when local variables are used. Since local variables have to be disjoint when merging two states, it is not possible to extract information about them directly, as can be seen in the following example.

Example 9 (Extracting Contextual Information with $\odot_{\mathcal{V}}$ [Rai10, p. 50]). The state $\langle \langle c(X); X = 1; \emptyset \rangle \rangle$ can be considered as a version of $\langle \langle c(X); T; \emptyset \rangle \rangle$, where the local variable $X$ is bound to 1. In the state $\langle \langle c(X), X = 1; \emptyset \rangle \rangle$, we would consider $X = 1$ as contextual information about $X$.

However, it is not possible to extract this information by

$$\langle \langle c(X); T; \emptyset \rangle \rangle \odot \langle \langle \emptyset; X = 1; \emptyset \rangle \rangle,$$

since $\langle \langle \emptyset; X = 1; \emptyset \rangle \rangle = \langle \langle \emptyset; T; \emptyset \rangle \rangle = [\sigma_{\emptyset}]$, i.e. the empty state. Hence, the contextual information is lost as a result of merging leading to

$$\langle \langle c(X); T; \emptyset \rangle \rangle \odot [\sigma_{\emptyset}] = [\langle c(X); T; \emptyset \rangle] \odot [\sigma_{\emptyset}] = [\langle c(X); T; \emptyset \rangle],$$

although we would like to see the result $\langle \langle c(X); X = 1; \emptyset \rangle \rangle$. 

$\odot_{\mathcal{V}}$ can be defined over equivalence classes of states, since the corresponding representatives exist and $\odot_{\mathcal{V}}$ maintains state equivalence. Extract contextual information with $\odot_{\mathcal{V}}$. 

It is necessary to rather make $X$ a global variable first that is reduced by the merge operator $\circ_{\{X\}}$:

$$[(c(X); \top; \{X\}) \circ_{\{X\}} (\emptyset; X\doteq1; \{X\})]$$

$$= [(c(X); X\doteq1; \emptyset)]$$

$$= [(c(1); \top; \emptyset)].$$

Global variables can thus be used to share information between two states that are merged. This is used later for state splitting (c.f. Lemma 8), an important technique of CHR program analysis.

In program analysis, states are often restricted such that they only have global variables to make them distinguishable over all considered states. As demonstrated above, it suffices to consider those states to reason about all other states, as variables can be made local by the merge operator. Monotonicity (c.f. Lemma 7) ensures that the same transitions are possible regardless of the fact if a variable is global or local. This means that the operational semantics of CHR as defined in Definitions 8 and 12 does not distinguish global and local variables. [Rai10, p. 80].

In general, $\circ_V$ is not associative. However, the following lemma shows a restricted form of associativity that is used in the proof of the confluence modulo equivalence criterion in Section 6.3.

**Lemma 4** (Restricted Associativity of $\circ_V$ [Rai10, p. 52]). Let $\sigma_1, \sigma_2, \sigma_3$ be CHR states such that no local variable of a state occurs in another state. Then $[\sigma_1] \circ_V ([\sigma_2] \circ_V [\sigma_3]) = ([\sigma_1] \circ_V [\sigma_2]) \circ_V [\sigma_3]$ holds for all $V$.

This insight can be used to show that the set of equivalence classes of states together with $\circ$ is a commutative monoid (but only for $\circ$ i.e. $\circ_V$ with $V = \emptyset$). This fact becomes important in Section 3.3.2, where a partial order on states is derived from the merge operator that is used for invariant-based program analysis (c.f. Section 3.3.7).

**Lemma 5**. $(\Sigma_{CHR}/ \equiv, \circ)$ is a commutative monoid with neutral element $[\sigma_\emptyset]$ [Rai10, p. 51].

Note that $[\sigma_\emptyset]$ only is the neutral element of $\circ_V$ for $V = \emptyset$. For other $V$, $\sigma \circ_V \sigma_\emptyset \neq \sigma$ in general, because $\sigma$ may have global variables that also occur in $V$ [Rai10, p. 51].

### 3.3.2 Partial Order on States

Since the merge operator $\circ$ is a commutative monoid (c.f. Lemma 5), there exists an implied preorder. Raiser proved that this preorder is also antisymmetric and proposes the following partial order on CHR states:
Lemma 6 (Partial Order $\triangleleft$ [Rai10, p. 53]). For the set of CHR states $\Sigma_{\text{CHR}}$, the relation $\triangleleft \subseteq \Sigma_{\text{CHR}} \times \Sigma_{\text{CHR}}$ defined as

$$[\sigma] \triangleleft [\sigma'] \text{ if and only if } \exists [\delta]. [\sigma] \circ [\delta] = [\sigma'],$$

where $\sigma, \sigma' \in \Sigma_{\text{CHR}}$, is a partial order.

Note that due to the definition of the merge operator (c.f. Definition 14), special care has to be taken for states that contain local variables:

Example 10 (Partial Order $\triangleleft$ [Rai10, p. 53]). As one would expect for $\sigma_1 = \langle c(X); \top; \{X\} \rangle$ and $\sigma_2 = \langle c(X); X \doteq 1; \{X\} \rangle$, it holds that $[\sigma_1] \triangleleft [\sigma_2]$ by extension $[\sigma] = \langle \emptyset; X \doteq 1; \{X\} \rangle$. Intuitively, $\sigma_2$ can be obtained from $\sigma_1$ by adding the information that $X \doteq 1$, i.e. $\sigma_1$ is smaller than $\sigma_2$.

This intuition breaks for similar states $\sigma'_1 = \langle c(X); \top; \emptyset \rangle$ and $\sigma_2 = \langle c(X); X \doteq 1; \emptyset \rangle$, where the global variable $X$ has been made local. For those two states $[\sigma'_1] \triangleleft [\sigma'_2]$ does not hold. There is no $[\sigma']$ such that $[\sigma'_1] \circ [\sigma'] = [\sigma'_2]$, because Definition 14 prohibits the usage of local variable $X$ in $[\sigma']$. Intuitively, this can be explained by the fact that the two local variables $X$ denote different variables and have no relation. In fact, the local variables may be renamed by the definition of state equivalence (c.f. Definition 7). Hence, there is no simple extension of $\sigma'_1$ that produces an equivalent state to $\sigma'_2$.

There are other propositions of partial orders of CHR states. However, the relation proposed in [DSS07; DSS06] has not been proven to satisfy the conditions of a partial order, and in fact, in [Rai10, p. 54] it is shown that the definition of the relation lacks antisymmetry and transitivity and therefore cannot be a partial order.

For this reason, we use the partial order from Lemma 6 for our program analysis tools.

3.3.3 Monotonicity

Intuitively, the common idea of the program analysis methods in this thesis is to find a finite subset of states from which reasoning can be transferred to every other state of interest by extension. Generally speaking, this is achieved by exploiting monotonic properties of a transition system, i.e. properties that do not change their truth value by extending a state. Formally, this means that if a property $p$ holds in a state $\sigma$, i.e. $p([\sigma])$ is true, then $p([\sigma] \circ [\sigma'])$ is true for all $[\sigma']$. This allows to only consider some kind of $\triangleleft$-minimal state to reason about all larger states, i.e. states that are obtained by extending the minimal state.

One important monotonic property of CHR is rule application: In the operational semantics $\omega_{\text{va}}$ and $\omega_{\text{es}}$, all rules that are applicable in a state can also be applied in any larger state. Since this property is so central to program analysis in CHR, it is called the monotonicity

The trait of many CHR program analyses is to exploit monotonicity.

Monotonicity ensures that all applicable rules remain applicable when a state is extended.
property [Frü09, p. 85]. Formally, monotonicity can be expressed by the merge operator as in the following lemma:

**Lemma 7 (Monotonicity [Rai10, p. 51]).** If $[σ] \mapsto_e [τ]$, then $[σ] \odot_V [σ'] \mapsto_e [τ] \odot_V [σ']$ for all $V$.

The idea of all program analysis methods in this thesis is to exploit monotonicity of CHR. The $\ll$-minimal states of interest usually are rule states (c.f. Section 3.3.4 and Definition 15) or overlap states (c.f. Section 3.3.5 and Definition 17).

Note that the abstract operational semantics $ω_t$ breaks monotonicity, since rules become inapplicable depending on the propagation history. Since monotonicity is such a central part of CHR program analysis, the analysis results presented in this chapter have to be adapted for $ω_t$ [c.f. e.g. Frü09, pp. 108–110]. Since we concentrate on $ω_e$ in this thesis, this does not play a role for our analysis.

### 3.3.4 Rule States

Program analysis typically has to deal with an infinitely large set of states. Since the operational semantics of CHR is built around matching a pattern established by the rules of the program to the contents of a state, it typically is sufficient to only consider states that are constructed from the rules – the so-called rule states:

**Definition 15 (Rule State [Rai10, p. 78]).** For a rule $r : H_k \setminus H_r \leftrightarrow G \mid B_c, B_b$ let $V$ be the variables occurring in $H_k, H_r$ and $G$. Then the state $⟨H_k \cup H_r ; G ; V⟩$ is called the rule state of $r$. In the literature, the rule states are sometimes called minimal states.

Intuitively, a rule state contains exactly the constraints that are necessary to fire the rule it is constructed from. Due to monotonicity and the fact that $ω_e$ with equivalence classes as states (c.f. Definition 13) only consists of the rule application transition, it is possible to reason from the rule states about every other state in the transition system.

The justification for this is that the rule state $σ_r$ contains the minimal information such that the rule $r$ is applicable, i.e. every state where $r$ can be applied contains the $[σ_r]$. Let $[σ_r] \mapsto [τ]$. The result of the application of $r$ to any state $σ$ can be derived by replacing the part of $σ$ that is equivalent to the rule state $σ_r$ by $τ$. The situation is depicted in Fig. 3.1. It can be seen that $r$ can be applied to every extension of $σ_r$ and the result can be obtained by replacing $σ_r$ with $τ$. Since $σ_r$ contains the minimal information such that $r$ is applicable, all other states where $r$ is applied are $\ll$-greater than $σ_r$, i.e. can be subsumed by the rule state.

We want to formalize the idea of describing rule applications on arbitrary states by using the merge operator $\odot$ to split the state the rule is applied to into the rule state and a state that contains the remaining
information. We first start with an example to clarify the idea and continue with the formal Lemma 8.

Note that in the definition, rule states include all variables from the rule as global variables. This is necessary to share information between the rule state and the state with the remaining information.

The following example demonstrates the use of the merge operator for state splitting and justifies the need for global variables in the rule state:

**Example 11 (State Splitting with \( \diamond \) [Rai, p. 78]).** Let \( r : a(X) \Leftrightarrow b(X) \) be a rule. It is obviously applicable to the state \( \sigma = \langle a(3); \top; \emptyset \rangle \). The rule state of \( r \) is \( \sigma_r = \langle a(X); \top; \{X\} \rangle \). We want to express the rule application of \( r \) to \( \sigma \) by splitting \( \sigma \) into the rule state and some rest (we will call \( \delta \) using the merge operator.

Consider a version of the rule state without global variables

\[
\sigma'_r := \langle a(X); \top; \emptyset \rangle.
\]

It is clear that \( \sigma \equiv \langle a(X); X\equiv3; \emptyset \rangle =: \sigma' \) and hence \( X\equiv3 \) is contextual information of \( \sigma \) that does not play a role for the rule application. Consider the state \( \delta = \langle \emptyset; X\equiv3; \emptyset \rangle \) that contains this contextual information of \( \sigma \).

To merge \( \sigma'_r \) and \( \delta \), one has to rename the local variables first according to Definition 14. Lemma 2.1 allows us to do so freely and we hence obtain \( \delta \equiv \langle \emptyset; Y\equiv3; \emptyset \rangle =: \delta' \). The result is

\[
\sigma'_r \diamond \delta' = \langle a(X); Y\equiv3; \emptyset \rangle
\]

\[
\equiv \langle a(X); \top; \emptyset \rangle
\]

\[
\equiv \sigma'_r \neq \sigma',
\]

i.e. the contextual information is lost, as already described in Example 9. In other words, the states \( \delta \) and \( \delta' \) can be reduced to \( \delta \equiv \delta' \equiv \sigma_\emptyset = \langle \emptyset; \top; \emptyset \rangle \), i.e. the empty state that does not hold any information at all. This is the case because the local variable \( X \) from \( \sigma'_r \) and the local variable \( X \) from \( \delta \) do not have a relation as they are local to their corresponding state.

It is now obvious that a global variable is needed to share the information between the rule state and the rest. Hence, consider the actual rule state \( \sigma_r \) with the global variable \( X \). We then get

\[
\sigma_r \diamond_{\{X\}} \langle \emptyset; X\equiv3; \{X\} \rangle = \langle a(X); \top; \{X\} \rangle \diamond_{\{X\}} \langle \emptyset; X\equiv3; \{X\} \rangle \equiv \sigma.
\]
Hence, we can obtain $\sigma$ (that only has local variables) by merging the rule state with a corresponding rest containing the contextual information, although the latter two states only have global variables. This is the reason why the merge operator $\circ V$ allows to define a set of variables $V$ that are removed from the set of global variables in the result.

All of the above also holds for equivalence classes of states. Since we are allowed to switch freely between equivalence classes and syntactic states, we use $\omega e$ modulo state equivalence for the following observations. We now want to exemplify how state splitting can be used for rule application with the rule state. By construction, rule $r$ can be applied to the rule state as follows:

$$ [\sigma_r] = [(a(X); true; \{X\}) \mapsto_e [(b(X); true; \{X\})]. $$

Due to monotonicity (c.f. Lemma 7), we can extend this derivation by the contextual information to apply the rules in the larger state $[\sigma]$:

$$ [\sigma] = [(a(X); true; \{X\}) \circ_X [(\emptyset; X=3; \{X\})] \mapsto_e [(b(X); true; \{X\}) \circ_X [(\emptyset; X=3; \{X\})] = [(b(X); X=3; \emptyset)] = [(b(3); true; \emptyset)]. $$

In the example, it can be seen that monotonicity (c.f. Lemma 7) justifies the restriction to states without local variables in program analysis, since it allows us to transfer rule applications from rule states with only global variables to larger states (even if they do not contain global variables as in the example).

At the first glance this seems to be a complication of reasoning about rule applications without any benefits. However, the idea becomes convenient when reasoning about completely unknown states, as it is usual in program analysis due to the universal quantification over the propositions of interest.

**Lemma 8** (State Splitting with $\circ V$ [Rai10, p. 79]). Let the state $[\sigma]$ be applicable to a rule $r$. Then for the rule state $\sigma_r$ of $r$ with global variables $V$ it holds that

$$ \exists [\delta]. [\sigma] = [\sigma_r] \circ V [\delta]. $$

This is a central proof technique used in Part ii to reason about CHR programs modulo user-defined equivalence relations. It formally justifies the claim from above that we can reason from rule states about all other states. Since every state where $r$ is applicable can be split into the rule state $\sigma_r$ and some rest $\delta$, it suffices to reason about the rule state itself. If $[\sigma_r] \mapsto [\tau]$, we can simply replace $\sigma_r$ by $\tau$ in every larger state where $r$ is applicable to obtain its result of the rule application. Since $\sigma_r$ contains the minimal information such that $r$ is applicable, there are not other states, where $r$ can be applied.
3.3 Program Analysis

3.3.5 Confluence

Intuitively, by Definition 2, confluence is the property of a program (transition system) that holds, if for every (initial) state the same (result) state is obtained independently from the applied transitions, i.e. the program is deterministic. This means for CHR in particular that the selected matching rules do not play a role.

Testing a transition system for confluence is undecidable in general. However, for the fragment of terminating CHR programs, there is a decidable sufficient and necessary criterion for confluence [AFM96; Abd97; AFM99; Frü09, p. 101]. We reproduce those results on confluence of CHR programs as they form the base for our work on confluence modulo equivalence in Chapter 6.

The idea for the criterion comes from the field of term rewriting systems [Hue80]. For CHR, the confluence criterion uses monotonicity to reason from a (finite) set of (small) states about any larger state in the transition system. In the following steps, we describe the canonical confluence criterion of CHR and how it reduces the number of states that have to be considered.

Typically, confluence is defined modulo state equivalence for CHR, i.e. over equivalence classes of states (by otherwise using the general confluence definition for state transition systems as in Definition 2):

**Definition 16 (CHR Confluence [Frü09, p. 102]).** A CHR program is confluent if for all states \( \sigma, \sigma_1, \sigma_2 \): If \( \sigma \rightarrow^* \sigma_1 \), \( \sigma \rightarrow^* \sigma_2 \) then there exist states \( \tau_1, \tau_2 \), such that \( \sigma_1 \rightarrow^* \tau_1 \), \( \sigma_2 \rightarrow^* \tau_2 \) and \( \tau_1 \equiv \tau_2 \).

The equivalence class based semantics \( \omega_e \) allows to use the general definition of confluence for state transition systems (c.f. Definition 2). We comment on this in Section 6.6.3.

Naively, infinitely many states have to be considered to prove that a program is confluent. To reduce the number of states that have to be considered, the confluence criterion uses local confluence (c.f. Definition 3) instead of the plain confluence property (c.f. Definition 2). Newman’s Lemma (c.f. Lemma 1) states that confluence and local confluence coincide for terminating programs. This allows us to only consider a finite number of possible rule applications for any given state \( \sigma \).

However, there are still infinitely many states that have to be considered. The idea of the canonical confluence criterion is to syntactically overlap the rules of the program to form states in which both rules are applicable (i.e. that possibly are problematic for confluence). Those overlap states are \( \triangleleft \)-minimal, i.e. the smallest states where both rules are applicable. Hence, the set of all overlap states of two rules covers all states where the two rules are applicable, due to monotonicity (c.f. Lemma 7). From the overlap states we form so-called critical pairs. A critical pair consists of two states that are derived from applying the two overlapping rules to the overlap state.
A terminating program is confluent if and only if all its critical pairs are joinable.

Definition 17 (Overlap, Critical Pair [Rai10, p. 82]). For any two (not necessarily different) rules of a CHR program of the form

\[ r_1 : H_k \setminus H_r \leftrightarrow G \mid B_c, B_b, \quad r_2 : H'_k \setminus H'_r \leftrightarrow G' \mid B'_c, B'_b \]

and with variables that are renamed apart, let

\[ O_k \subseteq H_k, O_r \subseteq H_r, \quad O'_k \subseteq H'_k, O'_r \subseteq H'_r \]

be subsets of the heads of the rules such that for

\[ B := ((O_k \cup O_r) \cup (O'_k \cup O'_r)) \land G \land G' \]

it holds that \( \text{CT} \models \exists. B \) and \( (O_k \cup O_r) \neq \emptyset \), where \( \exists. B \) is the existential closure over \( B \). Then the state

\[ \sigma := \langle K \cup K' \cup R \cup R' \cup O_k \cup O_r; B; \mathcal{V} \rangle \]

is called an overlap of \( r_1 \) and \( r_2 \) where \( \mathcal{V} \) is the set of all variables occurring in heads and guards of both rules and

\[ K := H_k \setminus O_k, \quad K' := H'_k \setminus O'_k, \quad R := H_r \setminus O_r, \quad R' := H'_r \setminus O'_r. \]

The pair of states \( (\sigma_1, \sigma_2) \) with \( \sigma_1 := \langle K \cup K' \cup R' \cup O_k \cup B_c; B \land B_b; \mathcal{V} \rangle \) and \( \sigma_2 := \langle K \cup K' \cup R \cup O'_k \cup B'_c; B \land B'_b; \mathcal{V} \rangle \) is called critical pair of the overlap \( \sigma \). The critical pair can be obtained by applying the rules to the overlap state.

This is the most recent definition of an overlap as defined by Raiser [Rai10]. It syntactically differs from prior definitions [e.g. AFM96, p. 10; Frü09, p. 103] to match the definitions of states, rules, the merge operator and the operational semantics \( \omega_e \) (modulo state equivalence) that we use for the rest of this work.

Example 12 (Overlap, Critical Pair [Frü09, p. 105]). Consider the program with the following two rules that implement a non-deterministic coin toss:

\[ h : \text{throw(Coin)} \leftrightarrow \text{Coin} \vdash \text{head}. \]
\[ t : \text{throw(Coin)} \leftrightarrow \text{Coin} \vdash \text{tail}. \]

The program only has the following non-trivial overlap state:

\[ \sigma_o := \langle \text{throw(Coin)}; \text{Coin} \vdash \text{Coin}; \{\text{Coin}, \text{Coin}\} \rangle \]

This leads to the following critical pair:

\[ \sigma_1 := \langle \emptyset; \text{Coin} \vdash \text{head} \land \text{Coin} \vdash \text{Coin}; \{\text{Coin}, \text{Coin}\} \rangle, \quad \sigma_2 := \langle \emptyset; \text{Coin} \vdash \text{tail} \land \text{Coin} \vdash \text{Coin}; \{\text{Coin}, \text{Coin}\} \rangle. \]

Note that for the states in the critical pair it holds that \( \sigma_o \mapsto^h \sigma_1 \) and \( \sigma_o \mapsto^t \sigma_2 \).
In the confluence criterion, the critical pairs are then tested for joinability, i.e. it is checked if there is an arbitrary number of derivation steps that lead to the same state for both states in the critical pair. The following theorem states that if this is possible for all critical pairs, the program is confluent.

**Theorem 5** (Confluence Criterion [Früo9, p. 104]). A terminating CHR program is confluent if and only if all its critical pairs are joinable.

This sufficient and necessary criterion for confluence of CHR programs is decidable for terminating programs: The number of critical pairs is finite (because CHR programs are finite sets of rules) and since the program is terminating, it is possible to test the critical pairs for joinability.

It even suffices to execute the program on both states of each critical pair until a final state is reached. This is possible, although the derivations in the definition of joinability (c.f. Definitions 2 and 16) are existentially quantified and therefore – in theory – all possible (non-deterministic) derivations have to be considered. However, the proof of Newman’s Lemma also shows that for terminating locally confluent programs, the final states are equivalent independent from the selected derivation path [Rai10, p. 83; New42]. This can be comprehended by the following considerations: If the two final states \( \sigma \) and \( \sigma' \) are not equivalent, the program cannot be locally confluent by Definition 3, since they cannot be joined. If they are equivalent, we already have found a derivation path that shows joinability and we are done.

**Example 13** (Confluence of Coin Toss [Früo9, p. 105]). The coin toss program from Example 12 is not confluent: The critical pair

\[
\sigma_1 := \langle \emptyset; \text{Coin}_1 = \text{head} \land \text{Coin}_1 = \text{Coin}_2; \{\text{Coin}_1, \text{Coin}_2\} \rangle,
\]

\[
\sigma_2 := \langle \emptyset; \text{Coin}_2 = \text{tail} \land \text{Coin}_1 = \text{Coin}_2; \{\text{Coin}_1, \text{Coin}_2\} \rangle.
\]

is not joinable, since \( \sigma_1 \) and \( \sigma_2 \) are both final states and obviously \( \sigma_1 \not\equiv \sigma_2 \).

### 3.3.6 Operational Equivalence

The idea of the confluence criterion can be extended to prove operational equivalence (c.f. Definition 4) of two programs.

**Definition 18** (\( P_1, P_2 \)-Joinability [Früo9, p. 128]). Let \( P_1, P_2 \) be two CHR programs and let the notation \( \rightarrow_p \) denote a transition using program \( P \). A state \( \sigma \) is \( P_1, P_2 \)-joinable if and only if there are computations \( \sigma \rightarrow_{P_1}^+ \tau_1 \) and \( \sigma \rightarrow_{P_2}^+ \tau_2 \) such that \( \tau_1 \equiv \tau_2 \) or \( \sigma \) is a final state in both programs.

For terminating and confluent programs, operational equivalence is decidable by the following criterion.

**Theorem 6** (Operational Equivalence Criterion [Früo9, p. 128]). Two terminating and confluent programs \( P_1, P_2 \) are operationally equivalent if and only if all rule states of the rules in \( P_1 \) and \( P_2 \) are \( P_1, P_2 \)-joinable.
3.3.7 Invariant-Based Program Analysis

So far, all analysis techniques refer to plain CHR without invariants on states and without user-defined equivalence relations different from state equivalence. In that world, everything is clean in a sense that monotonicity holds without restriction and therefore rule states can be used directly to reason about all CHR states. Furthermore, state equivalence harmonizes beautifully with the operational semantics of CHR and thus with the merge operator and the monotonicity property.

However, although those analysis methods are already powerful, they are often not applicable as they consider states that are not reachable in practice. This inhibits program analysis for practical programs.

For this purpose, invariants can be used to restrict the states of a program to the ones that are practically relevant ignoring all other states. This can be exploited to weaken the program property of interest such that it suffices for practical applications. For instance, if a program is not confluent because some states that are not reachable in practice yield different results depending on the chosen derivations, those states can be ignored as the program behaves like a confluent program in practice.

In this section the work of [Rai10; DSS06; DSS07] on invariant-based program analysis is reproduced as it serves as a foundation for our work on program analysis modulo equivalence relations.

3.3.7.1 Invariants

We start with the definition of an invariant.

**Definition 19 (Invariant).** A property \( I \) is an invariant if and only if for all states \([\sigma]\) where \( I([\sigma]) \) holds and for all \([\tau]\) with \([\sigma] \rightarrow^* [\tau]\) the invariant \( I([\tau]) \) holds as well. In CHR, invariants are restricted to equivalence classes of states, i.e., they either hold or do not hold for all states of an equivalence class.

Invariants can be used to restrict the state space of a program to the subset of states that satisfies the invariant. This produces a reduced state transition system over the subset of states defined by the invariant, where all transitions that contain states where the invariant does not hold are excluded.

**Definition 20 (Reduced State Transition System).** Let \( P = (\Sigma, \rightarrow) \) be an arbitrary state transition system and \( I \) be an invariant on the states in \( \Sigma \). Then the reduced transition system with respect to invariant \( I \) is defined as \( P^I := (\Sigma', \rightarrow') \) where \( \Sigma' = \{ \sigma \in \Sigma \mid I(\sigma) \text{ holds} \} \) and \( \rightarrow' \subseteq \Sigma' \times \Sigma' \) is defined as \( \rightarrow' := \{(\sigma, \sigma') \in \rightarrow \mid \sigma, \sigma' \in \Sigma'\} \).

Since many programs define an interface that restricts the allowed (initial) states, invariants can be used to formalize the correct use of
the program. In other words, invariants make implicit assumptions made during the creation of a program explicit. Additionally, some programs might only terminate for all states that satisfy a certain invariant $I$. We call such programs $I$-terminating.

**Definition 21 (I-Termination).** Let $I$ be an invariant. A state transition system $P$ is $I$-terminating, if the reduced state transition system $P^I$ is terminating.

We exemplify the notion of invariants by a program from the Blocks World domain.

**Example 14 (Blocks World [Rai10, pp. 84–85; DSS07, pp. 225–226]).** A classical example in artificial intelligence planning is the Blocks World. It features a robot arm that can pick up and move boxes that are stacked on top of each other. In this example, we do not consider stacks of objects, but are only interested in a simple single robot arm that can pick up one box at a time. This can be modeled by the following two rules:

$r_1 : \text{get}(X), \text{empty} \iff \text{hold}(X)$.
$r_2 : \text{get}(X), \text{hold}(Y) \iff \text{hold}(X), \text{clear}(X)$.

The constraints have the following meaning:

- $\text{get}(X)$ denotes that the robot arm intends to pick up box $X$.
- $\text{empty}$ denotes that the robot is not holding a box.
- $\text{hold}(X)$ denotes that the robot is holding box $X$.
- $\text{clear}(X)$ denotes that box $X$ is accessible to the robot arm, i.e. it is not held (and usually it also means that no box is stacked on top of it).

In general program analysis, inconsistent states are also considered as they are part of the transition system, although it is clear for humans that they do not describe valid states. For instance the state

$(\text{hold}(X), \text{hold}(Y); \top; \{X, Y\})$

is inconsistent, since the robot arm can only hold one box at a time (which is implicitly assumed in the rules $r_1$ and $r_2$). Another inconsistent example is the state $(\text{hold}(X), \text{clear}(X); \top; \{X\})$, since a box cannot be held and clear at the same time.

The state transition system can exclude such inconsistent states by defining an invariant $BW$ that holds if and only if

- either the agent holds some box $X$ or holds nothing and
- there is at most one $\text{get}(_)$ constraint at a time.

It can easily be seen that the invariant is preserved by both rules and therefore matches the requirements of Definition 19.
Consequently, program analysis should be restricted to states where the invariant holds, since all other states are anyway excluded from the transition system. If a desired program analysis property is broken by a state that is excluded by the invariant (and therefore does not appear in practice), the desired property may hold in the reduced state transition system. For instance, confluence is often broken by states that never appear in practice. When only the set of allowed states is considered, the program is confluent and therefore can be considered as confluent in practice. The concept of invariant-based confluence (also called observable confluence) as presented in Section 3.3.7.2 formalizes this idea.

However, introducing invariants to a state transition system comes at a cost: The program analysis techniques have to be adapted to invariant-based program analysis. The problem is that the typical program analysis criteria of CHR base on monotonicity (c.f. Lemma 7) and invariants break monotonicity: Consider some rule state (or overlap state) $\sigma$ of a program $P$ that does not satisfy the invariant $I$. Therefore, $\sigma$ is not part of the reduced transition system $P^I$. Hence, monotonicity cannot be used for this state to reason about other states in $P^I$.

Although, $\sigma$ is not part of $P^I$, it cannot be just ignored: There might be larger states obtained by extension of $\sigma$ that reestablish the invariant. This means that although $I([\sigma])$ does not hold, there might be extensions $\sigma'$ such that $I([\sigma \diamond \sigma'])$ holds. If we ignore those states of the form $[\sigma \diamond \sigma']$, we miss parts of our reduced transition system $P^I$ and therefore our analysis is invalid.

**Example 15 (Invariants Break Monotonicity).** Consider the simple CHR program consisting of one rule $r: a(X) \iff b(X)$. Let the invariant $I$ hold in the state $\sigma$ if and only if $\sigma$ contains at least two user-defined constraints.

Program analysis typically uses monotonicity to reason from small states about any larger state. For instance, rule states can be used to reason about rule applications in a program. According to Definition 15, the rule state of $r$ is $\sigma_r = \langle a(X); \top; \{X\} \rangle$. It can easily be verified that $I([\sigma])$ does not hold. This invalidates every program analysis results based on the rule state $\sigma_r$, since it is not part of the reduced state transition system and therefore Lemma 7 cannot be applied. This means that in the reduced state transition system no information can be gained from the rule state.

To reestablish reasoning using monotonicity, the smaller state used as a basis for reasoning has to be extended such that the invariant holds. In our example, $\sigma_r$ can be merged with the state $\delta = \langle a(X); \top; \{X\} \rangle$ such that $I([\sigma_r \diamond \delta])$ holds. Other possible extensions would be $\sigma_r \diamond \langle a(X); X=1; \{X\} \rangle$ or even $\sigma_r \diamond \langle b(X); \top; \{X\} \rangle$.

The idea is now to find a (desirably finite) set of small states that capture all states of interest, i.e. that are part of $P^I$, and therefore allow to reason about all other states in the reduced state transition system $P^I$. Those states can be constructed by extending the states of
interest from the original program $P$, such that they become part of the reduced transition system $P^I$.

**Definition 22.** ($I$-Extensions, Minimal Extensions [Rai10, p. 80]). For an invariant $I$, let the set

$$
\Sigma^I([\sigma]) := \{[\sigma'] \mid I([\sigma \circ \sigma']) \land \sigma' \text{ has no local variables}\}
$$

be the set of extensions of $[\sigma]$ such that the invariant $I$ holds.

The set $M^I([\sigma])$ is the set of $\triangleleft$-minimal elements of $\Sigma^I([\sigma])$ such that $\forall[\sigma'] \in \Sigma^I([\sigma]) \exists[\sigma_m] \in M^I([\sigma]). M^I([\sigma])$ is also referred to as the set of minimal $I$-extensions of $[\sigma]$.

The extensions of a state $\sigma$ that reestablish the invariant are collected in the set of extensions $\Sigma^I([\sigma])$. This set contains all of those extensions of arbitrary size, i.e. also possibly infinite chains like $[\sigma] \circ [\sigma'] \circ ([\sigma] \circ [\sigma']) \circ [\sigma'] \circ \ldots$ where both $I([\sigma] \circ [\sigma'])$ and $I(([\sigma] \circ [\sigma']) \circ [\sigma''])$ hold. Hence, since there are possibly infinitely many state extensions that satisfy the invariant, we need some minimality criterion for states that need to be considered to reestablish monotonicity when invariants are used.

The partial order $\triangleleft$ on states allows to extract a set of minimal elements $M^I([\sigma])$ that can be used to construct every other possible state that contains the $\sigma$ by using extension via the merge operator $\circ$. Hence, when considering all of those minimal extensions of $\sigma$, reasoning about monotonic properties is possible again. For instance, since monotonicity (c.f. Lemma 7) guarantees that every state a rule is applicable in is an extension of the rule state, the set $\Sigma^I([\sigma])$ contains all states where the rule is applicable and that satisfy the invariant $I$.

The situation is depicted in Fig. 3.2 that shows a Hasse Diagram with respect to the partial order $\triangleleft$. The diagram contains states that result from extending some state $\sigma$ of interest. The red states result from extending $\sigma$ by states that are not part of $\Sigma^I([\sigma_r])$, i.e. the red states do not satisfy the invariant. The green states are states where the invariant holds, i.e. that are obtained by extending $\sigma$ with elements from $\Sigma^I([\sigma])$. The minimal states where $I$ holds are framed. Those
are the states that are the result of extending \( \sigma \) by states \( \sigma_m \) that are both in \( \Sigma^I([\sigma]) \) and \( M^I([\sigma]) \). Note that the minimal extensions are not unique. In fact, it is possible that there are different extensions of a rule state that reestablish the invariant but are not extensions of each other. To capture all extensions in \( \Sigma^I([\sigma]) \) by using monotonicity, all those minimal extensions have to be considered.

There are also situations where the set of minimal extensions is infinitely large.

There are also extensions of \( \sigma \) that contain both the rule state and a (minimal) extension of it, but that invalidate the invariant. Those states are colored in red in Fig. 3.2. However, those states are not part of the reduced transition system and can hence be ignored when reasoning about programs with respect to invariants. In fact, those are the states that we want to exclude by the invariant as they represent invalid inputs of the program and, hence, are not part of the program. For example, the state \( \rho := \sigma \circ \delta_1 \circ \delta_{n_1} \) in Fig. 3.2 does not satisfy the invariant, although it is an extension of the state \( \sigma \circ \delta_1 \) that satisfies the invariant. This means that \( \delta_{n_1} \) adds some information to \( \sigma \circ \delta_1 \) that invalidates the invariant. Since \( \rho \) does not satisfy the invariant, it is not part of the reduced state transition system and therefore not of interest for our program analysis. All larger states of \( \rho \), that again satisfy the invariant, are already captured by \( \sigma \circ \delta_1 \) and rules that are applicable in \( \sigma \circ \delta_1 \) will also be applicable in any extension of \( \rho \). Hence, we can ignore \( \rho \) in our analysis.

As can be seen in Fig. 3.2, when considering all minimal extensions of a state of interest \( \sigma \), all larger states that contain \( \sigma \) and that are part of the reduced transition system (i.e. satisfy the invariant) are captured by the analysis and therefore the idea of exploiting monotonicity can be applied again. Altogether, this means that analyses about monotonic properties are possible by extending all states of interest by the minimal extensions.

Depending on the available built-in constraints and the invariant, the set of minimal extensions \( M^I([\sigma]) \) may not be well-defined or infinite. Here, well-definedness means that the condition in Definition 22 is met, i.e. such minimal elements exist. Although a finite set of minimal extensions is desired for automated program analysis, the general approach is still valid for infinitely large sets of minimal extensions (at the cost of losing decidability in general). A classical example of an invariant that produces an infinite set of minimal extensions are \(<\) constraints on integers:

\[\text{Example 16 (Infinite Set of Minimal Extensions [ Rai10, p. 80; DSS07, p. 235])}. \text{Consider a CHR system with } <\text{-built-in constraints over integer numbers with the usual meaning. Let } I \text{ hold if and only if for every constraint } n(N) \text{ the built-in store implies } N < k \text{ for some constant } k. \]

Let \( \sigma = \langle n(N); T; \{N\} \rangle \) and let \( \sigma_i = \langle \emptyset; N < i; \{N\} \rangle \) for any integer \( i \). Then the set \( \Sigma^I([\sigma]) \) of \( I\)-extensions of \( \sigma \) contains the infinite sequence of states \( [\sigma_2], [\sigma_1], [\sigma_0], [\sigma_{-1}], \ldots \) with \( \ldots \ll [\sigma_{-1}] \ll [\sigma_0] \ll [\sigma_1] \ll [\sigma_2] \).
For none of these states exists a \( \triangleleft \)-minimal element. Hence, \( M^I([\sigma]) \) is not well-defined.

Note that in Definition 22 the set \( \Sigma^I([\sigma]) \) is only defined over extensions without local variables. This is to avoid that the set of minimal elements \( M^I \) may become infinitely large for extensions with local variables: For some invariant \( \mathcal{I} \) and state \([\sigma]\), consider the two states \( \sigma_1 = \langle c(X); \top; \emptyset \rangle \) and \( \sigma_2 = \langle c(X); X \leftarrow 1; \emptyset \rangle \) that both are valid extensions of \([\sigma]\) in our example and therefore elements of \( \Sigma^I([\sigma]) \). Since \( \sigma_2 \) is more specific than \( \sigma_1 \), we would like to restrict our analysis to \( \sigma_1 \). However, since neither \( [\sigma_1] \triangleleft [\sigma_2] \) nor \( [\sigma_2] \triangleleft [\sigma_1] \) hold, both states – and consequently infinitely many states – have to be included to the set of minimal elements \( M^I([\sigma]) \). Hence in program analysis, w.l.o.g. the states are restricted to only global variables. This is further justified by the fact that monotonicity of CHR (c.f. Lemma 7) ensures that all results remain applicable if any of these variables are made local as already discussed in more detail for rule states and state splitting in Section 3.3.4 and Example 11 [Rai10, p. 80].

For states where the invariant already holds, we can show that the set of minimal extensions only consists of the empty state, since it is not necessary to extend the state such that the invariant holds.

**Lemma 9.** For an invariant \( \mathcal{I} \) and a state \( \sigma \) such that \( \mathcal{I}([\sigma]) \) holds, we have \( M^I([\sigma]) = \{ [\emptyset] \} \) [Rai10, p. 80].

Note that the set of minimal extensions is empty for states that cannot be extended such that the invariant holds. I.e., if there is no \( \tau \) such that \( \mathcal{I}([\sigma]) \) holds, then \( M^I([\sigma \circ \tau]) = \emptyset \) (where both \( \sigma \) and \( \tau \) have no local variables).

### 3.3.7.2 Invariant-Based Confluence

The definition of confluence can be restricted to only consider the reduced state transition system with respect to an invariant, i.e. only the states where the invariant holds. This allows to ignore states that are undesired and do not occur in practice.

**Definition 23** (Invariant-Based Confluence [Rai10, p. 85]). A CHR program is confluent with respect to an invariant \( \mathcal{I} \) if and only if for all states \([\sigma], [\sigma_1], \) and \([\sigma_2]\) where \( \mathcal{I}([\sigma]) \) holds, we have that

\[
[\sigma] \not\rightarrow^* [\sigma_1] \land [\sigma] \not\rightarrow^* [\sigma_2] \rightarrow \mathcal{I}([\sigma_1]) \land \mathcal{I}([\sigma_2]) \text{ hold as well.}
\]

As described before, invariants break monotonicity. Hence, the basic confluence criterion from Section 3.3.5 fails: If in the confluence test a constructed overlap does not satisfy the invariant, then this overlap state is not part of the transition system and therefore no information
can be gained from analyzing it. However, the confluence test uses the small overlap states that are syntactically generated from the rules to reason about all states where those rules are applicable. Although the invariant might not hold in the overlap state, there might be larger states, i.e. extensions of the overlap state, where the invariant holds. Hence, those extensions have to be considered in the confluence criterion as well.

For this reason, we use the methods described in Section 3.3.7.1, namely the set of minimal extensions $M^I(\sigma)$ to extend overlap states $\sigma$. In [DSS07; DSS06; Rai10] the following has been proven: If we can show that for all overlap states $\sigma$ of a terminating program the critical pairs derived from all states in $M^I(\sigma)$ are joinable, the program is confluent with respect to $I$.

**Theorem 7 (Criterion for Invariant-Based Confluence [Rai10, p. 86]).** Let $I$ be an invariant, $P$ an $I$-terminating CHR program, and let $M^I(\sigma)$ be well-defined for all overlap states $\sigma$, then:

$P$ is confluent with respect to $I$, if and only if for all overlap states $\sigma$ with critical pairs $(\sigma_1, \sigma_2)$ and all $[\sigma_m] \in M^I(\sigma)$ holds $([\sigma_1] \diamond [\sigma_m]) \downarrow ([\sigma_2] \diamond [\sigma_m])$.

Theorem 7 extends the basic confluence criterion from Theorem 5. In addition to the basic confluence criterion, we have to consider all minimal extensions of the overlap states that satisfy the invariant. If all of those extended overlap states are joinable, the program is confluent. Note that since we use equivalence classes of states, joinability from Definition 2 coincides with joinability from the CHR confluence definition in Definition 16 [RBF09, p. 13]. We comment on this in more detail in Section 6.6.3.

For overlap states where the invariant holds, the invariant-based confluence test coincides with the basic confluence criterion. This follows from Lemma 9.

For overlap states that cannot be extended such that the invariant holds, the corresponding critical pair does not play a role for the confluence criterion, since $M^I(\sigma) = \emptyset$, i.e. the precondition of Theorem 7 is satisfied trivially. The intuition behind this is that all states that are extensions of this state $\sigma$ are not part of the reduced transition system and can therefore be ignored.

**Example 17 (Confluence of Blocks World [Rai10, pp. 86–87]).** Reconsider the Blocks World program from Example 14 with the invariant $BW$. Due to the lack of built-in constraints it is clear that $M^{BW}(\sigma)$ is well-defined for all states $\sigma$. Hence, the prerequisites of Theorem 7 are met. The two rules $r_1$ and $r_2$ lead to the following single overlap state:

$$\sigma = \langle \text{get}(X), \text{empty}, \text{hold}(Y); \top; \{X, Y\} \rangle.$$ 

This state obviously invalidates invariant $BW$, since the robot arm cannot be empty and holding an object at the same time. However, the inconsistent
Intuition behind the Invariant-Based Confluence Criterion

Decidability of the Criterion

Operational equivalence can be generalized to invariants.

The operational equivalence property and the corresponding decision criterion can be adapted to settle some issues.

Interfaces define the constraints shared by the program.

The notion of operational equivalence as defined in Definition 4 can be generalized as proposed in [Rai10, pp. 91–97] such that it integrates program invariants. This allows to only consider the reduced state transition system, i.e., states that satisfy the invariant and therefore relaxing the conditions for operational equivalence. This can be used to restrict operational equivalence to states that appear in practice.

There are some other issues with the general notion of operational equivalence and the corresponding decision criterion (c.f. Theorem 6. The work of Raiser, addresses those issues while developing an invariant-based operational equivalence decision criterion [Rai10, pp. 91–97]. This section summarizes those efforts that serve as a basis for our work on operational equivalence.

A typical practical problem of classical operational equivalence is that the exact same set of constraint symbols is assumed in both programs. However, although a common interface might be defined for a certain problem, many programs may introduce constraint symbols that are used to modularize the program or compute intermediate results in utility libraries. For this reason, operational c-equivalence has been introduced [AF99; Frü09, pp. 129–132]. It only considers input states that merely contain user-defined constraints with constraint symbol c. Hence, analysis is restricted to some constraint of interest. The idea can be generalized to so-called interfaces of two programs. This is the approach pursued in this section.

**Definition 24** (Interface [Rai10, p. 92]). Let \( P_1, P_2 \) be two CHR programs, then their interface is the set \( I := \mathcal{C}(P_1) \cap \mathcal{C}(P_2) \).
The idea is that we only allow the constraint symbols from the interface in both input and final states of the two programs of interest.

**Definition 25** (Interface State [Rai10, p. 93]). A state $\sigma = (G; B; V)$ is called an $I$-state for an interface $I$, if and only if all constraint symbols in $G$ occur in $I$.

We straightforwardly extend this definition to rule states and equivalence classes of states. In the latter case, the equivalence class of failed states is considered an $I$-state for all interfaces $I$.

Note that the interface only considers user-defined constraints, since for operational equivalence analysis we implicitly assume the same constraint theory for both programs [Rai10, p. 92].

Furthermore, there are also some other restrictions of the classical operational decision criterion from Theorem 6. It requires terminating and confluent programs. To generalize the decision criterion for invariant-based operational equivalence such that it also is applicable to non-terminating or non-confluent programs, we do not check for joinability as in Theorem 6, but rather compare all result states. Therefore, the notions of normal forms and normal form equivalence are defined in the following.

**Definition 26** (Normal Forms, $\mathcal{NF}$-Equivalence [Rai10, p. 93]). For a state $\sigma$, the set

$$ \mathcal{NF}_P(\sigma) = \mathcal{NF}_P([\sigma]) := \{ [\tau] \mid [\sigma] \rightarrow^* P [\tau] \not\rightarrow P \} $$

is called the set of normal forms of $\sigma$.

A state $\sigma$ is $\mathcal{NF}$-equivalent with respect to $P_1$ and $P_2$ if and only if $\mathcal{NF}(P_1) = \mathcal{NF}(P_2)$. If $P_1$ and $P_2$ are clear from the context, we simply write $\sigma$ is $\mathcal{NF}$-equivalent. The definition can be lifted straightforwardly to equivalence classes of states.

As mentioned before, the restriction to only interface states should also hold for final states.

**Definition 27** ($\mathcal{NF}$-$I$-Equivalence [Rai10, p. 93]). Let $\sigma$ be a state and $P_1$ and $P_2$ be CHR programs with interface $I$. Then, $\sigma$ is $\mathcal{NF}$-$I$-equivalent w.r.t. $P_1$ and $P_2$ if and only if $\mathcal{NF}(P_1)$ and $\mathcal{NF}(P_2)$ are $I$-states.

Finally, invariant-based operational equivalence can be defined.

**Definition 28** (Invariant-Based Operational Equivalence [Rai10, p. 93]). Let $P_1, P_2$ be CHR programs with interface $I$ and $I$ an invariant. $P_1$ and $P_2$ are $I$-$I$-equivalent if and only if all $I$-states $\sigma$ where $I([\sigma])$ holds are $\mathcal{NF}$-$I$-equivalent.

Raiser proposes the following decision criterion for $I$-$I$-equivalence.
Theorem 8 (Deciding Invariant-Based Operational Equivalence [Rai10, p. 94]). Let \( \mathcal{P}_1, \mathcal{P}_2 \) be two CHR programs with interface \( \mathcal{I} \) and \( \mathcal{I} \) an invariant for which \( \mathcal{M}^I \) is well-defined for all \( \mathcal{I} \)-rule states. \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are \( \mathcal{I} \)-equivalent, if and only if for all \( \mathcal{I} \)-rule states \( \sigma_r \) of \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) and all \( [\sigma_m] \in \mathcal{M}^I([\sigma_r]) \) it holds that \( [\sigma_r] \circ [\sigma_m] \) is \( \mathcal{N} \mathcal{F} \)-\( \mathcal{I} \)-equivalent.

The theorem is a generalization of Theorem 6. The original theorem can be retrieved by using an invariant that holds for all states. Furthermore, all constraint symbols are shared between both programs (and are therefore part of the interface). The decision criterion is not restricted to terminating and confluent programs. However, decidability is lost for programs that are non-terminating or non-confluent.
Adaptive Control of Thought – Rational (ACT-R) is a popular cognitive architecture that is used in many cognitive models to describe and explain human cognition. There have been applications in language learning models [TA02] or in the field of human computer interaction [Byro1].

Using a cognitive architecture like ACT-R simplifies the modeling process, since well-investigated psychological results have been assembled to a unified theory about fundamental parts of human cognition. In the best-case, such an architecture constrains modeling to only plausible cognitive models [TLA06]. Computational cognitive models are described clearly and unambiguously since they are executed by a computer producing detailed simulations of human behavior [Sun08]. By performing the same experiments on humans and the implemented cognitive models, the resulting data can be compared and models can be validated.

4.1 Overview of the ACT-R Architecture

The ACT-R theory is built around a modular production rule system operating on data elements called chunks. A chunk is a structure consisting of a name and a set of labeled slots that are connected to other chunks. The slots of a chunk are determined by its type. The names of the chunks are only for internal reference – the information represented by a network of chunks comes from the connections. For instance, there could be chunks representing the cognitive concepts of numbers 1, 2, . . . . By chunks with slots number and successor we can connect the individual numbers to an ordered sequence describing the concept of natural numbers. This is illustrated in Fig. 4.1.

Figure 4.1: Two count facts with names b and c that model the counting chain 1, 2, 3 [GF15b, p. 115; GF18a, p. 22:4].

As shown in Fig. 4.2, ACT-R consists of modules. The goal module keeps track of the current (sub-) goal of the cognitive model. The
declarative module contains declarative knowledge, i.e. factual knowledge that is represented by a network of chunks. There are also modules for interaction with the environment like the visual and the manual module. The first perceives the visual field whereas the latter controls the hands of the cognitive agent. Each module is connected to a set of buffers that can hold at most one chunk at a time.

The heart of the system is the procedural module that contains the production rules controlling the cognitive process. It only has access to a part of the declarative knowledge: the chunks that are in the buffers. A production rule matches the content of the buffers and – if applicable – executes its actions. There are three types of actions:

MODIFICATIONS overwrite information in a subset of the slots of a buffer, i.e. they change the connections of a chunk.

REQUESTS ask a module to put new information into its buffer. The request is encoded in form of a chunk. The implementation of the module defines how it reacts on a request. For instance, there are modules that only accept chunks of a certain form like the manual module that only accepts chunks that encode a movement command for the hand according to a predefined set of actions.

Nevertheless, all modules share the same interface for requests: The module receives the arguments of the request encoded as a
chunk and puts its result in the requested buffer. For instance, a request to the declarative module is stated as a partial chunk and the result is a chunk from the declarative knowledge (the fact base) that matches the chunk from the request.

**SL** remove the chunk from a buffer.

The system described so far is the so-called *symbolic level* of ACT-R. It is similar to standard production rule systems operating on symbols (of a certain form) and matching rules that interact with buffers and modules. However, to simulate the human mind, a notion of timing, latency, priorities etc. are needed. In ACT-R, those concepts are subsumed in the *sub-symbolic level*. It augments the symbolic structure of the system by additional information to simulate the previously mentioned concepts.

Therefore, ACT-R has a progressing simulation time. Certain actions can take some time that depends on the information from the sub-symbolic level. For instance, chunks are mapped to an *activation level* that determines how long it takes the declarative module to retrieve it. Activation levels also resolve conflicts between chunks that match the same request. The value of the activation level depends on the usage of the chunk in the model (inter alia): Chunks that have been retrieved recently and often have a high activation level. Hence, the activation level changes with the simulation time. This can be used to model learning and forgetting of declarative knowledge. Similar to the activation level of chunks, production rules have a *utility* that also depends on the context and the success of a production rule in prior applications. Conflicts between applicable rules are resolved by their utilities which serve as dynamic, learned rule priorities.

### 4.2 Syntax

We use a simplified syntax of ACT-R that we have introduced in [GF15a]. It is based on sets of logical terms instead of the concatenation of syntactical elements. This enables an easier access to the syntactical parts. Our syntax can be transformed directly to the original ACT-R syntax and vice-versa.

An ACT-R architecture is defined over two possibly infinite, disjoint sets of (constant) symbols $\mathcal{C}_A$ and variable symbols $\mathcal{V}_A$. Each architecture defines a set of buffers $\mathcal{B} \subseteq \mathcal{C}_A$.

An ACT-R model in an architecture consists of a set of types $\mathcal{T} \subseteq \mathcal{C}_A$ with type definitions and a set of rules $\mathcal{R}$. A production rule has the form $L \Rightarrow R$ where $L$ is a finite set of buffer tests and queries. A buffer test is a first-order term of the form $= (b, t, P)$ where $b \in \mathcal{B}$ is a buffer, $t \in \mathcal{T}$ a type and $P \subseteq \mathcal{C}_A \times (\mathcal{C}_A \cup \mathcal{V}_A)$ is a set of slot-value pairs $(s, v)$ where $s \in \mathcal{C}_A$ and $v \in \mathcal{C}_A \cup \mathcal{V}_A$. This means that only the values in the slot-value pairs can consist of both constants and variables.
The architecture defines a set of action symbols $A$ from which the right-hand side of the rules can be formed. Usually, the action symbols are defined as $A := \{=,+,-\}$ for modifications, requests and clearings respectively.

The right-hand side $R \subseteq A$ of a rule is a finite set of actions where $A := \{a(b,t,P) \mid a \in A, b \in B, t \in T$ and $P \subseteq C_A \times (C_A \cup V_A)\}$. This means that an action is a term of the form $a(b,t,P)$ where the functor $a$ of the action is in $A$, the set of action symbols, the first argument $b$ is a constant (denoting a buffer), the second argument is a constant $t$ denoting a type, and the last argument is a set of slot-value pairs, i.e. a pair of a constant and a constant or variable. Only one action per buffer is allowed, i.e. if $a(b,t,P) \in R$ and $a'(b',t',P') \in R$ then $a = a'$, $t = t'$ and $P = P'$.

Note that the same symbol ($=$) is used for both buffer tests and modifications, although they denote different concepts. Nevertheless, this thesis adheres to those symbols as they appear in the ACT-R reference implementation and are widely employed.

Again, the function $\text{vars}$ maps an arbitrary set of terms to its set of variables in $V_A$. For a production rule $L \Rightarrow R$ the following must hold: $\text{vars}(R) \subseteq \text{vars}(L)$, i.e. no new variables must be introduced on the right-hand side of a rule. As we will see in the following sections about semantics, this restriction demands that all variables are bound on the left-hand side.

The syntactic restrictions on rules are summarized in the following definition of well-formedness as an abstract syntax description. We allow to transfer information in non-terminal symbols as arguments and denote enumerations of non-terminal symbols as subsets.

- production rule: $P ::= L \Rightarrow R(L)$
- left-hand side: $L ::= \{T_1, \ldots, T_n\}, n > 0$
- buffer test: $T ::= (b,t,\{ST_1(t),\ldots,ST_n(t)\}), \quad n \geq 0$
- slot test: $ST(t) ::= (s,v)$ if $s \in \tau(t)$
- right-hand side: $R(L) ::= \{RA_1(L),\ldots,RA_n(L)\}, n > 0$
  - if the buffers in $RA_1(L)$ and $RA_n(L)$ are disjoint
- rule action: $RA(L) ::= a(b,t,\{SA_1(L,t_1),\ldots,SA_n(L,t_n)\}), n \geq 0$
- slot action: $SA(L,t) ::= (s,v)$
  - if $s \in \tau(t)$ and $v \in \text{vars}(L)$

Thereby, $b \in B, t \in T, a \in A, s \in C_A$ and $v \in V_A$. This definition is a reduced version of the definition given in [Bot, p. 17] and has only been transformed from LISP style syntax to sets of first-order terms.
4.3 INFORMAL OPERATIONAL SEMANTICS

In this section, we describe ACT-R’s operational semantics informally. The production rule system constantly checks for matching rules and applies their actions to the buffers. This means that it tests the conditions on the left hand side with the contents of the buffers (which are chunks) and applies the actions on the right hand side, i.e. modifies individual slots, requests a new chunk from a module or clears a buffer.

The left hand side of a production rule consists of buffer tests – that are terms \((b, t, P)\) with a buffer \(b\), a type \(t\) and a set of slot-value pairs \(P\). The values of a slot-value pair can be either constants or variables. The test matches a buffer, if the chunk in the tested buffer \(b\) has the specified type \(t\) and all slot-value pairs in \(P\) match the values of the chunk in \(b\). Thereby, variables of the rule are bound to the actual values of the chunk. Values of a chunk in the buffers are always ground, i.e. they do not contain any variables. This is ensured by the previously mentioned condition in the syntax of a rule that the right hand side of a rule does not introduce new variables (see Section 4.2). Hence the chunks in the buffers stay ground.

If there is more than one matching rule, a conflict resolution mechanism which depends on the sub-symbolic layer chooses one rule that is applied. After a rule has been selected, it takes a certain time (usually 50 ms) for the rule to fire. i.e. actions are applied after this delay. During that time the procedural module is blocked and no rule can match.

The right hand side consists of actions \(a(b, t, P)\), where \(a \in A\) is an action symbol, \(b\) is a constant denoting a buffer and \(P\) is again a set of slot-value pairs. We have already explained the three types of actions (modifications, requests and clearings) roughly. In more detail, a modification overwrites only the slots specified in \(P\) with the values from \(P\). A request clears the requested buffer and asks a module for a new chunk. It can take some time specified by the module (and often depending on sub-symbolic values) until the request is processed and the chunk is available. During that time, other rules still can fire, i.e. requests are executed in parallel. However, a module can only process one request for a buffer at the same time. Buffer clearings simply remove the chunk from a buffer. In the following, we disregard clearings in our definitions since they are easy to add.

As mentioned before, a rule can only consist of one action per buffer. This excludes race conditions and conflicts between actions. For instance, a modification is typically applied without any delay whereas a request typically takes some time and overwrites the chunk in the requested buffer. This would lead to losing the modification that was applied before. The modification cannot be applied after the request, as it is not clear that the requested chunk is of the same type as the chunk
that has been in the buffer before. Hence, such rules are undesired. Note that a request can still result in different chunks as requests typically are not functional but non-deterministic. In implementations, they often include a noisy component that influences the result.

In the following example, a rule is defined and its behavior is explained informally.

**Example 18 (Production Rule).** We want to model the counting process of a little child that has just learned how to count from one to ten. We use the natural number chunks described in Section 4.1 as declarative knowledge. Furthermore, we have a goal chunk of another type \( g \) that memorizes the current number in a current slot. We now define a production rule, that increments the number in the counting process (and call this rule \( \text{inc} \)). We denote variables with capital letters in our examples. The left-hand side of the rule \( \text{inc} \) consists of two tests:

• \( =((\text{goal}, g, \{(\text{current}, X)\}) \) and

• \( =((\text{retrieval}, \text{succ}, \{(\text{number}, X), (\text{successor}, Y)\}) \).

This means that the rule tests if in the goal buffer there is a chunk of type \( g \) that has some number \( X \) (which is a variable) in the current slot. If this number \( X \) is also in the number slot of the chunk in the retrieval buffer, the test succeeds and the variable \( Y \) is bound to the value in the successor slot. The actions of the rule are:

• \( =((\text{goal}, g, \{(\text{current}, Y)\}) \) and

• \( +((\text{retrieval}, \text{succ}, \{(\text{number}, Y)\}) \).

The first action modifies the chunk in the goal. A modification cannot change the type, hence it has to be the same type symbol as in the test. The current slot of the goal chunk is adjusted to the successor number \( Y \) and the declarative module is asked for a chunk of type \( \text{succ} \) with \( Y \) in its number slot. This is called a retrieval request. After a certain amount of time, the declarative module will put a chunk with \( Y \) in its number and \( Y + 1 \) in its successor slot into the retrieval buffer and the rule can be applied again.
In this part of the thesis, program analysis modulo user-defined equivalence relations is introduced and combined with the existing analysis methods for (invariant-based) confluence and operational equivalence.

For this purpose, reasoning about states on the object language level, as it is typical for Constraint Handling Rules (CHR) program analysis, is formalized and justified. Then, the implications of user-defined equivalence relations for program analysis are discussed and a subset of well-behaving equivalence relations is identified (that we denote as compatible equivalence relations) for which the proposed analysis methods for confluence and operational equivalence can be applied. It is shown by the construction of a class of compatible equivalence relations – the $p$-state equivalence relations – that the defined subset of equivalence relations is meaningful in a sense that there are non-trivial equivalence relations that satisfy the restrictions of compatibility.
CHR program analysis usually uses state equivalence to distinguish and compare states. However, in practice, it might become useful to consider equivalence classes of states that are formally not equivalent with respect to state equivalence as demonstrated in the following example.

**Example 19 (Multi-Set Items [CK17]).** Consider the following small CHR program, that collects items represented in individual item/1 constraints to a multi-set represented by a constraint of the form mset(L) where L is a list of items. Hence, a multi-set is represented as a list and the program can add new items to the multi-set.

The program consists of the following rule:

\[ \text{mset}(L), \text{item}(A) \Leftrightarrow \text{mset}([A|L]) \]

that adds an item A to a multi-set represented by the list L.

In this program, the two states

- \( \langle \text{mset}([a,b]); \top; \emptyset \rangle \) and
- \( \langle \text{mset}([b,a]); \top; \emptyset \rangle \)

are not equivalent (with respect to state equivalence \( \equiv \)), although, semantically, they represent the same multi-set. As we will see in Chapter 6, this causes non-confluence of the above program which seems to be counter-intuitive.

In this chapter, the foundations for reasoning about CHR programs and states with user-defined equivalence relations are set up. Since reasoning about states is often performed in a mathematical meta-level language, the chapter first discusses how the reasoning can be moved to the object language level in CHR in Section 5.1. This step is the foundation of most CHR program analyses like confluence analysis or operational equivalence (c.f. Section 3.3) and the idea behind it will be reused in the new analysis methods for confluence modulo equivalence and operational equivalence modulo equivalence (c.f. Chapters 6 and 7).

Section 5.2 discusses the implications for typical CHR analysis methods when introducing user-defined equivalence relations. To overcome these problems, a subset of so-called *compatible equivalence relations* is identified in Section 5.3. It restricts equivalence relations such that the monotonicity property of CHR is maintained for user-defined equivalence relations and in consequence the proposed analysis results can be applied.
In Section 5.3.2 the structure of compatible equivalence relations is investigated further. It is shown that the set of compatible equivalence relations still includes some relations with possibly undesirable implications. However, all those undesirable effects can be excluded by the introduction of a meaningful class of equivalence relations in Section 5.3.3 – the \( p \)-state equivalence relations. This class is fully compatible with our analysis methods and is still powerful enough to cover many interesting applications.

The results and related work are discussed in Sections 5.4 and 5.5.

5.1 OBJECT LANGUAGE LEVEL REASONING

When working with user-defined equivalence relations, the question arises how those relations can be defined such that analysis can be simplified. Naturally, such equivalence relations are described by using a mathematical meta-language that is inspired from natural language and that allows to define patterns of CHR states together with a side condition that summarizes a set of states (called instances) where the variables in the pattern may be substituted by variables.

In this section, this meta-language concept is formalized and it is shown that for CHR it is possible to only reason about an object language level state that is formed from the pattern and the side-condition of the meta-language description. This allows to reuse all CHR analysis results to reason about sets of states defined by a mathematical meta language concept.

Furthermore, it is shown that reasoning about state transitions over sets of states can be performed directly on the meta-level. Due to the equivalence of meta-level and object language level reasoning in CHR, the object language representations of the meta-level states can also be considered instead.

Additionally, the proof method of case splitting is considered which is useful in the class of ground range-restricted CHR programs. It applies the result that meta-level reasoning over sets of states and object level reasoning are equivalent in the context of rule applications. The idea behind case splitting is that in ground CHR confluence proofs might get stuck, since the considered states are too abstract to actually apply a rule. Since groundness is assumed on those states, it can never appear that inference stops at such a state. By performing a case distinction over the values of the variables, all possible subsets of states are considered.

In this section, we first formalize reasoning on the meta-level and the object language level and show their equivalence in Section 5.1.1. We then use this formalization to show that we can either reason about state transitions by using meta-level states or their object language representations in Section 5.1.2. Eventually, this result is applied for
the class of ground range-restricted CHR programs by introducing the proof method of case splitting in Section 5.1.3.

5.1.1 Formalization

We start with a technical definition:

**Definition 29 (Constraint and State Representation of a Substitution).** Let \( \theta := \{ X_1/t_1, \ldots, X_n/t_n \} \) with \( n \geq 0 \) be a substitution. Then the constraint representation of \( \theta \) is defined as

\[
\theta^\perp := X_1 \doteq t_1 \land \ldots \land X_n \doteq t_n.
\]

The state representation of \( \theta \) is defined as

\[
\hat{\theta} := (\emptyset; \theta^\perp; \{ X_1, \ldots, X_n \}).
\]

The central definition of this section are state descriptions that are often used to describe sets of states.

**Definition 30 (Abstract State Description).** An abstract (CHR) state description is a term of the form \( \sigma \) where \( \sigma := \{ {\varphi} \mid \sigma \} \) is a CHR state with variables \( \mathbb{V} \) and \( \varphi(x) \) is some constraint with variables \( x \in \mathbb{V} \). We then call \( \sigma \) a (state) pattern and \( \varphi(x) \) a (side) condition. The variables \( \mathbb{V} \) are the free variables of the abstract state description.

A state \( [\sigma] \) is an instance of an abstract state \( \sigma \) where \( p, \) if and only if there is a substitution \( \theta \) over variables \( \mathbb{V} \) such that \( [\sigma] \circ \hat{\theta} = [\sigma'] \) and \( p(\bar{x}) \land \theta^\perp \) holds in \( [\sigma] \circ \hat{\theta} \).

Note that we restrict abstract reasoning to states with only global variables. With the merge operator from Definition 14, we can always remove the global variables when using the state in a greater context as exemplified in Examples 9 and 11. For a state with built-in constraint store \( B \), the definition of an instance implicitly requires \( B \land p(\bar{x}) \land \theta^\perp \) to be consistent, i.e. that \( [\sigma] \circ \hat{\theta} \neq [\mathbb{\bot}, \mathbb{\bot}, \mathbb{\bot}] \).

The semantics of an abstract state description is that it defines a set of actual CHR states which is exactly the set of its instances. This signifies that the variables in an abstract state description are interpreted as meta-variables which represent every possible value of the meta-variable including constant values and object level variables.

**Definition 31 (Interpretation of Abstract State Descriptions).** Let \( s \) be an abstract state description with \( \text{vars}(s) = \mathbb{V} \) and \( \theta \) a substitution over \( \mathbb{V} \). The interpretation function \( I \) of an abstract state description that maps an abstract state description and a substitution to a set of CHR states \( \subseteq \Sigma_{\text{CHR}} \) is defined as

\[
I(s, \theta) := \{ [\sigma] \circ \hat{\theta} \mid [\sigma] \circ \hat{\theta} \text{ is an instance of } s \}.
\]
The goal is to construct a state on the object language level that subsumes all states of an abstract state description.

The object level representation of an instance contains the side-condition and the variable bindings of the instance.

Equivalence of the Representations

Note that in the definition \( \hat{\theta} = \emptyset \) is a valid substitution. This set then contains all valid instances of \( s \) with no further restrictions and for all \( \theta \) it holds that \( I(s, \theta) \subseteq I(s, \emptyset) \).

The following example is inspired by [CK17, p. 22] and defines an equivalence relation on multi-sets using state descriptions as defined in Definition 30.

**Example 20.** Let \( \text{perm}(L, L') \) be a built-in constraint that is true if and only if \( L' \) is a permutation of \( L \). Then, we define \( \approx^S \) as the smallest congruence relation with respect to \( \odot_V \) such that for all instances \( \sigma_1 \) of

\[
\sigma_1 := \langle \text{mset}(L); \top; \{L, L'\} \rangle \text{ where } \text{perm}(L, L')
\]

and for all instances \( \sigma_2 \) of

\[
\sigma_2 := \langle \text{mset}(L'); \top; \{L, L'\} \rangle \text{ where } \text{perm}(L, L')
\]

it holds that \( \sigma_1 \approx^S \sigma_2 \) if and only if \( \sigma_1 \in I(s_1, \theta) \) and \( \sigma_2 \in I(s_2, \theta) \).

This means for example that

\[
[\sigma_1] := \left[\langle \text{mset}(L); L \models [a, b]; \{L, L'\} \rangle\right]
\]

\[
\approx^S \left[\langle \text{mset}(L'); L' \models [b, a]; \{L, L'\} \rangle\right] =: [\sigma_2],
\]

since \( \sigma_1 \in I(s_1, \{L/\{a, b\}\}) \) and \( \sigma_2 \in I(s_2, \{L/\{a, b\}\}) \).

Since \( \approx^S \) is defined as a congruence relation with respect to \( \odot_V \) for all \( V \), it follows that:

\[
\sigma_1' := \left[\langle \text{mset}([a, b]); \top; \emptyset \rangle\right] \approx^S \left[\langle \text{mset}([b, a]); \top; \emptyset \rangle\right] =: \sigma_2',
\]

because \( [\sigma_1] \odot_{(L, L')} [\sigma_2] = [\sigma_1'] \) and \( [\sigma_2] \odot_{(L, L')} [\sigma_2] = [\sigma_2'] \).

Note that in general, \( I(s_1, \emptyset) \) contains all states where it follows from the built-in store that \( L \) and \( L' \) are permutations of each other.

For the rest of this section it is shown how a CHR state can be constructed that contains all necessary information to reason about the whole set of instances of an abstract state description. The following definition shows the construction method of an object language representation of an instance from the abstract state description.

**Definition 32 (Object Language Representation).** Let \( \sigma_i \) be an instance of the abstract state description \( s := \sigma \text{ where } p(\bar{x}) \) with a state \( \sigma \) with global variables \( V \) and a substitution \( \theta \) over \( V \).

The CHR state \( \text{inst}(s, \theta) := \sigma \odot (\emptyset; p(\bar{x}) \land \theta^\dagger; V) \) is called the object language representation of instance \( \sigma_i \).

In the following lemma it is shown that the object language representation of an instance of an abstract state description is equivalent to the instance itself and that every object language representation derived from an abstract state description \( s \) is in fact an instance of \( s \).
Lemma 10 (Equivalence of Representations of Instances). Let $s$ be an abstract state description, $\sigma$ be an instance of $s$ with substitution $\theta := \{X_1/t_1, \ldots, X_n/t_n\}$ and $\sigma' := \text{inst}(s, \theta)$. Then the following two propositions hold:

1. $[\sigma] = [\sigma']$.
2. If $[\sigma'] \neq ([\_; \_; \_])$, then $\sigma'$ is an instance of $s$.

Proof. Let $\sigma$ be an arbitrary instance of an abstract state description $s := \langle G; B; V \rangle$ where $p(\bar{x})$, i.e. there is a substitution $\bar{\theta}$ such that $[\langle G; B; V \rangle \circ [\bar{\theta}] = [\sigma]$ and $p(\bar{x}) \land \theta^\dagger$ holds. By Definition 14, it follows that

$$[\sigma] = [\langle G; p(\bar{x}) \land \theta^\dagger \land B; V \rangle].$$

Since $p(\bar{x}) \land \theta^\dagger$ holds in $\sigma'$, it is clear that

$$CT \models \theta^\dagger \land B \iff p(\bar{x}) \land \theta^\dagger \land B \iff p(\bar{x}) \land \theta^\dagger \land B.$$

It follows that

$$[\sigma] = [\langle G; p(\bar{x}) \land \theta^\dagger \land B; V \rangle].$$

We can rewrite the resulting state to

$$[\sigma] = [\langle G; B; V \rangle \circ [\langle \varnothing; p(\bar{x}) \land \theta^\dagger; V \rangle].$$

2. Let $s := \sigma$ where $p(\bar{x})$ with $\sigma := \langle G; B; V \rangle$ be an abstract state description, $\theta$ be a substitution over $V$ and $\sigma' := \langle G; p(\bar{x}) \land B; V \rangle \circ \bar{\theta}$ a CHR state such that $p(\bar{x}) \land B \land \theta^\dagger$ is satisfied. By Definition 14, it follows that

$$[\sigma'] = [\langle G; p(\bar{x}) \land B \land \theta^\dagger; V \rangle].$$

Since $CT \models p(\bar{x}) \land B \theta \rightarrow p(\bar{x}) \land \theta^\dagger$ and $p(\bar{x}) \land B \land \theta^\dagger$ is satisfied by precondition, it is clear that $[\sigma'] = [\sigma] \circ [\langle \varnothing; p(\bar{x}); V \rangle] \circ [\bar{\theta}]$ and $p(\bar{x}) \land \theta^\dagger$ holds in $[\sigma']$. Hence, $\sigma'$ is an instance of $s$.

This result allows to move side-conditions over states that are formulated in a meta-language directly into the state. The reason for that is that the built-in constraint store in CHR is a logical conjunction of constraints and Definitions 7 and 13 consider logically equivalent constraint stores (with respect to a constraint theory) as equivalent, i.e. they are indistinguishable for the transition system. Additionally, the information of the substitution of an instance can be separated from the pattern and the side condition using the merge operator $\circ$.

Note that Lemma 10 implies that the state formed from the pattern of an abstract state description $s$, where the side condition is added to the built-in store, is always an instance of $s$ as long as the pattern and the side condition are consistent. In such states, no variables are substituted.
Corollary 1. Let $s := \sigma$ where $p(\bar{x})$ be an abstract state description with $\sigma := (G; B; \nu)$. If $p(\bar{x}) \land B$ is satisfiable, then $(G; p(\bar{x}) \land B; \nu)$ is an instance of $s$.

Proof. This follows directly from Lemma 10 with substitution $\emptyset$. □

The following technical observation shows that all transitions possible in an instance of an abstract state description are also applicable in the object language representation of the instance and vice-versa.

Corollary 2. Let $s$ be an abstract state description and let $\sigma$ be an instance of $s$. Then, $[\sigma] \rightarrow_r [\tau]$ if and only if $[\text{inst}(s, \theta)] \rightarrow_r [\tau]$.

Proof. This follows directly from Lemma 10 and Definition 13. □

Since in both Lemma 10 and Corollary 2 the instances have been chosen arbitrarily, it is clear that universally quantified propositions over instances of abstract state descriptions on the meta-language level can be considered directly on the object language level. This means that propositions of the form “For all states of the form $(G; B; \nu)$ where $p(\bar{x})$, it holds that…” can also be written as “For all $[\sigma]$ of the form $(G; X_1 = t_1 \land \cdots \land X_n = t_n; \nu)$ with $X_1, \ldots, X_n \in \nu$ and if $\langle G; p(\bar{x}) \land B; \nu \rangle \diamond [\sigma] \neq [\langle \_; \_; \_ \rangle]$, it holds that…”

5.1.2 State Transitions on the Meta-Level

For state transitions it suffices to only consider the state that is obtained by adding the side condition to the built-in store of the pattern. This is ensured by monotonicity of CHR:

Lemma 11 (Equivalence of Meta-Level and Object Level Rule Application). Let $s := \sigma$ where $p(\bar{x})$ be an abstract state description.

If $[\sigma] \diamond [(G; p(\bar{x}); B); \nu]] \rightarrow [\tau] \diamond [(G; p(\bar{x}); B); \nu]]$, then for all instances $\sigma'$ of $s$ with substitutions $\theta$ it holds that $[\sigma'] \rightarrow \tau \diamond [\hat{\theta}]$ and $[\tau] \diamond [\hat{\theta}]$ is an instance of $\tau$ where $p(\bar{x})$, i.e. $p(\bar{x})$ holds in $[\tau] \diamond [\hat{\theta}]$.

Proof. Let $s := \sigma$ where $p(\bar{x})$ be an abstract state description with $\sigma := (G; B; \nu)$. Let $[\sigma] \diamond [(G; p(\bar{x}); B); \nu]] \rightarrow [\tau]$ and let $\sigma'$ be an instance of $s$ with substitution $\theta$.

Since $\sigma'$ is an instance of $s$, Lemma 10 can be applied and therefore

$[\sigma'] = [\sigma] \diamond [(G; p(\bar{x}); B); \nu]] \diamond [\hat{\theta}]$.

Due to monotonicity (c.f. Lemma 7):

$[\sigma'] = [\sigma] \diamond [(G; p(\bar{x}); B); \nu]] \diamond [\hat{\theta}] \rightarrow [\tau] \diamond [\hat{\theta}]$.

Due to monotonicity (c.f. Lemma 7), it is clear that $[\tau] = [\delta] \diamond [(G; p(\bar{x}); B); \nu]]$ for some $[\delta]$ and hence $[\tau] \diamond [\hat{\theta}]$ is an instance of the abstract state description $\tau$ where $p(\bar{x})$. □
We can use both the abstract state description itself (meta-level) or its most general instance (object language level) to reason about rule applications.

**Example 21 (Meta-Rule Applications).** Consider the program

\[ r : c(X) \iff \text{even}(X) \mid d(X) \]

and the abstract state description

\[ s := \sigma \text{ where even}(X) \text{ for } \sigma := \langle c(X); \top; \{X\} \rangle. \]

By Lemma 11, we can apply the rule to \( \sigma \) under the assumption that \( \text{even}(X) \) holds:

\[
\begin{align*}
[\sigma] \text{ where even}(X) \\
\mapsto_{r} \left[ \langle d(X); \top; \{X\} \rangle \text{ where even}(X) \right] =: [\tau] \text{ where even}(X).
\end{align*}
\]

Here we use a notation closer to the meta-language instead of writing \([\sigma] \diamond [\langle \emptyset; \text{even}(X); \{X\} \rangle] \). This is the conclusion from Lemmas 10 and 11.

Consider the following instance of \( s \): \( \sigma' := \langle c(X); X=2; \{X\} \rangle \). We can apply the rule as follows:

\[
[\sigma'] \mapsto_{r} \left[ \langle d(X); X=2; \{X\} \rangle \right] =: [\tau'].
\]

It is clear that \( \text{even}(X) \) holds in \( \tau' \) and \( \tau' \) is an instance of \( \tau \).

Since \( [\sigma'] = [\sigma] \diamond [\langle \emptyset; X=2; \{X\} \rangle] \), we can draw this conclusion directly from Lemmas 7 and 11:

\[
[\sigma] \mapsto [\tau] \\
\Rightarrow [\sigma'] = [\sigma] \diamond [\langle \emptyset; X=2; \{X\} \rangle] \mapsto [\tau] \diamond [\langle \emptyset; X=2; \{X\} \rangle] = [\tau'].
\]

It follows that \( \tau' \) is an instance of \( \tau \) and also that \( \text{even}(X) \) holds in \( \tau' \).

As demonstrated in the above example, Lemma 11 allows to reason from the pattern and the side condition of an abstract state description \( s \) about every instance of \( s \). The lemma can be generalized from reasoning about state transitions to any monotonic proposition \( Q \) over a set of states. A proposition \( Q \) is monotonic if the following holds: If \( Q([\sigma]) \) holds, then \( Q([\sigma] \diamond [\tau]) \) for all \([\tau] \).

Note that Lemma 11 assumes the side condition of the abstract state description to be monotonic (sometimes also called purely logical), i.e. constraints like the built-in constraint \( \text{var}/1 \) of SWI-Prolog, that is true for an argument \( X \) if and only if \( X \) is not bound, are not allowed as they violate monotonicity of CHR and its built-in constraint store.
5.1.3 Case Splitting

The elegance of CHR allows for programs to be executed for non-ground states. This makes reasoning on the object language level possible by simply executing rules on the non-ground states. For the ground, range-restricted fragment of CHR, i.e. the fragment of CHR where all initial states are ground and where no rule contains local variables, reasoning about rule applications becomes more complicated. Groundness and range-restrictedness imply the invariant that every state in each derivation is ground.

**Definition 33 (Groundness Invariant \( G \)).** The groundness invariant \( G \) over CHR states is satisfied in a state \( \sigma \) if and only if all variables in \( \sigma \) are ground.

It directly follows that \( G \) is an invariant in any range-restricted program.

**Corollary 3 (\( G \) is an Invariant [Frü09, p. 142]).** \( G \) is an invariant in any range-restricted CHR program.

For reasoning in the ground range-restricted fragment, it is still possible to execute the program on a non-ground state and every derivation in such a non-ground state is also possible in a ground state that is consistent with the built-in store. However, it might be possible that some rules are not applicable in the incomplete state, although they would be for any ground state. This is shown in the following example.

**Example 22.** Consider the following CHR program with built-in constraints odd\((X)\) and even\((X)\) that are true if and only if \( X \) is a number and is odd or even, respectively. The built-in constraint number\((X)\) is true, if and only if \( X \) is a number.

\[
\begin{align*}
    a_1 : a(X) & \iff b(X). \\
    a_2 : a(X) & \iff \top. \\
    b_1 : b(X) & \iff \text{even}(X) \mid \top. \\
    b_2 : b(X) & \iff \text{odd}(X) \mid \top. \\
    b_3 : b(X) & \iff \neg \text{number}(X) \mid \top.
\end{align*}
\]

The program is range-restricted. According to the classical confluence analysis techniques as summarized in Section 3.3.5, the program has the overlap state \( \langle a(X); \top; \{X\} \rangle \). The critical pair consists of the states \( \langle b(X); \top; \{X\} \rangle \) and \( \langle \emptyset; \top; \{X\} \rangle \equiv \sigma_\emptyset \). The two states are not equivalent and in none of them a rule is applicable. Therefore the critical pair is not joinable and the program is not confluent (for plain CHR).

The witness of this observation is the overlap state \( \langle a(X); \top; \{X\} \rangle \) itself, that leads to two non-joinable states that are both part of the transition system. However, when ground CHR is assumed, then every possible input leads to the state \( \sigma_\emptyset \), for instance:
Case splitting can be used to continue an execution that is stuck in an incomplete state by adding additional built-in constraints.

The above example derivations are possible for all ground terms. This means that the program is joinable for all observable inputs. Nevertheless, in plain CHR this example is actually non-joinable as the two non-ground states from the critical pair are part of the transition system and obviously not joinable.

However, if the program is assumed to be executed in ground CHR, then the two states from the critical pair are only placeholders for a (possibly infinite) set of ground states. In each of those ground states, the guards of the example program are decidable and therefore one of the rules will be applicable. Hence, the methods of plain CHR cannot be transferred directly to ground, range-restricted CHR, since they do not process all available information to decide if the two states are joinable.

The proof method of case splitting can be used to reason about such states. The idea is to assume two cases for every rule: One where the guard holds and one where the guard does not hold. Those assumptions on the state can be included as side conditions into the abstract state description of the states (or as shown above directly as built-in constraints into the states themselves). If in all cases the property that has to be shown is satisfied – in our example: If in all cases the two states are joinable –, then it holds for all (ground) states.

Case splitting has been used informally in some confluence proofs. For instance, Frühwirth describes a critical pair for the canonical greatest common divisor program, where “computation cannot proceed until more about the relationship between [. . . the variables] is known” [Frü09, p. 180]. The conclusion then is that the program is only confluent for the groundness invariant, but not in general. A similar example is the canonical exchange sort algorithm [Frü09, p. 184]. A formalization of case splitting is proposed in [CK17, pp. 26–27] using a meta-level semantics of CHR. We now show how case splitting is possible on the object language level by using Lemmas 10 and 11.

We start by defining a monotonic variant of the built-in constraint ground/1. In typical implementations, ground/1 is defined in a non-monotonic way. For instance, consider the state

\[
\langle c(X); X\oplus a \land \text{ground}(X); \{X\}\rangle.
\]

This state is equivalent to \([\langle c(X); X\oplus a; \{X\}\rangle]\), because ground(X) is satisfied in the constraint store.

However, consider the state \([\langle c(X); \text{ground}(X); \{X\}\rangle]\). In most implementations, this state is equivalent to the failed state \([\_; \bot; \_]\), because the current constraint store does not entail \text{ground}(X).
For our purpose of defining a groundness invariant, we do not want to check, if the ground(X) constraint is entailed by the built-in constraint store, but rather add the information, that X is ground. This means that we want to maintain the ask and tell nature of monotonic constraints, where we can tell that a constraint holds by adding it to the store. If we ask for a constraint (by checking it in the guard), the computation does not fail, if the constraint is not entailed. The computation rather waits until the constraint is entailed by the current constraint store. In that case, it chooses its candidate non-deterministically. This follows from the operational semantics of CHR (c.f. Definition 12). The relationship of ask and tell constraints is explained for the rule-based formalism of concurrent constraint programming in [Frü09, pp. 167–168].

**Definition 34 (Monotonic Version of ground/1).** The built-in constraint ground/1 is defined as follows:

For some ground term a,

\[ CT \models B \rightarrow X= a \]

if and only if

\[ CT \models B \rightarrow \text{ground}(X), \]

where B is the built-in constraint store of a state.

This means that ground(X) is equivalent to the proposition that X has a ground value. Note that the definition can be read in both directions. For instance, consider the state \( \sigma := \langle \emptyset; \text{ground}(X); \{X\} \rangle \).

The state implies that there is some \( a \) such that \( X = a \). It holds that

\[
[\sigma] \diamond [\langle \emptyset; X = a; \{X\} \rangle] = \left[ \langle c(X); \text{ground}(X) \land X = a; \{X\} \rangle \right] = \left[ \langle c(X); X = a; \{X\} \rangle \right].
\]

In other words: Once the information is added to a state that variable X is ground, either by adding \( \text{ground}(X) \) or \( X = a \) to the built-in constraint store, it cannot be removed. Since binding X to some ground term implies \( \text{ground}(X) \), all extensions of the state that are consistent themselves, consistently contain \( \text{ground}(X) \). Hence, this version of ground/1 is monotonic.

We now show that for the groundness invariant \( \mathcal{G} \), there is a unique minimal extension.

**Lemma 12 (Minimal Extension of \( \mathcal{G} \)).** For all states \( \sigma \), where the groundness invariant \( \mathcal{G} \) is not satisfied for an arbitrary variable X, it holds that \( \mathcal{M}^\mathcal{G}(\sigma) = \{ \sigma_g \} \), where \( \sigma_g := \langle \emptyset; \text{ground}(X); \{X\} \rangle \).

**Proof.** Let \( \sigma \) be a state where \( \mathcal{G} \) is not satisfied for an arbitrary variable X. The set of extensions that reestablish \( \mathcal{G} \) is then defined as:

\[
\Sigma^\mathcal{G}(\sigma) := \{ \langle \emptyset; X = a; \{X\} \rangle \diamond \sigma' \}
\]

for all ground terms \( a \) and all states \( \sigma' \).
Since $[\langle \varnothing; X = a; \{ X \} \rangle] = [\sigma_8] \circ [\langle \varnothing; X = a; \{ X \} \rangle]$, it holds by Lemma 6 that
\[ [\sigma_8] \prec [\langle \varnothing; X = a; \{ X \} \rangle] \text{ for all ground terms } a. \]

Hence, $\sigma_8$ is a $\prec$-minimal element of $\Sigma^{G}(\sigma)$ for all states $\sigma$ where $G$ does not hold and therefore:
\[ M^{G}(\sigma) = \{ [\sigma_8] \}. \]

Since $X$ is an arbitrary variable, the result can be generalized accordingly for multiple variables that do not satisfy the invariant $G$.

Lemma 12 allows us to use the invariant-based confluence test from Theorem 7 by simply extending every overlap state with the built-in constraint ground($X$) for all variables $X$.

**Corollary 4.** Let $P$ be a CHR program with groundness invariant $G$. Then, $P$ is confluent with respect to $G$ if and only if for all overlap states $\sigma$ with critical pairs $(\sigma_1, \sigma_2)$ it holds that $[\sigma_1 \circ \sigma_8] \downarrow [\sigma_1 \circ \sigma_8]$ for $\sigma_8 := \langle \varnothing; \text{ground}(X_1) \land \ldots \land \text{ground}(X_n); \{ X_1, \ldots, X_n \} \rangle$ where the $X_i (1 \leq i \leq n)$ are the variables for which $G$ is not satisfied.

**Proof.** This follows directly from Theorem 7 and Lemma 12.

We now informally explain case splitting by the continuation of our running example under the application of Corollary 4 and Lemmas 10 and 11.

**Example 23.** We revisit Example 22. The overlap $\sigma := \langle a(X); \top; \{ X \} \rangle$ leads to the critical pair $(\langle b(X); \top; \{ X \} \rangle, \varnothing)$. In classic confluence analysis for non-ground programs, $\langle b(X); \top; \{ X \} \rangle$ is a final state since no rule is applicable, because $X$ is neither even nor odd nor not a number.

However, if we assume ground CHR, one of the rules $b_1, b_2$ or $b_3$ must be applicable, since for ground states we can decide if they are a number, even or odd. We now show $G$-confluence of the program by applying Corollary 4. This means that for the overlap state $\sigma$, where $G(\sigma)$ does not hold, we have to consider the state $\sigma' := \sigma \circ \sigma_8 = \langle a(X); \text{ground}(X); \{ X \} \rangle$ for $\sigma_8 := \langle \varnothing; \text{ground}(X); \{ X \} \rangle$. The new overlap $\sigma'$ contains the groundness invariant, but is still a final state, since none of the rules are applicable. This is the case because $CT \not\models \text{ground}(X) \leftrightarrow p(X)$ for $p/1 \in \{ \text{even}, \text{odd}, \neg\text{number} \}$.

For this purpose, we perform a case distinction over $X$ to make one of the rules applicable. The aim is to find one derivation, that leads to $[\sigma_{\varnothing}]$ in all cases (since this is the final state for the second component of the critical pair). The condition of the case distinction can be added as a constraint to the built-in constraint store according to Lemmas 10 and 11. The idea is depicted in Fig. 5.1.1. There, we abbreviate $\text{ground}(X)$ with $g(X)$.

For instance, we can assume that $X$ is a number and even by adding the built-in constraint even$(X)$ to the built-in store and obtain the state
In this state, the rule \( b_1 \) is applicable which leads to the final state \([\sigma_\emptyset]\). Additionally, we now have to consider the opposite case where \( X \) is not even. By adding \(-\text{even}(X)\) to the built-in store, we obtain the state \([\langle b(X); \text{ground}(X) \land \neg \text{even}(X); \{X\}\rangle]\). In this state, no rule is applicable. Hence, we have to perform a nested case distinction. We want to make the rule \( b_3 \) applicable by adding \(-\text{number}(X)\) to the built-in constraint store. The opposite case can be obtained by adding \(\text{number}(X)\) to the built-in constraint store. For reasons of space, the nested case distinction is illustrated in Fig. 5.2.

Again, we abbreviate \(\text{ground}(X)\) with \(g(X)\).

\[
\begin{align*}
\text{Figure 5.2: Confluence diagram with case distinctions for the built-in constraints (cont.).}

In the first case, we obtain the state
\[
[\langle b(X); \text{ground}(X) \land \neg \text{even}(X) \land \neg \text{number}(X); \{X\}\rangle].
\]
In this state, the rule $b_3$ is applicable leading to the final state $[\sigma_\emptyset]$. The opposite case is the state

$$\langle b(X); \text{ground}(X) \land \neg\text{even}(X) \land \text{number}(X); \{X\} \rangle.$$ 

This state is equivalent to $[\langle b(X); \text{odd}(X); \{X\} \rangle]$ due to the constraint theory of the built-in constraints and Definition 7.2. In this state, the rule $b_2$ is applicable, leading to the final state $[\sigma_\emptyset]$.

In all possible cases there exists a transition $[\sigma]$ to $[\sigma_\emptyset]$ and hence, the critical pair for overlap $\sigma$ is joinable and the program is confluent in ground CHR.

Example 23 already indicates that case splitting may lead to nested case distinctions. In total, many cases might to be considered and the confluence diagram might become large and complicated. Depending on the built-in constraints it even is possible that the method does not terminate leading to an endless recursion of case distinctions.

However, the method is sound: Since we always consider a case and its opposite in each case distinction, there are no cases left out. Hence, if the individual cases all lead to joinable states, it follows that for all possible ground states the critical pair is joinable.

Note that it may appear that in a case distinction the built-in store might become inconsistent by adding a built-in constraint and hence the resulting state is equivalent to $[\langle_; \bot_; \rangle]$. Such cases can be ignored in confluence analysis, since there are no ground representatives that satisfy their built-in constraints and the failed state itself is equivalent to all other states with an inconsistent built-in constraint store.

Finally, remember that case splitting only comes into play, if a groundness invariant is assumed for states. In plain CHR the non-ground states are part of the transition system and therefore are witnesses for cases where the property to show does not hold. Note that the problem is not restricted to program analysis on the object language level as introduced above, but occurs both on the object language level and the meta-language level.

In our definition of an abstract state description, the non-ground states are included in the interpretation (c.f. Definition 31). This is also taken into account in the object level reasoning.

### 5.2 Implications for Program Analysis

As it is the case for invariants, extending program analysis with user-defined equivalence relations, introduces some benefits but also some problems that do not occur in plain confluence analysis. In this section, those problems are discussed and a restriction of equivalence relations is motivated that makes program analysis modulo user-defined equivalence relations manageable. In the following, state equivalence is referred to by $\equiv$ or by the corresponding equivalence class brackets $[\cdot]$. 

For program analysis with user-defined equivalence relations, some restrictions have to be considered.
The symbol \( \approx \) denotes some general user-defined equivalence relation that is potentially different from \( \equiv \) (but is not required to be).

In general, CHR program analysis often makes use of the idea of exploiting monotonicity of CHR to reason from small states that come from the rules in the program over all states. However, monotonicity can be broken by user-defined equivalence relations. This means that in general for two states with \( [\sigma] \approx [\sigma'] \), it is possible that an extension with \( [\tau] \approx [\tau'] \) leads to states that are not equivalent, i.e. \( [\sigma] \circ_V [\tau] \not\approx [\sigma'] \circ_V [\tau'] \) as shown in the following example.

**Example 2.4.** We construct an equivalence relation that breaks monotonicity. Let \( \#c : \Sigma \to \mathbb{N}_0 \) be a function that returns the number of constraints \( c \) in the goal store of a state. We separate the CHR state space \( \Sigma_{\text{CHR}} \) into two disjoint subsets:

\[
\Sigma_1 := \{ [\sigma] \mid \#c(\sigma) < 3 \}, \\
and \Sigma_2 := \{ [\sigma] \mid \#c(\sigma) \geq 3 \}.
\]

The partition of the state space clearly defines an equivalence relation \( \approx \) with equivalence classes \( \Sigma_1 \) and \( \Sigma_2 \).

Let \( [\sigma_1] = [(c; T; \emptyset)] \) and \( [\sigma_2] = [(c, c; T; \emptyset)] \). Since \( [\sigma_1], [\sigma_2] \in \Sigma_1 \), it holds that \( [\sigma_1] \approx [\sigma_2] \). Let \( [\tau] = [(c; T; \emptyset)] \). If we extend the two states by \( [\tau] \), the extended states are not equivalent any more:

\[
[\sigma_1] \circ [\tau] = [(c, c; T; \emptyset)] \in \Sigma_1, \\
but [\sigma_2] \circ [\tau] = [(c, c, c; T; \emptyset)] \in \Sigma_2.
\]

Hence, although \( [\sigma_1] \approx [\sigma_2] \), \( [\sigma_1] \circ [\tau] \not\approx [\sigma_2] \circ [\tau] \). This is not necessarily a problem for program analysis based on rule states: By showing a property for a rule state and all its equivalent states, we capture too many larger states due to the above observation.

However, consider the converse case: Let \( [\sigma_3] = [(c, c; T; \emptyset)] \in \Sigma_2 \). Then \( [\sigma_2] \not\approx [\sigma_3] \). However, if the two states are extended by \( [\tau] \), we get

\[
[\sigma_2] \circ [\tau] = [(c, c, c; T; \emptyset)] \in \Sigma_2, \\
and [\sigma_3] \circ [\tau] = [(c, c, c; T; \emptyset)] \in \Sigma_2.
\]

Hence, \( [\sigma_2] \circ [\tau] \approx [\sigma_3] \circ [\tau] \), although \( [\sigma_2] \not\approx [\sigma_3] \). This is critical for program analysis based on rule states: Now it is not possible any more to use a rule state and its equivalent states to reason about all larger states as we miss some of those larger states by this attempt that we cannot capture by extension.

### 5.3 Compatibility of Equivalence Relations

To ensure monotonicity in the context of equivalence relations, we need equivalence to be maintained by the merge operator. We call this property compatibility with respect to \( \circ_V \) for all \( V \). We also introduce
a restricted form of compatibility called $\Sigma$-compatibility with respect to $\diamondsuit_V$ for some set $\Sigma \subseteq \Sigma_{CHR}$.

We first define the notion of compatibility in Section 5.3.1. We continue with some observations on the proposed restricted class of equivalence relations that allow for exploiting monotonicity in Section 5.3.2. Finally, a non-trivial class of equivalence relations with interesting applications is defined that satisfies the restrictions of compatibility.

5.3.1 Definition

The first requirement of compatibility is that equivalent states in a merge expression can be interchanged freely. The equivalence relation is then called a congruence relation with respect to the merge operator.

**Definition 35 (Congruence Relation).** An equivalence relation $\approx \subseteq A \times A$ is called a congruence relation with respect to an operator $\bullet : A \times A \to A$ if and only if for all $x, x', y, y'$: If $x \approx x'$ and $y \approx y'$ then $x \bullet y \approx x' \bullet y'$.

Congruence relations can be found in many practical examples, e.g. arithmetic equivalence is a congruence relation with respect to the ring of congruence classes or the ring of real numbers with the usual definitions of addition and multiplication. In the following, we are interested in congruence relations with respect to the merge operator:

**Definition 36 (Congruence Relation w. r. t. $\diamondsuit_V$).** An equivalence relation $\approx \subseteq \Sigma_{CHR} \times \Sigma_{CHR}$ is called a congruence relation with respect to the merge operator $\diamondsuit_V$ if and only if for all $\sigma, \sigma', \tau, \tau'$: If $\sigma \approx \sigma'$ and $\tau \approx \tau'$ then $\sigma \diamondsuit_V \tau \approx \sigma' \diamondsuit_V \tau'$ for all $V$.

Unfortunately, this does not suffice to reason from rule states about any other state. It must be ensured that if two states $[\sigma]$ and $[\sigma']$ are equivalent and $[\sigma]$ can be decomposed into two parts, then $[\sigma']$ must be decomposable into two parts that are equivalent to the decomposition of $[\sigma]$. The idea of many proofs is to decompose a complex state into a rule state and some rest. To ensure that this decomposition exists, we often require the split property.

**Definition 37 ($\Sigma$-Split Property).** An equivalence relation $\approx \subseteq \Sigma_{CHR} \times \Sigma_{CHR}$ has the $\Sigma$-split property with respect to $\diamondsuit_V$ for a set of states $\Sigma \subseteq \Sigma_{CHR}$ and all $V$ if and only if for all $\sigma, \sigma_2, \sigma' \in \Sigma_{CHR}$ and for all $\sigma_1 \in \Sigma$: If $[\sigma] = [\sigma_1] \diamondsuit_V [\sigma_2]$ for some $V$ and $[\sigma] \approx [\sigma']$ then $\exists \sigma_1', \sigma_2'$ such that $[\sigma_1] \approx [\sigma_1'], [\sigma_2] \approx [\sigma_2']$ and $[\sigma'] = [\sigma_1'] \diamondsuit_V [\sigma_2']$.

If $\Sigma = \Sigma_{CHR}$, then the equivalence relation $\approx$ has the full split property with respect to $\diamondsuit_V$.

The split property assumes a syntactic relation between two states that are equivalent under an equivalence relation. If a state can be split
into two parts and is equivalent to another state, this state can be split into equivalent parts.

For congruence relations that have the $\Sigma$-split property, all extensions of the states in $\Sigma$ can be added to the set $\Sigma$, i.e. the congruence relation does not only have the split property for all states in $\Sigma$, but also for all extensions of $\Sigma$.

**Lemma 13 (\(\Sigma\)-Split Property).** Let $\approx$ be a congruence relation with respect to $\diamond_V$ that has the $\Sigma$-split property for some set of states $\Sigma \subseteq \Sigma_{CHR}$. For all $\tau \in \Sigma_{CHR}$ it holds that $\approx$ has the $\{[\sigma] \diamond [\tau] \mid \sigma \in \Sigma\}$-split property.

**Proof.** This follows directly from Definitions 35 and 37.

Note that the $\{[\sigma] \diamond [\tau] \mid \sigma \in \Sigma\}$-split property for all $\tau \in \Sigma_{CHR}$ includes the $\Sigma$-split property by setting $\tau = \emptyset$.

**Split Monotonicity**

The split property only requires that for every partition of a state there exists some partition of all equivalent states where the individual parts of both partitions are equivalent to each other. The following property extends this requirement such that if there are partitions of two equivalent states where one of the individual parts in the partition is already equivalent to the corresponding part in the partition of the equivalent state, then the second parts of the partitions must be equivalent as well. Section 5.3.2 discusses, among other things, the consequences of loosening this requirement.

**Definition 38 (\(\Sigma\)-Split Monotonicity of $\approx$).** An equivalence relation $\approx \subseteq \Sigma_{CHR} \times \Sigma_{CHR}$ is $\Sigma$-split monotonic for a set of states $\Sigma \subseteq \Sigma_{CHR}$ if and only if for all $\sigma, \sigma_2, \sigma', \sigma'_2 \in \Sigma_{CHR}$ and for all $\sigma_1 \in \Sigma$ it holds that if $[\sigma] \approx [\sigma']$, $[\sigma] = [\sigma_1] \diamond_V [\sigma_2]$, $[\sigma'] = [\sigma'_1] \diamond_V [\sigma'_2]$ and $[\sigma_1] \approx [\sigma'_1]$, then $[\sigma_2] \approx [\sigma'_2]$.

If $\Sigma = \Sigma_{CHR}$, then the equivalence relation $\approx$ is fully split monotonic.

**Compatibility**

**Definition 39 (Compatibility).** An equivalence relation $\approx$ is called to be $\Sigma$-compatible with respect to the merge operator $\diamond_V$, if it is a $\Sigma$-split monotonic congruence relation with respect to $\diamond_V$ that has the $\Sigma$-split property. If $\Sigma = \Sigma_{CHR}$, then the equivalence relation $\approx$ is fully compatible.

We also say that $\Sigma \diamond_V$-compatible. If they are clear from the context, we may omit $\Sigma$, $\diamond_V$ or both of them.

Note that $\diamond_V$-compatibility means compatibility to the merge operator for all $V$.

For our work on confluence modulo equivalence and operational equivalence modulo equivalence in Chapters 6 and 7 the user-defined equivalence relations are required to be at least congruence relations with respect to the merge operator $\diamond_V$ or even $\Sigma$-compatible with respect to $\diamond_V$. The set $\Sigma$ is defined by the rule states of the program of interest.
At first glance, the split property seems to introduce some assumptions on the cardinality of user-defined constraints, i.e. the presumption that two equivalent states must have the same number of user-defined constraints seems to be reasonable. In the following example it is shown that a $\diamond V$-compatible equivalence relation can be constructed that violates this assumption.

**Example 25** (Cardinality of Constraints). Consider a CHR system with user-defined constraints $a, b, c$ and $x$ and built-in constraints $\top, \bot$ and $\vDash$.

We want to construct a $\Sigma \diamond V$-compatible equivalence relation for $\Sigma := \{([C; T; \emptyset]) \mid C$ is a user-defined constraint$\}$. Furthermore, the equivalence $[(x; T; \emptyset)] \approx [(a, b; T; \emptyset)]$ shall be true. By congruence, it follows that every extension of $[(x; T; \emptyset)]$ is also equivalent to the same extension of $[(a, b; T; \emptyset)]$. Hence, the following equivalences hold:

- $[(x, c; T; \emptyset)] \approx [(a, b, c; T; \emptyset)]$ due to reflexivity of $\approx$, i.e.
  $$[(c; T; \emptyset)] \approx [(c; T; \emptyset)].$$

- $[(x, c, a, b; T; \emptyset)] \approx [(a, b, c, a, b; T; \emptyset)] \approx [(a, b, c, x; T; \emptyset)].$

Since

$$[(a, b; T; \emptyset)] = [(a; T; \emptyset)] \circ [(b; T; \emptyset)]$$

and

$$[(x; T; \emptyset)] \approx [(a, b; T; \emptyset)],$$

it follows from the split property of $\approx$ that there exists a partition

$$[(x; T; \emptyset)] = \lbrace \emptyset \rbrace \circ \lbrace \emptyset \rbrace$$

with $\emptyset \approx [(a; T; \emptyset)]$ and $\emptyset \approx [(b; T; \emptyset)]$. This can be handled by introducing either $\emptyset \approx [\emptyset]$ or $\emptyset \approx [\emptyset]$, i.e. either $\emptyset \approx [(a; T; \emptyset)]$ or $\emptyset \approx [(b; T; \emptyset)]$. We decide for $\emptyset \approx [(a; T; \emptyset)]$ and hence

$$[(x; T; \emptyset)] \approx [(b; T; \emptyset)].$$

For instance, the following equivalences hold:

- $[(a, b, c; T; \emptyset)] \approx [(x, c; T; \emptyset)]$ by replacing the constraints $a, b$ by $x$ (possible due to the congruence relation property).
There are equivalence relations that satisfy equivalent states are called zipping equivalence relations.

Hence, it can be seen that with the minimal equivalences

- \([\langle a, b, c; T; \emptyset \rangle] \approx [\langle a, x, c; T; \emptyset \rangle] \approx [\langle x, c; T; \emptyset \rangle] \approx [\langle a; T; \emptyset \rangle]\) (both supported by the congruence relation property).
- \([\langle a, b, c; T; \emptyset \rangle] \approx [\langle b, c; T; \emptyset \rangle] \approx [\langle x, c; T; \emptyset \rangle]\) by first replacing \(a\) by the empty state and then \(b\) by \(x\).

The last example demonstrates that there are \(\odot_{\Sigma}\)-compatible equivalence relations that allow for different cardinalities of user-defined constraints in two equivalent states. However, the split property then requires some counter-intuitive equivalences to ensure its conditions that every two equivalent states can be split into equivalent parts. In the following, we call such relations zipping equivalence relations.

**Definition 40** (Zipping Equivalence Relation). An equivalence relation \(\approx: \Sigma_{\text{CHR}} \times \Sigma_{\text{CHR}}\) is called a zipping equivalence relation if and only if it contains at least one element of the form \([\langle G; \_ , \_ \rangle] \approx [\langle G'; \_ , \_ \rangle]\) where \(|G| \neq |G'|\).

Zipping \(\odot_{\Sigma}\)-compatible equivalence relations appear to be hard to understand and are therefore not recommended for modeling.

Another problem occurs for congruence relations that have the \(\Sigma\)-split property, but are not \(\Sigma\)-split monotonic. In the following it is shown that such equivalence relations introduce a form of recursive chains of equivalences of the form \([\sigma] \approx [\sigma] \odot \rho \approx [\sigma] \odot \rho \odot \rho \approx \ldots\) for some states \([\sigma]\).

**Lemma 14** (Non-Split Monotonic Relation). Let \(\approx: \Sigma_{\text{CHR}} \times \Sigma_{\text{CHR}}\) be a congruence relation that has the split property. Let \([\sigma]\) be a state with partition \([\sigma] = [\sigma_1] \odot [\sigma_2]\). Let \([\sigma'] \approx [\sigma]\) be a state with partition \([\sigma'] = [\hat{\sigma}_1] \odot [\hat{\sigma}_2]\) where \([\hat{\sigma}_1] \approx [\sigma_1]\), but \([\hat{\sigma}_2] \neq [\sigma_2]\).

Then there exists a state \([\rho] \neq [\sigma_0]\) such that \([\sigma_1] \approx [\sigma_1] \odot [\rho] \approx [\sigma_1] \odot [\rho] \odot [\rho] \approx \ldots\) or \([\sigma_2] \approx [\sigma_2] \odot [\rho] \approx [\sigma_2] \odot [\rho] \odot [\rho] \approx \ldots\)

Proof: Let \([\sigma] = [\hat{\sigma}_1] \odot [\hat{\sigma}_2]\) and \([\sigma'] = [\hat{\sigma}_1] \odot [\hat{\sigma}_2]\) where \([\hat{\sigma}_1] \approx [\sigma_1]\) and \([\hat{\sigma}_2] \neq [\sigma_2]\). Due to the split property, there exists a partition \([\sigma'] = [\sigma'_1] \odot [\sigma'_2]\) such that \([\sigma'_1] \approx [\sigma_1]\) and \([\sigma'_2] \approx [\sigma_2]\).

Since \([\sigma'_1] \approx [\sigma'_1] \odot [\sigma'_2]\), it is clear that \([\sigma'_1] \neq [\sigma'_2]\), because otherwise reflexivity of \(\approx\) would be violated. In fact, the two partitions overlap. There are two cases:
1. \([\hat{\sigma}_1] \not< [\sigma_1]\).

It follows that there is some state \([\rho]\) such that \([\sigma_1] = [\hat{\sigma}_1] \odot [\rho]\) and \([\hat{\sigma}_2] = [\rho] \odot [\sigma_2]\), i.e.

\[
[\sigma] = [\hat{\sigma}_1] \odot [\rho] \odot [\sigma_2].
\]

Due to the congruence relation property of \(\approx\), it holds that

\[
[\sigma_1] = [\hat{\sigma}_1] \odot [\rho] \\
\approx [\sigma_1] \odot [\rho] \quad \text{(congruence relation and } [\sigma_1] \approx [\hat{\sigma}_1]) \\
\approx [\sigma_1]. \quad \text{(reflexivity and transitivity of } \approx) 
\]

Hence, due to transitivity, the following equivalences hold:

\[
[\sigma_1] \approx [\sigma_1] \odot [\rho] \approx [\sigma_1] \odot [\rho] \odot [\rho] \approx \ldots
\]

2. \([\hat{\sigma}_1] \not< [\sigma_1].\)

The proof is analogous. It follows that \([\hat{\sigma}_1] = [\sigma_1] \odot [\rho] \approx [\hat{\sigma}_1] \odot [\rho].\)

\[\square\]

**Example 26 (Recursive Relation).** In this example, the construction method of the proof of Lemma 14 is used to define a congruence relation \(\approx\) that has the split property and allows for infinite transitive extension with a state \([\rho] \neq [\sigma_0]\). We consider a CHR system with user-defined constraints \(a, b, c, d, x, y, z\) and built-in constraints \(\top, \bot, \doteq\).

Let

\[
[\sigma] = [(a, b; \top; \emptyset)] \odot [(c, d; \top; \emptyset)] \quad \text{and} \\
[\sigma'] = [(x; \top; \emptyset)] \odot [(y, z, d; \top; \emptyset)]
\]

be two \(\approx\)-equivalent CHR states with \([(a, b; \top; \emptyset)] \approx [(x; \top; \emptyset)]\), but \([(c, d; \top; \emptyset)] \not\approx [(y, z, d; \top; \emptyset)]\).

Due to the congruence relation property and reflexivity of \(\approx\), it follows that \([(c; \top; \emptyset)] \not\approx [(y, z; \top; \emptyset)]\).

Due to the split property, there exists a partition \([\sigma'] = [\sigma'_1] \odot [\sigma'_2]\) with \([\sigma'_1] \approx [(a, b; \top; \emptyset)]\) and \([\sigma'_2] \approx [(c, d; \top; \emptyset)]\). The two states \([\sigma'_1]\) and \([\sigma'_2]\) must be chosen such that the partition is different from above, since the second part of the partition cannot be equal and not equal to \([(c, d; \top; \emptyset)]\) at the same time. We choose \([\sigma'_1] = [(x, y; \top; \emptyset)]\) and \([\sigma'_2] = [(z, d; \top; \emptyset)]\). At this point a cycle is introduced to the equivalence relation and it follows that

- \([(x; \top; \emptyset)] \approx [(a, b; \top; \emptyset)] \approx [(x, y; \top; \emptyset)] \approx [(x, x, y; \top; \emptyset)] \approx \ldots\) and
- \([(c; \top; \emptyset)] \approx [(z; \top; \emptyset)].\)
Note that to satisfy the split property, the equivalence relation has to include according equivalences for the split

\[(a; T; \emptyset) \diamond (b; T; \emptyset) = (a, b; T; \emptyset) \approx (x; T; \emptyset)\]

as in Example 25, e.g. by introducing \[(a; T; \emptyset) \approx [\sigma_{\emptyset}]\] or \[(b; T; \emptyset) \approx [\sigma_{\emptyset}]\].

The class of \(p\)-state equivalence relations according to Definition 42 contains neither zipping nor recursive equivalence relations and hence the aforementioned problems do not occur.

5.3.3 Class of \(p\)-State Equivalence Relations

The analyses modulo equivalence defined in the following chapters rely on congruence or even \(\Sigma\)-split compatibility of the user-defined equivalence relations with respect to the merge operator \(\diamond V\). The restriction on this subset of equivalence relations gives rise to the question if the analysis methods are applicable for non-trivial equivalence relations. For this reason, a class of fully \(\diamond V\)-compatible equivalence relations is developed in this section that satisfies all preconditions of our analysis results.

We start by defining some desired properties of the equivalence relation in our running example of multi-sets (c.f. Example 19) in Section 5.3.3.1 in addition to \(\diamond V\)-compatibility. We then define the equivalence relation such that it meets these requirements. Then, the example is generalized in Section 5.3.3.2, where a construction method for compatible equivalence relations is defined. The result is a class of equivalence relations compatible to our analysis results. The class allows to define side-conditions on states as built-in constraints and direct replacements of user-defined constraints. This makes the class flexible to be used in practice.

5.3.3.1 Construction of a Scheme for Compatible Equivalence Relations

In this section, an equivalence relation for states over multi-sets is defined as a continuation of Example 19. We start by exemplifying some desired properties of this equivalence relation.

**Example 27** (Requirements for Multi-Set Equivalence). Example 19 is continued by introducing the following equivalence relation \(\approx^S\) that considers permutations of arguments in \(\text{mset}/1\) constraints as equivalent. In more detail, we want to construct an equivalence relation where the following equivalences are imposed:

1. \([\text{mset}([1, 2]); T; \emptyset] \approx^S [\text{mset}([2, 1]); T; \emptyset]\)
2. \([\text{mset}([a, b]), \text{mset}([c, d]), \text{item}(e); T; \emptyset] \approx^S [\text{mset}([b, a]), \text{mset}([d, c]), \text{item}(e); T; \emptyset]\)
3. \([\langle \text{item}(a); T; \emptyset \rangle] \approx^S [\langle \text{item}(a); T; \emptyset \rangle]\).

However, the following states are not equivalent:

4. \([\langle \text{mset}([1, 2]); T; \emptyset \rangle] \not\approx^S [\langle \text{mset}([1]); T; \emptyset \rangle], \) because \([1]\) is not a permutation of \([1, 2]\).

5. \([\langle \text{mset}([a, b]), \text{item}(c); T; \emptyset \rangle] \not\approx^S [\langle \text{mset}([a, b]), \text{item}(d); T; \emptyset \rangle],\) because the constraint \text{item}(c) does not have a partner in the second state.

6. \([\langle \text{mset}([a, b]), \text{mset}([b, a]); T; \emptyset \rangle] \not\approx^S [\langle \text{mset}([a, b]); T; \emptyset \rangle]\) because there is only one \text{mset} constraint on the right hand side.

For states with unbound variables, it should be possible to share information between two states through global variables. Let \text{perm}(X, Y) be a built-in constraint that is true, if and only if \(X\) is a permutation of \(Y\).

7. \([\langle \text{mset}(L); T; \{L\} \rangle] \approx^S [\langle \text{mset}(L'); \text{perm}(L, L'); \{L\} \rangle], \) because the local variable \(L'\) is a permutation of \(L\) and the free variable \(L\) is shared in both states.

8. \([\langle \text{mset}(L); T; \{L\} \rangle] \not\approx^S [\langle \text{mset}(L'); T; \{L'\} \rangle], \) because the free variables \(L\) and \(L'\) are different and there is no information in the built-in constraints that they are permutations of each other.

9. \([\langle \text{mset}(L); T; \{L\} \rangle] \approx^S [\langle \text{mset}(L'); \text{perm}(L, L'); \{L, L'\} \rangle].\)

10. \([\langle \text{mset}(L); \text{perm}(L, L'); \emptyset \rangle] \approx^S [\langle \text{mset}(L'); \text{perm}(L, L'); \emptyset \rangle] = [\langle \text{mset}(Y); \text{perm}(X, Y); \emptyset \rangle].\)

Since the information that \(L\) is a permutation of \(L'\) is explicit in both states, we do not need to share the variables in the states. We can even rename the local variables due to Lemma 2.1 about state equivalence (which we are allowed to use since we use equivalence classes of states).

In the next step, the equivalence relation over states with multi-sets can be defined as follows:

**Definition 4.1** (Multi-Set Equivalence \(\approx^S\)). The equivalence relation \(\approx^S\) is the smallest congruence relation with respect to \(V\) for all sets of variables \(V\) that satisfies the following condition:

\[
[\langle \text{mset}(L); \text{perm}(L, L') \wedge q(L'); \{L\} \rangle] \approx^S [\langle \text{mset}(L'); \text{perm}(L, L') \wedge q(L'); \{L, L'\} \rangle]
\]

for all built-in constraints \(q/1\).
The property defines that all states that contain a constraint $mset(L)$ are equivalent to a state where $L$ is replaced by $L'$ with the side-condition that $L'$ is a permutation of $L$. It also allows to make local variables from the original state global – under the side-condition that they are a permutation of $L$.

Furthermore, more restrictions on those local variables can be introduced by an arbitrary constraint $q$, as long as they are compatible with the overall state – i.e. they do not lead to a contradiction or in other words the built-in store $B$ is equivalent to $\exists L'. B \land \text{perm}(L, L') \land q(L)$. The restrictions can be made global in the equivalent states. Since global variables are free variables in the logical reading of a state, this allows to fixate a choice of the existential quantification of the local variable. It simply means that a general state with some variable $L$ is equivalent to all states that contain a (fixed) permutation of $L$, i.e. states where the choice of the actual permutation is fixed. This makes Definition 41 a scheme of equivalences introducing infinitely many tuples to the relation.

It can already be seen from the definition that $\approx^S$ is fully $\otimes_V$-compatible. The reason is the monotonicity of the built-in constraint store. Since $L'$ is local in the state on the left-hand side, it holds that $\langle mset(L); \top; \{L\} \rangle = \langle mset(L); \text{perm}(L, L') \land q(L'); \{L\} \rangle$. The minimal split of this state is the state $\langle mset(L); \top; \{L\} \rangle \otimes \langle \emptyset; \text{perm}(L, L') \land q(L'); \{L\} \rangle$.

It is clear that the state on the right hand side of the equation in Definition 41 can be split analogously and the components are equivalent with their corresponding counterparts.

The congruence relation property lifts the equality to all equivalent extensions. Since the merge operator is only defined over two states with different local variables, it excludes equivalences where there is a contradiction with respect to the local variable $L'$. In other words: We cannot add the local information $\text{perm}(L, L') \land q(L')$ to a state if this is a contradiction to the original built-in store, since $\otimes_V$ ensures that the local variables in both states are disjoint. Since $\exists L'. \text{perm}(L, L') \land q(L')$ for any strictly local variable $L'$, this information can be added.

We now demonstrate that Definition 41 meets the previously imposed requirements from Example 27 in the following Corollary. Furthermore, some more implications of this definition are exemplified and discussed.

**Corollary 5.** The desired properties of $\approx^S$ from Example 27 are satisfied by Definition 41.

**Proof.** We use Definition 41 to show the desired properties of $\approx^S$ from Example 27. The numbering refers to the numbering of that particular example. We use the following abbreviations to annotate the derivation steps:
(T): Transformation of the Constraint Store (Definition 7.2)
(S): Equality as Substitution (Definition 7.1)
(R): Variable Renaming (Lemma 2.1)
(M): Merge Operator (Definition 14)
(E): Multi-Set Equivalence (Definition 41)

1. \[[\langle mset([1,2]); \top; \emptyset\rangle]\]
   \[= [\langle mset([1,2]); L\equiv[1,2]; \emptyset\rangle]\]
   \[= [\langle mset(L); L\equiv[1,2]; \emptyset\rangle]\] (T)
   \[= [\langle mset(L); L\equiv[1,2] \land \text{perm}(L, L') \land L'\equiv[2,1]; \emptyset\rangle]\] (S)
   \[= [\langle mset(L); \text{perm}(L, L') \land L'\equiv[2,1]; \{L\}\rangle]\]
   \[\diamond_{\{1\}} [\langle \emptyset; L\equiv[1,2]; \{L\}\rangle]\] (M)
   \[\approxS [\langle mset(L'); \text{perm}(L, L') \land L'\equiv[2,1]; \{L, L'\}\rangle]\]
   \[\diamond_{\{1\}} [\langle \emptyset; L\equiv[1,2]; \{L\}\rangle]\] (E)
   \[= [\langle mset([2,1]); \top; \emptyset\rangle]\] (M/T/S)

   Note that in the first steps, the strictly local variables \(L\) and \(L'\) can be introduced by transforming the constraint store (c.f. Definition 7.2) because \(\exists L. L\equiv[1,2]\) is a tautology. In the second (T) step, the same is applied for the tautology \(\exists L, L'. L\equiv[1,2] \land \text{perm}(L, L') \land L'\equiv[2,1]\). Note that this is possible for any \(q/1\)-constraint with \(q(L')\) that does not contradict with \(\text{perm}(L, L')\). In the (E) step, the \(q/1\) constraint from Definition 41 is instantiated with \(CT \models \forall(q(L') \leftrightarrow L'\equiv[2,1])\).

2. \[[\langle mset([a,b]), mset([c,d]), \text{item}(e); \top; \emptyset\rangle]\]
   \[= [\langle mset([a,b]); \top; \emptyset\rangle] \diamond [\langle mset([c,d]); \top; \emptyset\rangle] \diamond [\langle \text{item}(e); \top; \emptyset\rangle]\]
   \[\approxS [\langle mset([b,a]); \top; \emptyset\rangle] \diamond [\langle mset([d,c]); \top; \emptyset\rangle] \diamond [\langle \text{item}(e); \top; \emptyset\rangle]\]
   \[= [\langle mset([b,a]), mset([d,c]), \text{item}(e); \top; \emptyset\rangle]\]

   Here we simply use the merge operator to split up the state into three parts. Since \(\approxS\) is a congruence relation, the operands of the merge operation can be treated individually. For the first two operands we apply \(\approxS\) analogously to number 1 of this example. The third operand is not touched. Reflexivity allows to use the congruence relation property on this operand. The resulting state is then merged again.

3. \([\langle \text{item}(a); \top; \emptyset\rangle]\) \(\approxS [\langle \text{item}(a); \top; \emptyset\rangle]\) follows directly from reflexivity.

4. \([\langle mset([1,2]); \top; \emptyset\rangle]\) \(\not\approxS [\langle mset([1]); \top; \emptyset\rangle]\), because:
   \[\langle mset([1,2]); \top; \emptyset\rangle\]
   \[\not= [\langle mset(L); L\equiv[1,2] \land \text{perm}(L, L') \land L'\equiv[1]; \emptyset\rangle]\]
   \[= \langle \bot \bot; \bot \rangle\].
Since the built-in constraint store of the state cannot be transformed to \( L \models [1, 2] \land \text{perm}(L, L') \land L' \models [1] \), because \( \text{perm}([1, 2], [1]) \) is a contradiction, Definition 41 cannot be applied.

5. \[
\begin{align*}
&[(\{a, b\}, \text{item}(c); T; \emptyset)] \\
&= [(\{a, b\}; T; \emptyset) \circ (\{item(c); T; \emptyset\]) \\
&\neq^S [(\{a, b\}; T; \emptyset) \circ (\{item(d); T; \emptyset\]) \\
&= [(\{a, b\}, \text{item}(d); T; \emptyset)].
\end{align*}
\]

This follows from the fact that
\[
[(\{item(c); T; \emptyset\]) \neq [(\{item(d); T; \emptyset\}].
\]

Thus, the congruence property cannot be applied.

6. \[
\begin{align*}
&[(\{a, b\}, \text{mset}([b, a]; T; \emptyset)] \\
&\neq^S [(\{a, b\}; T; \emptyset)]
\end{align*}
\]

Analogously to number 5, this fails because
\[
[(\{a, b\}; T; \emptyset)] \neq^S \sigma_\emptyset.
\]

7. \[
\begin{align*}
&[(\{L\}; T; \{L\})] \\
&= [(\{L\}; \text{perm}(L, L'); \{L\})] \quad \text{(T)} \\
&= [(\{L\}; \text{perm}(L, L'); \{L\}) \circ_{\{L\}} \sigma_\emptyset] \quad \text{(M)} \\
&\approx^S [(\{L'\}; \text{perm}(L, L'); \{L, L'\}) \circ_{\{L\}} \sigma_\emptyset] \quad \text{(E)} \\
&= [(\{L'\}; \text{perm}(L, L'); \{L\})] \quad \text{(M)}
\end{align*}
\]

Note that, again, we can transform the built-in constraint store according to Definition 7.2 in the first derivation step, because \( \exists L'. \text{perm}(L, L') \) is a tautology. In step (E), the constraint \( q/1 \) from Definition 41 is instantiated with \( CT \models \forall(q(L') \leftrightarrow T) \).

Furthermore, the following observations are implied:

\[
\begin{align*}
&[(\{L'\}; \text{perm}(L, L'); \{L\})] \\
&\neq^S [(\{L'\}; T; \{L\})] \quad \text{(and \neq)} \\
&= [(\{L'\}; T; \emptyset)] \\
&\neq^S [(\{L'\}; T; \{L'\})] \quad \text{(and \neq)}
\end{align*}
\]

This means that it is not possible to remove the side condition that the local variable \( L' \) is a permutation of the global variable \( L \). The reason for that is that \( L' \) is local, but not strictly local. Hence, it is not possible to transform the state according to Definition 7.2.

This inequality becomes obvious when we consider that the resulting state would be equivalent to \([\{\{L'\}; T; \emptyset\})\] according to Definition 7.3, i.e. the state where \( L \) is local. Hence, the connection to the original state \([\{\{L\}; T; \emptyset\})\] is lost.
8. \( \text{\texttt{\{mset}(L); \top; \{L\}\}} \not\approx^S \text{\texttt{\{mset}(L'); \top; \{L'\}\}} \)

This follows directly from the observations in number 7.

9. \( \text{\texttt{\{mset}(L); \top; \{L\}\}} \)
   \( = \text{\texttt{\{mset}(L); \text{\texttt{perm}}(L, L'); \{L\}\}} \)
   \( \approx^S \text{\texttt{\{mset}(L'); \text{\texttt{perm}}(L, L'); \{L, L'\}\}} \)

We again apply that \( \exists L'. \text{\texttt{perm}}(L, L') \) is a tautology. Furthermore, the constraint \( q/1 \) from Definition 41 is instantiated with \( CT \models \forall(q(L') \leftrightarrow \top) \).

10. \( \text{\texttt{\{mset}(L); \text{\texttt{perm}}(L, L'); \emptyset\}\}} \)
    \( = \text{\texttt{\{mset}(L); \text{\texttt{perm}}(L, L'); \{L\}\}} \circ_{\{L,L'\}} [\sigma_{\emptyset}] \quad (M) \)
    \( \approx^S \text{\texttt{\{mset}(L'); \text{\texttt{perm}}(L, L'); \{L, L'\}\}} \circ_{\{L,L'\}} [\sigma_{\emptyset}] \quad (E) \)
    \( = \text{\texttt{\{mset}(L'); \text{\texttt{perm}}(L, L'); \emptyset\}\}} \quad (M) \)
    \( = \text{\texttt{\{mset}(Y); \text{\texttt{perm}}(X, Y); \emptyset\}\}} \quad (R) \)

Note that due to the symmetric nature of \text{\texttt{perm}}/2, the two states \( \{\text{\texttt{mset}(L); \text{\texttt{perm}}(L, L'); \emptyset\}\} \) and \( \{\text{\texttt{mset}(L'); \text{\texttt{perm}}(L, L'); \emptyset\}\} \) are even state equivalent, i.e., we do not need the reasoning via \( \approx^S \) in this particular case: \( CT \models \forall(\text{\texttt{perm}}(X, Y) \leftrightarrow \text{\texttt{perm}}(Y, X)) \).

However, this is only possible for symmetric built-in constraints and cannot be generalized. Contrarily, the reasoning demonstrated above is possible for arbitrary built-in constraints. Note that merging the two states with a variable assignment, e.g. \( \sigma := \langle \emptyset; L \equiv [1, 2] \land L' \equiv [2, 1]; \emptyset \rangle \) would require to make the variables global in the state \( \sigma \) containing the contextual information (c.f. Example 9). To merge \( \sigma \) with one of the two states, the local variables have to be renamed to make them different from the global \( L, L' \). I.e., there is no connection between those variables, and in particular

\[
\text{\texttt{\{mset}(L); \text{\texttt{perm}}(L, L') \land L \equiv [1, 2] \land L' \equiv [2, 1]; \emptyset\}\}} \\
\neq \text{\texttt{\{mset}(L'); \text{\texttt{perm}}(L, L') \land L \equiv [1, 2] \land L' \equiv [2, 1]; \emptyset\}\}}.
\]

However, the two states are \( \approx^S \)-equivalent.

Note that the equivalence relation \( \approx^S \) is compatible \( \circ_Y \). The proof is provided for the generalization of \( \approx^S \) in Section 5.3.3.2.

In Corollary 5, it can be seen that the handling of local and global variables in Definition 41 is quite subtle. It is not obvious how the local variable in the \text{\texttt{mset}} constraint can be made global such that it matches the pattern of Definition 41. The central ideas to understand how the states can be transformed to establish the pattern are the following:

- Transformation of the constraint store (c.f. Definition 7.2) is used to introduce fresh strictly local variables that are bound to the
old variables or (ground) terms to hold the same information. It is always possible to introduce new strictly local variables with
side conditions on them because the strictly local variables are
existentially quantified in the logical reading of the state (c.f. Definition 7.2). Hence, as long as the side condition is satisfiable,
we can always add such built-ins to the built-in store.

• Equality by substitution (c.f. Definition 7.1) and partial substitu-
tion (c.f. Lemma 2.2) can be used to replace the old variables or
(ground) terms by fresh versions where needed.

• Variables are made global as in Example 9 to share contextual
information. This brings us closer to match the pattern of Defini-
tion 41. To maintain state equivalence between the two derivation
steps, \( \diamond V \) is used to remove the global variables from the state to
make them local in the result.

• Sometimes it is necessary to merge an expression with the empty
state \( \sigma_\emptyset \) or the corresponding neutral element of the merge
expression.

• The congruence relation property with respect to \( \diamond V \) is exploited
to replace the pattern of Definition 41 with its \( \approx^S \)-equivalent
counterpart. We are going to show that \( \approx^S \) is a congruence
relation with respect to merge in Lemma 15 in the following
section.

5.3.3.2 Generalization of the Equivalence Relation Scheme

In the previous section, the multi-set equivalence relation has been
constructed by example. Now, the main idea of defining an equiv-
alanee relation over replacements of user-defined constraints and
side-conditions formulated as built-in constraints is generalized to
arbitrary user-defined and built-in constraints. As the central result of
this section, we show that the defined class of equivalence relations is
\( \Sigma \)-compatible with respect to the merge operator.

On a more abstract level, equivalence relations such as \( \approx^S \) defined
in Definition 41, correspond to equivalences over abstract state
descriptions. This means that two subsets of the state space defined by
abstract state descriptions are defined as equivalent as demonstrated
in Example 20.

As we observed in Section 5.1, reasoning on the object language
level is equivalent to reasoning on the meta-level for CHR states and
hence we define the following class of equivalence relations on the
object level.

**Definition 42 (p-State Equivalence).** Let \( p \) be a built-in constraint. An
equivalence relation \( \approx \) is called a \( p \)-state equivalence relation, if it is the
5.3 Compatibility of Equivalence Relations

The smallest congruence relation with respect to $\approx_V$ for all sets of variables $V$ that has the following property:

$$[[c(\bar{x}); p(\bar{x}, \bar{y}) \land q(\bar{y}); \{\bar{x}\}]]$$

$$\approx [[c'(\bar{x}'); p(\bar{x}, \bar{y}) \land q(\bar{y}); \{\bar{x}, \bar{y}\}]],$$

for all built-in constraints $q$, where $c / m$, $c' / n (m \geq 0, n \geq 0)$ are user-defined constraint symbols, $\bar{x}, \bar{y}$ are sequences of variables and $\bar{x}'$ is a sequence of variables from $\bar{x}, \bar{y}$ with length $n$.

The idea of $p$-state equivalence is to consider states equivalent, if the values of the user-defined constraints satisfy some condition defined by the built-in constraint $p$. For example, the equivalence relation $\approx_S$ from Definition 41 is a $\text{perm}$-state equivalence relation.

**Corollary 6:** $\approx_S$ is a $\text{perm}$-state equivalence relation with user-defined constraint $\text{mset}$.

**Proof.** The definition of $\approx_S$ in Definition 41 matches Definition 42.

$p$-state equivalence relations are defined over equivalence classes of states, reflexivity of $\approx_S$ ensures that the definition extends traditional state equivalence, i.e. if $\sigma \equiv \sigma'$, then $[\sigma] \approx [\sigma']$.

Note that the condition in Definition 42 is an axiom scheme that can be instantiated for arbitrary constraints $c, c'$ and $p$. Hence, the definition allows to define multiple state pairs to be equivalent, for instance consider the smallest congruence relation with

$$[[c_1(\bar{x}); p_1(\bar{x}, \bar{y})\{\bar{x}\}]]$$

$$\approx [[c_1'(\bar{x}'); p_1(\bar{x}, \bar{y})\land q_1(\bar{y}); \{\bar{x}, \bar{y}\}]], \text{ and}$$

$$[[c_2(\bar{x}); p_2(\bar{x}, \bar{y})\{\bar{x}\}]]$$

$$\approx [[c_2'(\bar{x}'); p_2(\bar{x}, \bar{y})\land q_2(\bar{y}); \{\bar{x}, \bar{y}\}]]$$

for all built-in constraints $q_1$ and $q_2$ with corresponding arity. This relation is both a $p_1$-state equivalence relation for the pair of user-defined constraints $(c_1, c'_1)$ and a $p_2$-state equivalence relation for $(c_2, c'_2)$.

Furthermore, the arity of $p$ can be smaller than the sum of the lengths of $\bar{x}$ and $\bar{y}$. This can be achieved by introducing an artificial built-in constraint $p'/n + m$ with $CT \models p'(x_1, \ldots, x_n, y_1, \ldots, y_m) \leftrightarrow p(x'_1, \ldots, x'_n, y'_1, \ldots, y'_m)$, where $n' \leq n$ and $m' \leq m$ and $x'_1, \ldots, x'_n$ and $y'_1, \ldots, y'_m$ are arbitrary sub-sequences of $x_1, \ldots, x_n$ and $y_1, \ldots, y_m$, respectively.

We want to make our program analyses in the following chapters available to arbitrary $p$-state equivalence relations. Therefore, we want to show that those relations are not only congruence relations, but are even $\Sigma$-compatible with respect to $\approx_V$ for all $V$. The following lemma shows that $p$-state equivalence relations also have the $\Sigma_{\text{CHR}}$-split property and are $\Sigma_{\text{CHR}}$-split monotonic, i.e. they are fully $\approx_V$-compatible for all $V$. 

$p$-state equivalence relations are fully $\approx_V$-compatible.
Lemma 15 (Compatibility of p-State Equivalence). Let \( \approx \) be a p-state equivalence relation for some constraint \( c \) and built-in constraint \( p \). Then \( \approx \) is fully \( \odot_V \)-compatible.

Proof. The following properties have to be shown to prove that \( \approx \) is \( \Sigma \odot_V \)-compatible:

\[ \text{congruence relation } \approx \text{ has to be a congruence relation with respect to } \odot_V \text{ for all } V. \] This follows directly from Definition 42.

\[ \Sigma_{\text{CHR-split property}} \] Let \( [\sigma] \approx [\sigma'] \) and \( [\sigma] = [\sigma_1] \odot_V [\sigma_2] \) for an arbitrary \( V \). We want to construct states \( [\sigma'_1] \approx [\sigma_1] \) and \( [\sigma'_2] \approx [\sigma_2] \) such that \( [\sigma'] = [\sigma_1] \odot_V [\sigma_2] \). There are the following two cases:

1. \( [\sigma] = [\sigma'] \).
   
   It follows directly that every split of \( [\sigma] = [\sigma_1] \odot_V [\sigma_2] \) is also a possible split of \( [\sigma'] = [\sigma_1] \odot_V [\sigma_2] \) and hence the \( \Sigma_{\text{CHR-split}} \) property holds.

2. \( [\sigma] \neq [\sigma'] \).

   Since \( [\sigma] \neq [\sigma'] \), but \( [\sigma] \approx [\sigma'] \) and since \( \approx \) is the smallest \( \odot_V \)-congruence relation with the property of Definition 42, it follows that the following holds for \( [\sigma] \):

\[ [\sigma] = [(c(x); T; \{x\})] \odot_V [\delta] \]

for some \( V \) and \( \delta \).

W.l.o.g. we assume that \( [(c(x); p(x, y) \land q(y); \{x\})] \) is part of \( [\sigma_1] \), i.e. \( [\sigma_1] = [(c(x); p(x, y) \land q(y); \{x\})] \odot_{V_1} [\delta_1] \) for some \( V_1 \) and \( [\delta_1] \) (possibly different from above).

We choose \( [\sigma'_1] := [(c'(x'); p(x, y) \land q(y); \{x', y\})] \odot_{V_1} [\delta_1] \) for the sub-sequence \( x' \) of \( x, y \). It holds that \( [\sigma'_1] \approx [\sigma_1] \) by Definition 42.

If \( [\sigma_2] = [(c(x); p(x, y) \land q(y); \{x\})] \odot_{V_1} [\delta_2], \) the construction of \( [\sigma'_2] \) is analogous.

\[ \Sigma \text{-split monotonicity} \] Proof by contradiction. We assume that \( [\sigma] = [\sigma_1] \odot_V [\sigma_2] \) and \( [\sigma'] = [\sigma'_1] \odot_V [\sigma'_2], \) where \( [\sigma] \approx [\sigma'] \) and \( [\sigma_1] \approx [\sigma'_1] \), but \( [\sigma_2] \neq [\sigma'_2] \).

Then \( [\sigma'_2] \neq [\sigma_2]. \) Let \( [\sigma_2] = [(c(x); p(x, y) \land q(y); \{x\})] \) for some built-in constraint \( q/1. \) Since \( [\sigma_2] \neq [\sigma'_2], \) it holds that \( [\sigma'_2] \neq [(c'(x'); p(x, y) \land q(y); \{x', y\})] \) for a sub-sequence \( x' \) of \( x, y. \)

It follows that \( [\sigma] \neq [\sigma']. \)

\[ \Box \]
Hence, according to Lemma 8, such that it is a permutation of $L$. In theory, there are arbitrarily many such equivalent states with different concretions of $L$. For instance, the rule in Example 19 has the rule state $\sigma_r := \langle \text{mset}(L), \text{item}(A); \top; \{L, A\}\rangle$. The rule is applicable in the state

$$[\sigma] := \langle \text{mset}([1, 2]), \text{item}(3); \top; \emptyset\rangle.$$ 

Hence, according to Lemma 8, it holds that

$$[\sigma] = [\sigma_r] \diamond_{\{L, A\}} [\delta],$$

where $\delta := \langle \emptyset; L \vdash [1, 2] \land A \vdash 3; \{L, A\}\rangle$. It follows that

$$[\sigma_r] = \langle \text{mset}(L), \text{item}(A); \top; \{L, A\}\rangle = \langle \text{mset}(L), \text{item}(A); \text{perm}(L, L') \land q(L'); \{L, A\}\rangle$$

for some built-in constraints $q/1$.

From Definitions 41 and 42 it follows that

$$[\sigma] = [\sigma_r] \diamond_{\{L, A\}} [\delta]$$

where $\sigma'_r := \langle \text{mset}(L), \text{item}(A); \text{perm}(L, L') \land q(L'); \{L, L', A\}\rangle$. It can be seen that the split-property is satisfied.

Furthermore, this leads to some interesting equivalences to $[\sigma]$ that can all be split according to the above pattern. For instance, let $CT \models \forall (q(L') \leftrightarrow L' \equiv [2, 1])$. Then it follows that

$$[\sigma] \approx^S \langle \text{mset}(L'), \text{item}(A); L \equiv [1, 2] \land A \equiv 3$$

$$\land \text{perm}(L, L') \land L' \equiv [2, 1]; \emptyset\rangle$$

$$= \langle \text{mset}([2, 1]), \text{item}(3); \top; \emptyset\rangle.$$ 

In theory, there are arbitrarily many such equivalent states with different concretions of $L'$, as long as they are compatible with the fact that $L'$ is a permutation of $L$.

However, it follows that for $CT \models \forall (q(L') \leftrightarrow L' \equiv [1, 2, 3])$ the state $[\sigma] \not\approx^S \langle \text{mset}([1, 2, 3]), \text{item}(3); \top; \emptyset\rangle$.

For $CT \models \forall (q(L') \leftrightarrow \top)$ it follows that

$$[\sigma] \approx^S \langle \text{mset}(L'), \text{item}(A); \text{perm}(L, L') \land L \equiv [1, 2] \land A \equiv 3; \emptyset\rangle$$

$$= \langle \text{mset}(L'), \text{item}(3); \text{perm}([1, 2], L'); \emptyset\rangle,$$

where $L'$ is local. The logical reading of this state means that there exists a $L'$ such that it is a permutation of $[1, 2]$. 

**Example 28.** A typical application of $\diamond_{\nu}$-compatibility is to split $\approx$-equivalent states such that the split corresponds to splitting a state into a rule state and some contextual information as described in Lemma 8 and Example 11. For instance, the rule in Example 19 has the rule state $\sigma_r := \langle \text{mset}(L), \text{item}(A); \top; \{L, A\}\rangle$. The rule is applicable in the state

$$[\sigma] := \langle \text{mset}([1, 2]), \text{item}(3); \top; \emptyset\rangle.$$
The last observation in the example corresponds exactly with the observation that there are infinitely many states equivalent to \([\sigma']\) depending on \(q/1\). In our \(p\)-state equivalence relations we allow to make those existentially quantified local variables global with the side-condition that they satisfy some property \(p\) with respect to the (global) information available in the state. Since global variables are free variables in the logical reading, this introduces the ability to fixate a concrete witness of the existential quantified \(L'\) with property \(p\). The existential quantification then introduces infinitely many equivalent states with different concretions. In Section 6.4, we discuss how we still can exploit monotonicity in our confluence analysis due to the structure of \(p\)-state equivalence relations.

5.4 DISCUSSION

The results of this chapter suggest that due to the expressiveness of CHR and its close relation to (first-order) logic, a meta-language to describe sets of states is often unnecessary. In particular, equivalence relations between states can be defined formally sound by simply using (global) variables and (built-in) constraints on the object language level as it is already done for the plain state equivalence relation in Definition 7, for example. This is supported by the general definition of \(p\)-state equivalence that is entirely defined on the object language level.

However, the reasoning on the object level is only sound for purely logical built-in constraints. This means that built-in constraints like \texttt{var/1} that succeeds, if its argument is a variable, cannot be used. When such constraints are included, a meta-level analysis or methods from abstract interpretation may become helpful. This approach would then include a corresponding meta-semantics of CHR, as the current descriptions of operational semantics are not designed for such meta-analyses. Furthermore, they typically rely on purely logical built-in constraints or accept the consequence of breaking monotonicity by introducing non-logical constraints. This is e.g. the case for most implementations.

For groundness in particular, the method of case splitting as discussed in Section 5.1.3 is a promising proof technique. Our result in Lemma 11 allows to use this proof technique on both the meta-language and the object language level. We are therefore confident that it can also be applied when performing analysis on the meta-level.

Since, in general, user-defined equivalence relations on states violate monotonicity, the proposed subset of compatible equivalence relations is necessary for every program analysis that relies on this property. The restriction of program analysis to \(\Sigma\)-compatible equivalence relations (for some set of states \(\Sigma\)) is reasonable and includes meaningful
equivalence relations like the class of $p$-state equivalence that is even fully $\Diamond_V$-compatible, i.e. for $\Sigma = \Sigma_{\text{CHR}}$.

Note that for the confluence analysis in Chapter 6, the relations are not required to be fully compatible, but only with respect to the rule states and their minimal extensions. This makes the methods available for more exotic equivalence relations. In particular, the relaxation of $\Sigma$-compatibility allows to also consider equivalence relations in our analyses that do not behave as well as $p$-state equivalence relations. Although such equivalence relations by construction introduce some counter-intuitive properties as shown in Section 5.3.2, they might become useful in some practical applications.

Apart from the class of $p$-state equivalence relations, we are confident that there are other interesting examples of equivalence relations that are $\Diamond_V$: For instance equivalence relations that consider constraints with differing orders of arguments or renamed constraints equivalent. Another example could be equivalence relations that ignore certain arguments of a constraint.

5.5 RELATED WORK

The first work about reasoning about user-defined equivalence relations over states in CHR was the work on confluence modulo equivalence by Christiansen and Kirkeby [CK15]. This work has been extended and formalized more strictly in Christiansen and Kirkeby [CK15]. In contrast to most work in the literature, their formalization is based on their own operational semantics of CHR that is a mixture of analytical semantics like $\omega_{va}$ or $\omega_t$ and actual implementations. This allows them to reason about non-logical built-in constraints or constraints that they call incomplete like $\text{is}/2$ that may cause run-time exceptions for unbound variables. To reason about sets of such states, Christiansen and Kirkeby use a meta-language with an operational semantics over sets of states in their original, technical operational semantics.

We follow those ideas in a simplified form in Section 5.1, by introducing a similar form of meta-level propositions over sets of states (but related closely to the canonical operational semantics $\omega_v$ that is equivalent to $\omega_{va}$). We show that for purely logical built-in constraints, CHR allows to reason about such sets of states directly on the object language level. In contrast to the work of Christiansen and Kirkeby, this allows to reuse all standard analysis techniques from the literature.

We also show that the prominent example of the groundness invariant used in [CK17, pp. 3, 6] as a showcase of where object level analysis methods from the literature fail, are still possible to analyze with the invariant-based confluence criterion of Raiser described Theorem 7. One motivation for introducing meta-level reasoning of Christiansen
and Kirkeby is their claim that “the logical subsumption principle [. . .]
leads to infinitely many proof cases for even simple invariants such as
groundedness; our meta-language approach handles such examples
in a more satisfactory way” [CK17, p. 3]. Thereby, they refer to the
work of [DSS07] that also serves as a foundation for the improved
methods of Raiser we use in this thesis [Rai10] (c.f. Section 3.3.7).
However, as we show in Corollary 4, at least the improved methods
of Raiser are applicable to the groundedness invariant. The claim that
ground is a non-logical built-in constraint [CK17] can be mitigated by
introducing a monotonic version of it as shown in Definition 34. With
this version of the ground constraint, the groundedness invariant can
be realized and invariant-based analysis techniques can be applied.
Lemma 12 shows, that the groundedness invariant always has a finite set
of minimal extensions. Hence, the invariant-based confluence criterion
is decidable.

Note that these considerations make it possible to use the ground-
ness invariant in our own confluence modulo equivalence criterion in
Chapter 6.

Christiansen and Kirkeby also formalize the proof method of case
splitting in their meta-language semantics that is based on their own
operational semantics described above [CK17, pp. 26–27]. Our work
uses the well-established standard semantics \( \omega_e \) that is equivalent
to the very abstract semantics \( \omega_{va} \). In addition to our work, case
splitting has been used informally before in many classical examples
for \( \omega_{va} \) [Frü09, pp. 180, 184, 196]. In contrast to [CK17, Frü09],
we have defined the formal foundations for this approach by applying the
well-known invariant-based confluence property for the groundedness
invariant in Corollary 4 and showing the correspondence of the meta-
language level and the object language level in Lemmas 10 and 11.
We demonstrate how case splitting can be applied to complete such
proofs in the ground range-restricted fragment of CHR.

Nevertheless, for non-logical or incomplete built-in constraints or
for more complex invariants like in Example 16 that lead to an infinite
set of minimal elements, the meta-level reasoning of Christiansen and
Kirkeby may be of use. An interesting question is how the meta-level
approach of Christiansen and Kirkeby and our object level approach
can be combined.

Aside from the difference that our work concentrates on the object
language level, whereas the work of Christiansen and Kirkeby is set up
in a meta-level framework, we also want to discuss the delimitations
of our work concerning the reasoning about user-defined equivalences.
The equivalence relations in [CK17] follow a pattern closely related to
Example 20. This class of equivalence relations is similar to our defi-
nition of \( p \)-state equivalence as defined in Section 5.3.3. In contrast to the
work in [CK17], we consider the properties of this class of equivalence
relations and generalize them for arbitrary user-defined equivalence
5.5 Related Work

relations (namely the properties required for $\diamondsuit \gamma$-compatibility). In [CK17], those properties are not required formally in the proof methods. The reason for this is that the proof methods in [CK17] do not describe how the so-called abstract joinability diagrams can be proven, since the sets of states they represent may be arbitrarily complex. However, in their examples, all equivalence relations have the properties for what we call $\diamondsuit \gamma$-compatibility and hence our findings are applied implicitly [CK17, pp. 28, 31].

Therefore, the most significant differences of our work compared to [CK17] are the following:

- [CK17] gives a new, more abstract formalization of CHR (with user-defined equivalence relations) that tries to generalize the problem for a more technical operational semantics that allows for non-logical and incomplete built-in constraints, whereas our work embeds the reasoning in the existing semantics and analysis frameworks.

- [CK17] concentrates on building those new reasoning techniques, whereas our work concentrates on exploring the space of user-defined equivalence relations and the implications of certain properties. Thereby, the aim is to find a manageable subset of equivalence relations.

Apart from work about user-defined equivalence relations in CHR, the foundations of this research area go back to the domain of term rewriting systems – most prominently [Hue80]. In contrast to this general examination of the field, our work concentrates on applying the findings to CHR and finding a manageable subset of equivalence relations that allows to use the important monotonicity property of CHR.
While confluence is a desirable property, in practical applications it is often too strict. For instance, it requires even states that can never be reached in a practical context to satisfy the confluence property. Therefore, invariant-based confluence [DSS07; DSS06; Rai10] has been established (c.f. Section 3.3.7.2).

Another method of making the confluence property available for more practical programs is to define an equivalence relation on states. A program is confluent modulo a (user-defined) equivalence relation if all states in the same equivalence class lead to final states of the same equivalence class [CK15; CK17]. In many programs, some states can be considered as equivalent with respect to a user-defined equivalence relation, although their actual representation in the program differs. For example, if sets of numbers are represented as lists, all states with permutations of the same list represent the same set and it might be reasonable to consider them equivalent. Hereby, confluence modulo equivalence can be used to show that for the same start state a program yields the same set as a result, although the actual representation as a list might differ.

**Example 29 (Multi-Set Items (cont.) [CK17]).** Consider the program from Example 19 that collects items represented in individual item/1 constraints to a multi-set. The program only consists of the following rule:

\[ mset(L), item(A) \Leftrightarrow mset([A|L]). \]

For the initial constraint store

\[ \sigma := \langle item(a), item(b), mset([]); T; \emptyset \rangle \]

the following derivation can be applied:

\[ \sigma \mapsto_e (item(a), mset([b]); T; \emptyset) \]
\[ \mapsto_e (mset([a,b]); T; \emptyset). \]

However, \( \omega_e \) also allows for the following derivation:

\[ \sigma \mapsto_e (item(b), mset([a]); T; \emptyset) \]
\[ \mapsto_e (mset([b,a]); T; \emptyset). \]

Hence, the program is not confluent (c.f. Definition 2). Naturally, the two final states should be considered as semantically equivalent, since the order of the list does not play a role for the multi-set it represents.
There is a trade-off between the applicability in practical contexts and the simplicity of proving a confluence property: There is a decidable, sufficient and necessary criterion for strict confluence of terminating CHR programs [Frü09]. When adding invariants, decidability of the criterion is lost depending on the invariant. For confluence modulo equivalence, the proofs even become harder as all states in the same equivalence class have to be considered.

Since confluence modulo equivalence and invariants are orthogonal concepts that can be combined in practice, this chapter introduces the concept of invariant-based confluence modulo equivalence. In our running example, we can see that the two concepts complement each other meaningfully.

Example 30 (Multi-Set Items (cont.)). Consider again the program from Example 19. For the initial constraint store

\[ \sigma := \langle \text{item}(x), \text{mset}([a]), \text{mset}([b]); \top; \emptyset \rangle \]

the following derivation can be applied:

\[ \sigma \rightarrow_e \langle \text{mset}([a, x]), \text{mset}([b]); \top; \emptyset \rangle. \]

However, the following derivation is possible as well:

\[ \sigma \rightarrow_e \langle \text{mset}([a]), \text{mset}([b, x]); \top; \emptyset \rangle. \]

The example reveals the following insight: If two mset/1 constraints are present, it is not clear to which of the two options the item is added. Hence, the program is not confluent (c.f. Definition 2).

Let \( I_S \) be the invariant that only one mset/1 constraint is allowed in a state. \( I_S \) actually is an invariant in the example program from Example 19, since the only rule only replaces one mset/1 constraint with a new one. With this invariant, the above problem can be fixed. Together with the argumentation in Example 29, it is clear that both invariants and user-defined equivalence relations should be considered already for this simple example.

In this chapter, a sufficient and necessary criterion for invariant-based confluence modulo equivalence is presented. The approach is based on our considerations on general program analysis with user-defined equivalence relations in Chapter 5. Hence, the subset of \( \diamondsuit \gamma \)-compatible equivalence relations is used to construct the criterion based on the canonical (invariant-based) confluence criterion for CHR (c.f. Theorems 5 and 7).

First, confluence modulo equivalence is defined and the most important results for abstract rewriting systems are reproduced in Section 6.1. The definition is then generalized accordingly to integrate invariants in Section 6.2. Section 6.3 develops a general criterion for invariant-based confluence modulo equivalence. In Section 6.4, a simplified criterion for \( p \)-state equivalence relations is presented. For invariants that are maintained by the equivalence relation, the criterion can be simplified further as presented in Section 6.5. Sections 6.6 and 6.7 discuss the results and related work, respectively.
6.1 Confluence modulo equivalence

In this section, the notion of confluence modulo equivalence is defined for general state transition systems. Furthermore, some important results that are the foundation to our work on confluence modulo equivalence for CHR are repeated.

Informally, confluence modulo equivalence means that all possible computations in a transition system starting in equivalent states finally lead to equivalent states again. We start by lifting the definition of joinability and confluence to (c.f. Definition 2) to user-defined equivalence relations.

**Definition 43 (Joinability Modulo Equivalence).** In a state transition system $(\Sigma, \rightarrow)$ two states $\sigma, \sigma' \in \Sigma$ are joinable modulo an equivalence relation $\approx$ if and only if $\exists \tau, \tau' \in \Sigma$ such that $\sigma \rightarrow^* \tau \land \sigma' \rightarrow^* \tau' \land \tau \approx \tau'$. We then write $\sigma \downarrow^\approx \sigma'$.

Note that if $\approx$ is the identity equivalence relation $=\equiv$, joinability modulo $\equiv$ coincides with joinability according to Definition 2. Hence, we write $\sigma \downarrow \sigma'$ and say that $\sigma$ and $\sigma'$ are joinable.

**Definition 44 (Confluence Modulo Equivalence [Hue80, p. 802]).** A state transition system $(\Sigma, \rightarrow)$ is confluent modulo an equivalence relation $\approx$, if and only if for all $\sigma_1, \sigma'_1, \sigma_2, \sigma'_2$:

$$(\sigma_1 \approx \sigma'_1) \land (\sigma_1 \rightarrow^* \sigma_2) \land (\sigma'_1 \rightarrow^* \sigma'_2) \rightarrow (\sigma'_1 \downarrow^\approx \sigma'_2).$$

The definitions are illustrated in Fig. 6.1. The dashed lines denote existentially quantified transitions that stem from joinability (c.f. Definition 43). This means that it suffices to find one transition (with arbitrary many steps – including zero) that joins the two states. If $\approx$ is the state equivalence relation $\equiv$, confluence modulo $\equiv$ coincides with basic confluence for CHR. This is addressed in detail in Section 6.6.3.

![Figure 6.1: Confluence diagram for confluence modulo a user-defined equivalence relation $\approx$ [GF88b, p. 118].](image)

For terminating transition systems, it suffices to show local confluence, as we will see in the following definition and theorem. Thereby, the definition of local confluence is generalized for user-defined equivalence relations.

**Definition 45 (Local Confluence [Hue80]).** A state transition system $(\Sigma, \rightarrow)$ has the $\alpha$- and $\beta$-property with respect to an equivalence relation $\approx$ of and only if it satisfies the $\alpha$- and $\beta$-conditions, respectively:
A state transition system is locally confluent modulo an equivalence relation \( \approx \) if and only if it has the \( \alpha \) and the \( \beta \)-property.

The \( \alpha \)-property corresponds to classical local confluence (c.f. Definition 3), whereas the \( \beta \)-property basically defines a sense of weak monotonicity modulo an equivalence relation where one rule application is not required to correspond to exactly one other rule application, but joinability modulo equivalence suffices. It guarantees that if a rule is applicable in a state \( \sigma \) leading to a result state \( \tau \), then any equivalent state \( \sigma' \) is joinable modulo \( \approx \) with the result \( \tau \). This means that there are rules that can be applied to the result \( \tau \) and the equivalent state \( \sigma' \) such that the resulting states are equivalent again. The two properties are illustrated in Fig. 6.2. The dashed arrows denote existentially quantified transitions that stem from joinability (c.f. Definition 43).

Theorem 9 (Theorem of Huet [Hue80]). Let \((\Sigma, \rightarrow)\) be a terminating transition system. For any equivalence \(\approx\), \((\Sigma, \rightarrow)\) is confluent modulo \(\approx\) if and only if \((\Sigma, \rightarrow)\) is locally confluent modulo \(\approx\).

The Theorem of Huet generalizes Newman’s Lemma (Lemma 1) to confluence modulo equivalence. Note that the \( \alpha \)-property of local confluence modulo equivalence coincides with local confluence (when state equivalence or identity is used as equivalence relation). The \( \beta \)-property is an additional requirement such that local confluence and confluence coincide for terminating transition systems when arbitrary equivalence relations are considered.

6.2 Extension of Confluence Modulo Equivalence by Invariants

In this section, the definition of confluence modulo equivalence is combined with invariant-based confluence. First of all, the notion of invariant-based confluence modulo equivalence is defined.
**Definition 46** (I-Confluence Modulo ≈). A state transition system is confluent modulo an equivalence relation ≈ with respect to an invariant I if and only if

\[ \forall \sigma_1, \sigma_2, \sigma_1', \sigma_2'. \ I(\sigma_1) \land I(\sigma_1') \land \sigma_1 \approx \sigma_1' \land \sigma_1 \rightarrow^* \sigma_2 \land \sigma_1' \rightarrow^* \sigma_2' \rightarrow \exists \sigma_3, \sigma_3'. \ \sigma_2 \rightarrow^* \sigma_3 \land \sigma_3' \approx \sigma_3'. \]

We also say that the state transition system is I-confluent modulo ≈.

This definition naturally combines the definitions for invariant-based confluence as in Definition 23 and confluence modulo equivalence as in Definition 44.

### 6.3 Decision Criterion

In this section, the sufficient and necessary criterion for invariant-based confluence modulo equivalence of CHR programs is developed. First, the joinability corollary from [Rai10, p. 85] that uses monotonicity to reason from joinability of rule states about joinability of any larger state is generalized for user-defined equivalence relations. Then a criterion for both the α-property and the β-property is given. Both criteria are summarized to the central theorem of this chapter: A sufficient and necessary criterion for invariant-based confluence modulo equivalence of CHR programs (c.f. Theorem 10).

#### 6.3.1 Joinability Modulo Equivalence

For this purpose, an important technical lemma on joinability modulo equivalence is presented. The section continues with a test for the α and the β-property. Based on those two tests, the sufficient and necessary criterion for invariant-based confluence modulo equivalence is described.

The following lemma is an important generalization of the classical joinability corollary from [Rai10, p. 85] that forms the foundation of invariant-based confluence analysis (c.f. Theorem 7). The corollary from [Rai10, p. 85] is a direct consequence of monotonicity. The idea was that if two states are joinable, they are still joinable if they are extended by identical states, i.e. if \([\sigma] \downarrow \downarrow [\sigma']\) then \([\sigma_1] \oo V [\sigma'] \downarrow \downarrow [\sigma_2] \oo V [\sigma']\) for all \(\sigma'\) and \(V\).

This approach can be generalized to the context of confluence modulo equivalence. Monotonicity can be exploited such that the state extensions are not required to be syntactically identical, but equivalent for some user-defined compatible equivalence relation.

**Lemma 16** (Joinability Modulo Equivalence). Let \(\approx\) be a congruence relation with respect to \(\oo V\) for all \(V\) and \([\sigma_1], [\sigma_2], [\sigma_1'], [\sigma_2']\) be CHR states with \([\sigma_1'] \approx [\sigma_2']\). If \([\sigma_1] \downarrow^\approx [\sigma_2]\) then \(([\sigma_1] \oo V [\sigma_1']) \downarrow^\approx ([\sigma_2] \oo V [\sigma_2'])\) for all \(V\).
Proof. Let \([σ₁],[σ₂],[σ₁′],[σ₂′]\) be CHR states with \([σ₁′] \equiv [σ₂′]\) and \([σ₁] \Downarrow [σ₂]\). Hence, there are CHR states \([τ],[τ′]\) with \([τ] \equiv [τ′]\) and \([σ₁] \Rightarrow [τ]\) and \([σ₂] \Rightarrow [τ′]\). Due to monotonicity (c.f. Lemma 7), we have that \((σ₁ \circ_ρ σ₂′) \Rightarrow (τ \circ_ρ σ₁′)\) and \((σ₂ \circ_ρ σ₁′) \Rightarrow (τ′ \circ_ρ σ₂′)\). Since \([σ₁′] \equiv [σ₂′]\), \([τ] \equiv [τ′]\) and \([σ₁] \equiv [σ₂]\) is a congruence relation with respect to \(\circ_ρ\), we have that \((τ \circ_ρ [σ₁′]) \equiv (τ′ \circ_ρ [σ₂′])\).

This lemma is central to all program analysis results of this work.

6.3.2 Criterion for the α-Property

In this section, we provide a test for the α-property in the context of an invariant. The basic idea is that we gather all overlap states and extend them with a minimal extension such that the invariant does hold. For all those minimal extensions of all overlap states we have to show joinability modulo equivalence. Formally, this leads to the following lemma.

**Lemma 17 (α-Property Test).** Let \(P\) be a CHR program, \(I\) an invariant, \(\equiv\) a congruence relation and let \(M^I([σ])\) be well-defined for all overlaps σ of rules in \(P\), then: \(P\) has the α-property with respect to \(I\) and \(\equiv\) if and only if for all overlaps σ with critical pairs \((σ₁,σ₂)\) and all \([σₘ]\) \(\in M^I([σ])\) holds \((σ₁ \circ_ρ σ₂) \Downarrow [σₘ]\).

**Proof.** In the first steps, the α-property test coincides with the invariant-based confluence test (c.f. Theorem 7) first presented for CHR in [Raiho, p. 86]. However, the proof has to be adapted in the last step as joinability now allows states to join modulo an equivalence relation.

\[ ⇒ \]

This follows directly from Definition 45 and Lemma 16.

\[ ⇐ \]

Let \([σ],[σ₁]\) and \([σ₂]\) be CHR states where \(I([σ])\) holds and \([σ] \Rightarrow_{r₁} [σ₁]\) for some rule \(r₁\) and \([σ] \Rightarrow_{r₂} [σ₂]\) for some rule \(r₂\). By Definition 17, there exists an overlap state \(σ₀ = (σ;.;V)\) of rule \(r₁\) and \(r₂\) where \(V\) contains all variables from \(r₁\) and \(r₂\) such that for some \(δ := ([G;B;V'])\) it holds that \([σ] = [σ₀] \circ_ρ [δ]\). The variables \(V\) from the rules \(r₁\) and \(r₂\) are not part of \([σ]\) and are therefore removed by the merging \(\circ_ρ\). Due to monotonicity (c.f. Lemma 7), we have that

- \([σ₀] \Rightarrow_{r₁} [σ₁′]\) with \([σ₁] = [σ₁′] \circ_ρ [δ]\), and
- \([σ₀] \Rightarrow_{r₂} [σ₂′]\) with \([σ₂] = [σ₂′] \circ_ρ [δ]\).

If no such overlap exists, the two rule applications are independent and therefore trivially joinable.

As \(I([σ])\) holds, we have that \([δ] \in Σ^I([σ₀])\). Therefore there is a element in the set of minimal extensions that is less than or equal to \([δ]\), i.e. \(∃[σₘ] \in M^I([σ₀]).[σₘ] \preceq [δ]\). This means that there is a
minimal element $[\sigma_m]$ in the set of extensions of the overlap state $[\nu_0]$ that extend $[\nu_o]$ such that the invariant holds.

It follows by definition of $\prec$ that $\exists [\delta'], [\delta] = [\sigma_m] \diamond [\delta']$ and hence $[\sigma] = [\sigma_o] \circ \nu ([\sigma_m] \diamond [\delta'])$. By Lemma 4, we get $[\sigma] = ([\sigma_o] \circ [\sigma_m]) \circ \nu [\delta']$. Analogously, since $\mathcal{I}([\sigma_i]), i = 1, 2$ holds due to the definition of an invariant, we find that $[\sigma_i] = ([\sigma_i'] \circ [\sigma_m]) \circ \nu [\delta']$ for $i = 1, 2$.

At this point the proof differs from confluence without an equivalence relation. By the precondition we now only have that $( [\sigma_i'] \circ [\sigma_m] ) \downarrow \approx ( [\sigma_2] \circ [\sigma_m] )$, i.e. modulo an equivalence relation. Since $[\sigma_1] = ([\sigma_1'] \circ [\sigma_m]) \circ \nu [\delta']$ and $[\sigma_2] = ([\sigma_2'] \circ [\sigma_m]) \circ \nu [\delta']$, we can apply Lemma 16 due to reflexivity of $\approx$ (i.e. $[\delta'] \approx [\delta']$) and get $( [\sigma_1] \downarrow \approx [\sigma_2] )$. $\Box$

6.3.3 Criterion for the $\beta$-Property

To prove local confluence modulo equivalence, we also have to prove the $\beta$-property, i.e. we have to consider that if in a state a CHR transition is possible and the state is equivalent to another state, then the successor state and the equivalent state have to be joinable modulo equivalence. In the following lemma, we adapt the test for the $\alpha$-property to cover the $\beta$-property.

The main idea is to reason from rule states, i.e. the head and guard constraints of rules, over all states. Then, all states equivalent to the rule state have to be considered. Every pair of those equivalent states has to be extended by the minimal extension such that the invariant holds in both states, since otherwise the generalization to larger states fails analogously to invariant-based confluence (c.f. Section 3.3.7.2). If all the extended pairs are joinable modulo the equivalence relation, the $\beta$-property holds.

First of all, the definition of minimal extensions is generalized to invariants and equivalence relations.

**Definition 47** ($\mathcal{I}, \approx$-Extensions, Minimal Extensions). For an invariant $\mathcal{I}$ and an equivalence relation $\approx$, let

$$\Sigma^{\mathcal{I}, \approx}([\nu_1], [\nu_2]) := \{ ([\sigma_1'], [\sigma_2']) | \mathcal{I}([\sigma_1 \circ \sigma_1']) \wedge \mathcal{I}([\sigma_2 \circ \sigma_2']) \\
\wedge [\sigma_1 \circ \sigma_1'] \approx [\sigma_2 \circ \sigma_2'] \\
\wedge \sigma_1' \text{ and } \sigma_2' \text{ have no local variables} \}.$$

The set $\mathcal{M}^{\mathcal{I}, \approx}([\nu_1], [\nu_2])$ is defined as the set of $\prec$-minimal elements of $\Sigma^{\mathcal{I}, \approx}([\nu_1], [\nu_2])$ such that

$$\forall ([\sigma_1'], [\sigma_2']) \in \Sigma^{\mathcal{I}, \approx}([\nu_1], [\nu_2]). \exists ([\sigma_1^1], [\sigma_2^1]) \in \mathcal{M}^{\mathcal{I}, \approx}([\nu_1], [\nu_2]). [\sigma_m^1] \prec [\sigma_1'] \wedge [\sigma_m^2] \prec [\sigma_2'].$$

The set $\mathcal{M}^{\mathcal{I}, \approx}([\nu_1], [\nu_2])$ consists of the minimal elements of $\Sigma^1([\nu_1])$ and $\Sigma^2([\nu_2])$ that extend $[\nu_1]$ or $[\nu_2]$, respectively, such that the resulting states are $\approx$-equivalent and the invariant $\mathcal{I}$ holds. Hence, given
two (equivalent) small states, the set produces the minimal extensions of the two states, such that the resulting states are (still) equivalent and the invariant holds.

In fact, for every element \((\sigma_1', \sigma_2') \in \Sigma^{\approx,}\) it holds that \(\sigma_1' \in \Sigma^\approx(\sigma_1, \sigma_2)\) and \(\sigma_2' \in \Sigma^\approx(\sigma_2, \sigma_2)\).

Hence, for the minimal elements \((\sigma_1', \sigma_2') \in \mathcal{M}^{\approx,}(\sigma_1, \sigma_2)\) it holds that there are \(\sigma_1^1_m \in \mathcal{M}^\approx(\sigma_1)\) and \(\sigma_2^2_m \in \mathcal{M}^\approx(\sigma_2)\) such that \(\sigma_1^1_m \diamond \delta_1 = \sigma_1'\) and \(\sigma_2^2_m \diamond \delta_2 = \sigma_2'\) for some \(\delta_1, \delta_2\).

The situation is depicted in Fig. 6.3 that contains two (incomplete) Hasse diagrams for the states \(\sigma_1\) and \(\sigma_2\). For the sake of simplicity, we only consider the extensions of the states that are of interest for the definition, the others are omitted in the diagram. In the diagram, the invariant is not satisfied in red states, whereas it is satisfied in green states. Hence, \(I(\sigma_1)\) and \(I(\sigma_2)\) do not hold for the two equivalent states \(\sigma_1\) and \(\sigma_2\). The states \(\sigma_1^1_m\) and \(\sigma_2^2_m\) are the minimal extensions of \(\sigma_1\) and \(\sigma_2\), respectively, such that the invariant holds, i.e. \(\sigma_1^1_m \in \mathcal{M}^\approx(\sigma_1)\) and \(\sigma_2^2_m \in \mathcal{M}^\approx(\sigma_2)\). However, the two states \(\sigma_1 \diamond \sigma_1^1_m\) and \(\sigma_2 \diamond \sigma_2^2_m\) are not equivalent. The extensions \(\sigma_1 \diamond \sigma_1^1_m\) and \(\sigma_2 \diamond \sigma_2^2_m\) are the minimal extensions where the invariant is satisfied in both states and the extended states are equivalent, i.e. \((\sigma_1 \diamond \sigma_1^1_m, \sigma_1 \diamond \sigma_2^2_m) \in \mathcal{M}^\approx(\sigma_1, \sigma_2)\). Note that such complex situations as depicted in Fig. 6.3 only occur in the most general case. For equivalence relations that maintain the invariant, the situation becomes much more straightforward, as described in Section 6.5.

We now continue with the general criterion for the \(\beta\)-property.

**Lemma 18 (\(\beta\)-Property Test).** Let \(P\) be a CHR program, \(I\) an invariant, \(\approx\) a \(\Sigma\)-compatible equivalence relation and let \(\mathcal{M}^\approx(\sigma_1, \sigma_2)\) be well-defined for all rule states \(\sigma\) in \(P\) and their \(\approx\)-equivalent states \(\sigma_2\). Let for all rule states \(\sigma\) hold that \(\sigma\) in \(\Sigma\) and let for all rule states \(\sigma\) and all \(\sigma_2 \approx \sigma\) hold that if \((\sigma^1_m, \sigma_2) \in \mathcal{M}^\approx(\sigma, \sigma_2)\), then \(\sigma^1_m \in \Sigma\).

Then: \(P\) has the \(\beta\)-property with respect to \(I\) and \(\approx\) if and only if for all rule states \(\sigma\) with successor state \(\sigma_1\), all \(\sigma_2\) with \(\sigma \approx \sigma_2\) and all \((\sigma^1_m, \sigma_2^2_m) \in \mathcal{M}^\approx(\sigma, \sigma_2)\), it holds that \((\sigma^1_1 \diamond \sigma^1_m) \downarrow \approx (\sigma_2 \diamond \sigma^2_m)\).

**Proof.** “\(\Rightarrow\):”

This follows from Definition 45 and Lemma 16.

“\(\Leftarrow\):”


Let $[\sigma], [\sigma_1]$ and $[\sigma_2]$ be CHR states where $\mathcal{I}([\sigma])$ and $\mathcal{I}([\sigma_2])$ hold and $[\sigma] \mapsto_r [\sigma_1]$ for some rule $r$ and $[\sigma] \approx [\sigma_2]$. Since a rule is applicable in $[\sigma]$, there is a rule state $\sigma_r = \langle \_; \_; \mathbb{V} \rangle$ of rule $r$ such that for some $[\delta_1] := ([G_r; B_r; V_r])$ it holds that $[\sigma] = [\sigma_r] \circ \mathbb{V} [\delta_1]$ according to Lemma 8. The variables $\mathbb{V}$ from the rule $r$ are not part of $[\sigma]$ and are therefore removed by the merging $\circ \mathbb{V}$.

By state splitting (c.f. Lemma 8), there is a state $[\sigma'_1]$ such that $[\sigma_r] \mapsto [\sigma'_1]$. Due to monotonicity (c.f. Lemma 7) it holds that $[\sigma_r] = [\sigma'_1] \circ \mathbb{V} [\delta_1]$. Let $[\sigma_2] = [\sigma'_2] \circ \mathbb{V} [\delta_2]$ be a partition of $[\sigma'_2]$ such that $[\sigma'_2] \approx [\sigma_r]$ and $[\delta_2] \approx [\delta_1]$. Such a partition exists since $\approx$ is $\Sigma$-compatible, $[\sigma] \approx [\sigma_2]$ and $[\sigma] \in \Sigma$ by preconditions.

As both $\mathcal{I}([\sigma])$ and $\mathcal{I}([\sigma_2])$ hold, $[\sigma] \approx [\sigma_2]$ and w.l.o.g. $[\sigma]$ and $[\sigma_2]$ have no local variables (since we can always remove global variables using the merge operator as described in Section 3.3.1): $([\delta_1], [\delta_2]) \in \Sigma^2 \approx ([\sigma_r], [\sigma'_2])$ and therefore $\exists ([\sigma^m_1], [\sigma^m_2]) \in \mathcal{M}^2 \approx ([\sigma_r], [\sigma'_2])$ such that $[\delta_1] \wedge [\sigma^m_1] < [\delta_2]$. This means that there are minimal extensions $[\sigma^m_1], [\sigma^m_2]$ that extend $[\sigma_r]$ and $[\sigma'_2]$ such that the invariant holds in the resulting extended states $[\sigma_r] \circ [\sigma^m_1]$ and $[\sigma'_2] \circ [\sigma^m_2]$, where $[\sigma_r] \circ [\sigma^m_1] \approx [\sigma'_2] \circ [\sigma^m_2]$. Since $\approx$ is split monotonous and $[\sigma_r] \approx [\sigma'_1]$ it follows that $[\sigma^m_1] \approx [\sigma^m_2]$.

It follows by definition of $\circ$ that $\exists [\delta'_1], [\delta_1] = [\sigma^m_1] \circ [\delta'_1]$ and hence $[\sigma] = [\sigma_r] \circ \mathbb{V} ([\sigma^m_1] \circ [\delta'_1])$. By Lemma 4, we get

$$[\sigma] = ([\sigma_r] \circ [\sigma^m_1]) \circ \mathbb{V} [\delta'_1].$$

Analogously, $\exists [\delta'_2], [\delta_2] = [\sigma^m_2] \circ [\delta'_2]$ and $[\sigma_2] = [\sigma'_2] \circ \mathbb{V} ([\sigma^m_2] \circ [\delta'_2])$ and hence $[\delta_2] = [\sigma^m_2] \circ [\delta'_2]$.

Since $[\delta_1] \approx [\delta_2]$ and $[\sigma^m_1], [\sigma^m_2] \in \Sigma$, the $\Sigma$-split property ensures that there exist $[\delta^m_1], [\delta^m_2]$ such that $[\delta_2] = [\delta^m_2] \circ [\delta_1], [\delta^m_1] \approx [\sigma^m_1]$ and $[\delta_1] \approx [\delta^m_2]$. Hence, it follows that there are the following two partitions of $[\delta_2]$:

$$[\delta_2] = [\sigma^m_1] \circ [\delta'_2] = [\sigma^m_2] \circ [\delta_1],$$

where $[\sigma^m_2]$ stems from the minimal extensions of the rule state $[\sigma_r]$ and its equivalent state $[\sigma'_2]$ with the corresponding rest $[\delta'_2]$, whereas $[\sigma^m_1]$ and $[\delta_1]$ are just the corresponding equivalent parts of the partition of $[\sigma]$ according to the split property of $\approx$.

Since $[\sigma^m_1] \approx [\sigma^m_2] \approx [\sigma^m_1]$, it follows that $[\sigma^m_2] \approx [\delta^m_1]$. Due to split monotonicity, we receive $[\delta^m_1] \approx [\delta'_1]$. Analogously to above, we get

$$[\sigma_2] = ([\sigma'_2] \circ [\sigma^m_2]) \circ \mathbb{V} [\delta'_2].$$

By substitution of $[\delta_1]$ in $[\sigma_1]$ by $[\sigma^m_1] \circ [\delta'_1]$, we find that $[\sigma_1] = ([\sigma'_2] \circ [\sigma^m_2]) \circ \mathbb{V} [\delta'_1]$. 


We have by precondition that \((\sigma_1' \circ [\sigma_1^m]) \downarrow \approx (\sigma_2' \circ [\sigma_2^m])\). Since 
\([\sigma_1] = (\sigma_1' \circ \sigma_1^m) \circ \delta_1' \approx \delta_1' \circ \delta_1 \approx (\sigma_2' \circ \sigma_2^m) \circ \delta_2' \approx \delta_2' \circ \delta_2\), we have by Lemma 16 also that \((\sigma_1 \downarrow \approx \sigma_2)\). \(\square\)

6.3.4 Criterion for Invariant-Based Confluence Modulo Equivalence

The criteria for the \(\alpha\)- and the \(\beta\)-property can be combined to form an invariant-based confluence modulo equivalence result.

**Theorem 10** (Confluence Modulo \(\approx\) w. r. t. an Invariant). Let \(I\) be an invariant and \(P\) an \(I\)-terminating CHR program. \(P\) has the \(\alpha\) and \(\beta\)-property with respect to \(I\) and an equivalence relation \(\approx\) if and only if \(P\) is \(I\)-confluent modulo \(\approx\).

**Proof.** Theorem 9 is used on the reduced state transition system that only contains states where the invariant holds. \(\square\)

Note that for testing the \(\alpha\)-property, the criterion only assumes a congruence relation, whereas for proving the \(\beta\)-property the split property must hold as well.

**Example 31** (Multi-Set Items (cont.)). It is shown that the program from Example 19 is \(S\)-confluent modulo \(\approx^S\).

\(\alpha\)-property There are several overlaps with more than one mset constraint, like for instance

\[\sigma_1 := \langle mset(L_1), mset(L_2), item(A); \top; \{L_1, L_2, A\} \rangle.\]

Since states like \(\sigma_1\) invalidate the invariant \(I_S\) such that it cannot be re-instantiated by extension, the set \(\Sigma^S(\sigma_1) = \emptyset\) and therefore, there are no minimal elements. The condition of Lemma 17 is then trivially satisfied for those states.

The only non-trivial overlap state is

\[\sigma_2 := \langle item(A), item(B), mset(L); \top; \{L, A, B\} \rangle\]

and satisfies the invariant \(I_S\). Hence, \(M^S(\sigma_2) = \{[\sigma_2]\}\). This leads to the critical pair

\[\langle item(B), mset([A|L]); \top; \{L, A, B\} \rangle,\]
\[\langle item(A), mset([B|L]); \top; \{L, A, B\} \rangle.\]

It can be reduced to

\[[mset([B, A|L]); \top; \{L, A, B\}]\]
\[\approx^S [mset([A, B|L]), \top; \{L, A, B\}].\]
**β-property** The rule state is

\[ \sigma := (\text{item}(A), \text{mset}(L); \top; \{A, L\}) . \]

If \([\sigma] = [\sigma] \circ_{L}\{1\} \{ (\emptyset; \text{perm}(L, L') \land q(L'); \{L, L'\}) \} =: [\sigma'(q)]\) for a built-in constraint \(q/1\), then \([\sigma] \approx^S [\sigma'(q)]\). The preconditions of Lemma 18 are satisfied, since both states satisfy the invariant and hence \(M^{A_0}_S, \approx^S([\sigma]) = M^{A_0}_S, \approx^S([\sigma'(q)]) = \\{([\sigma_\emptyset], [\sigma_\emptyset])\}\). We can show on the meta-level that for all states \([\sigma'(q)]\), the two states reduce to the two equivalent states

\[ ([\text{mset}([A|L]); \text{perm}(L, L') \land q(L'); \{A, L\}]) \]
\[ \approx^S ([\text{mset}([A|L']); \text{perm}(L, L') \land q(L'); \{A, L, L'\}]) \]

by rule application. Since those two final states are equivalent, they are joinable modulo \(\approx^S\).

In Example 31, we use a reasoning on the meta-level about an infinite set of states to show the \(\beta\)-property. In Section 6.4, we show that for the class of \(p\)-state equivalence relations, the reasoning can be moved to the object-level layer again. By monotonicity, the set can be reduced to a finite set of states reestablishing decidability.

The following example demonstrates the necessity of the split property for the confluence test to work.

**Example 32.** Consider the smallest equivalence relation over states that satisfies the following conditions:

- \([\langle c, c; \top; \emptyset \rangle] \approx [\langle d; \top; \emptyset \rangle] \]
- \([\langle c, c, c; \top; \emptyset \rangle] \approx [\langle e; \top; \emptyset \rangle] \]

It is clear that \(\approx\) does not satisfy the split property.

Let \(r : c, c \Rightarrow d\) be the only rule of a CHR program. The \(\alpha\)-property trivially holds. The \(\beta\)-property test succeeds as well: The only rule state is \([\sigma_r] := [\langle c, c; \top; \emptyset \rangle]\). \([\sigma_r]\) is only equivalent to the state \([\sigma_d] := [\langle d; \top; \emptyset \rangle]\) by definition. Since \([\sigma_r] \mapsto_r [\sigma_d]\), the \(\beta\)-property test succeeds.

However, consider the state \([\sigma] := [\langle c, c, c; \top; \emptyset \rangle]\) and its equivalent state \([\sigma'] := [\langle e; \top; \emptyset \rangle]\). By applying \(r\) to \([\sigma]\), we obtain the state \([\langle d, c; \top; \emptyset \rangle]\) that is clearly not equivalent to the final state \([\sigma']\). Hence the two states are not joinable modulo \(\approx\) and the \(\beta\)-property does not hold.

**6.4 Criterion for \(p\)-state equivalence**

For the class of \(p\)-state equivalence relations defined in Definition 42 in Section 5.3.3.2, the \(\beta\)-property test from Lemma 18 typically requires to consider infinitely many equivalent states.
Example 33. The rule from our multi-set items example (c.f. Example 19) has the rule state

\[ \sigma_1 := \langle \text{mset}(L), \text{item}(A); \top; \{L, A\} \rangle. \]

As already discussed in Example 31, it follows that the set of states that are equivalent to \( [\sigma_1] \) is defined as

\[ \{ \langle \text{mset}(L), \text{item}(A); \text{perm}(L, L') \wedge q(L'); \{L, L', A\} \rangle \mid \text{for all built-in constraints } q/1 \text{ such that } \exists L'.(\text{perm}(L, L') \wedge q(L')) \}. \]

This set is potentially infinitely large, because there might be infinitely many such constraints \( q \).

Due to the structure of \( p \)-state equivalence relations, we can still exploit monotonicity to reduce the number of states that have to be considered in the \( \beta \)-property test.

Lemma 19. Let \( \approx \) be a \( p \)-state equivalence relation with user-defined constraint \( c \) and \( \sigma := \langle c(x); \top; \{x\} \rangle \) with \( [\sigma] \mapsto [\sigma_1] \). Then the following two propositions are equivalent:

1. \( [\sigma_1] \Downarrow [\sigma_2] \), where \( [\sigma_1] := \langle c'(x'); p(x, y); \{x, y\} \rangle \) and \( [\sigma_1] \approx [\sigma_2] \equiv \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \).

2. \( [\sigma_1] \Downarrow [\sigma_2] \), where \( [\sigma_2] := \langle c'(x'); p(x, y) \wedge q(y); \{x, y\} \rangle \), where \( q \) is a built-in constraint and \( x \) is a sequence of variables from \( x, y \), for all built-in constraints \( q \) such that \( [\sigma] = \langle \langle c(x); p(x, y) \wedge q(y); \{x\} \rangle \rangle \) and \( [\sigma_2] \approx [\sigma] \).

Proof. \( \Rightarrow \): Assume that \( [\sigma] \mapsto [\sigma_1] \), \( [\sigma] \approx [\sigma_2] \equiv \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \) and \( [\sigma_1] \Downarrow [\sigma_2] \), where \( [\sigma_1] := \langle c'(x'); p(x, y); \{x, y\} \rangle \). From joinability, it follows that there are \( \tau_1, \tau_2 \) such that \( [\sigma_1] \mapsto^* [\tau_1], [\sigma_2] \mapsto^* [\tau_2] \) and \( [\tau_1] \approx [\tau_2] \).

From Definitions 7.2 and 42 it follows that

\[ [\sigma] = [\sigma] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle, \]

since built-in constraints can be duplicated due to idempotence of conjunction. Then, the following transitions are possible:

\[ [\sigma] = [\sigma] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \]
\[ \mapsto [\sigma_1] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \]
\[ \mapsto^* [\tau_1] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \].

Due to Lemma 7, \( [\sigma_2] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \) \( \mapsto^* [\tau_2] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \). From Definition 42 it follows that \( [\tau_1] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \approx [\tau_2] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \). Hence, \( [\sigma_1] \Downarrow [\sigma_2] \circ_{\{y\}} \langle (\mathcal{O}; q(y)); \{x, y\} \rangle \) for all \( q \).
“⇐”: Assume that \([\sigma] \mapsto [\sigma_1], [\sigma] \approx [\sigma_2] \text{ and } [\sigma_1] \downarrow^\approx [\sigma_2]\) for all built-in constraints \(q\) with \([\sigma] = [(c(x); p(x, y) \land q(y)); \{x\}]\).

We choose the built-in constraint \(q\) such that \(CT \models q(\bar{y}) \iff \top\). It follows that \([\sigma] = [(c(x); p(x, \bar{y}); \{x\})]\). From that the hypothesis follows directly.

The lemma basically states that it suffices to show joinability of the smaller state without the fixation of its actual instance to prove joinability for all instances. In other words, monotonicity of the built-in constraint store and the fact that all instances \(p(x, y) \land q(y)\) imply \(p(x, \bar{y})\) guarantees that the two states remain joinable without the information about the actual instance.

We can now formulate a simplified \(\beta\)-property test:

**Corollary 7 (\(\beta\)-Property Test for p-State Equivalence).** Let \(\mathcal{P}\) be a CHR program, \(\mathcal{I}\) an invariant, \(\approx\) a p-state equivalence relation and let \(\mathcal{M}_{\approx}^{L,\infty}([\sigma], [\sigma_2])\) be well-defined for all rule states \([\sigma] \in \mathcal{P}\) and their \(\approx\)-equivalent states \([\sigma_2]\). Let for all rule states \([\sigma]\) hold that \([\sigma] \in \Sigma\) and let for all rule states \([\sigma]\) and all \([\sigma_2] \approx [\sigma]\) hold that if \(([\sigma_1], \_)[I] \in \mathcal{M}_{\approx}^{L,\infty}([\sigma], [\sigma_2]),\) then \([\sigma_1] \in \Sigma\).

Then: \(\mathcal{P}\) has the \(\beta\)-property with respect to \(\mathcal{I}\) and \(\approx\) if and only if for all rule states \([\sigma]\) with successor state \([\sigma_1]\), the \(<\)-smallest state \([\sigma_2]\) with \([\sigma] \approx [\sigma_2]\) and all \(([\sigma_1], [\sigma_2]) \in \mathcal{M}_{\approx}^{L,\infty}([\sigma], [\sigma_2]),\) it holds that \(([\sigma_1] \bowtie [\sigma_2]) \downarrow^\approx ([\sigma_2] \bowtie [\sigma_2]).\)

**Proof.** This follows directly from Lemmas 18 and 19.

By the *smallest state* we denote the equivalent state without constraint \(q\), but only constraint \(p\) of the p-state equivalence relation. This corresponds to the construction of state \(\sigma_2\) in Lemma 19.

**Example 34 (Multi-Set Items (cont.)).** Reconsider Example 31. The proof for of the \(\beta\)-property contained some meta-level analysis. We now use Corollary 7 to prove the \(\beta\)-property. The rule state is

\[\sigma := (\text{item}(A), \text{mset}(L); \top; \{A, L\}).\]

By Corollary 7, it suffices to consider the \(<\)-smallest state \(\sigma'\) with \([\sigma] \approx^S [\sigma']\). This is the case for

\[\sigma' := (\text{item}(A), \text{mset}(L'); \text{perm}(L, L'); \{A, L, L'\}).\]

By rule application on both states, the following can be derived:

\[([\text{mset}([A|L]); \text{perm}(L, L'); \{A, L\})] \approx^S ([\text{mset}([A|L']); \text{perm}(L, L'); \{A, L, L'\})].\]

Since those two final states are equivalent, they are joinable modulo \(\approx^S\). Hence, the program satisfies the \(\beta\)-property according to Corollary 7.
6.5 CRITERION FOR INVARIANT-MAINTAINING EQUIVALENCE RELATIONS

In the general criterion for invariant-based confluence modulo equivalence, the \(\beta\)-property test is complicated, since for all rule states and their equivalent states the set of minimal extensions has to be calculated such that the invariant holds in both extended states and the extended states are equivalent.

This is necessary in general, since there are equivalence relations that do not maintain the invariant, i.e. there are equivalent states where in one state the invariant holds but not in the other state.

**Example 35.** Let \(I_{\#c \geq 1}([s])\) be the invariant that holds if and only if in the state \(s = (G; B; V)\) there is at least one \(c\) constraint in the CHR constraint store \(G\).

Let \(\approx_{cd}\) be the smallest congruence relation such that

\[
[[\{(c) \cup G; B; V\}] \approx_{cd} [[\{(d) \cup G; B; V\}],
\]

i.e. every \(c\) constraint may be replaced by a \(d\) constraint in equivalent states. For instance,

\[
[[s_1] := \langle c; T; \emptyset \rangle \approx_{cd} [[d; T; \emptyset \rangle =: [s_2].
\]

It is clear that \(I_{\#c \geq 1}([s_1])\) holds, whereas \(I_{\#c \geq 1}([s_2])\) does not. It is clear that \(\approx_{cd}\) is fully \(\diamond_V\)-compatible.

Consider the following CHR program that consists of only one rule:

\[
r : d \iff e.
\]

In this program, \(I_{\#c \geq 1}\) is clearly an invariant, since the number of \(c\)-constraints remains unchanged after a rule application.

The \(\alpha\)-property for the program trivially holds according to Lemma 17. For the \(\beta\)-property test with respect to \(I_{\#c \geq 1}\) modulo \(\approx_{cd}\) we have to consider the rule state \(s_r := \langle d; T; \emptyset \rangle\). The only state \(\approx_{cd}\)-equivalent to \([s_r]\) is \(s_r' := \langle c; T; \emptyset \rangle\). Note that \(I_{\#c \geq 1}([s_r])\) does not hold, whereas \(I_{\#c \geq 1}([s_r'])\) holds.

The set of minimal extensions with respect to \(I_{\#c \geq 1}\) and \(\approx_{cd}\) is

\[
M_{I_{\#c \geq 1}, \approx_{cd}}([s_r], [s_r']) = \{([[\langle c; T; \emptyset \rangle], [[\langle d; T; \emptyset \rangle]]), (\langle c; c; T; \emptyset \rangle, [[\langle c; T; \emptyset \rangle]])).
\]

This means, we can either extend \([s_r]\) by an additional \(c\) constraint and \([s_r']\) by a \(d\) constraint or both states by a \(c\) constraint to reestablish the invariant and maintain equivalence. In the first case we obtain

\[
[s_r \diamond \langle c; T; \emptyset \rangle] = [[d, c; T; \emptyset \rangle] \Rightarrow \langle e, c; T; \emptyset \rangle =: [\tau_1]
\]

and

\[
[s_r' \diamond \langle d; T; \emptyset \rangle] = [[c, d; T; \emptyset \rangle] =: [\tau_1']
\]
It is clear that \([\tau_1]\) and \([\tau'_1]\) are joinable modulo \(\approx_{cd}\) by applying \(r\) to \([\tau'_1]\), which yields \([\tau'_1] \mapsto_r ([c; c; \top; \emptyset])\).

The second minimal extension in \(\mathcal{M}^{I_{\geq 1}}_{\approx_{cd}}([\sigma_r], [\sigma'_r])\) yields the following:

\[
[\sigma_r \circ (c; \top; \emptyset)] = [(d, c; \top; \emptyset)] \mapsto (e, c; \top; \emptyset) =: [\tau_2]
\]

and

\[
[\sigma'_r \circ (c; \top; \emptyset)] = [(c, c; \top; \emptyset)] =: [\tau'_2].
\]

The latter state is a final state and therefore \([\tau_2]\) and \([\tau'_2]\) are not joinable modulo \(\approx_{cd}\). Hence, the program does not satisfy the \(\beta\)-property (and is not \(I_{\geq 1}\)-confluent modulo \(\approx_{cd}\)).

However, in practice it is usually desirable that in two equivalent states the invariant either holds or does not hold in both states. Hence, the equivalence relation should maintain the invariant in practice.

**Definition 48 (\(\approx\) Maintains \(I\)).** An invariant \(I\) is maintained by an equivalence relation \(\approx\), if and only if for all states \([\sigma] \approx [\sigma']\) it holds that \(I([\sigma]) \leftrightarrow I([\sigma'])\).

This restriction ensures the practicability of Definition 46, since it may be inelegant and misleading if in a program that is \(I\)-confluent modulo \(\approx\) there exist two equivalent states where one is part of the reduced transition system (i.e. the invariant holds) and the other is not. This may have undesired effects in further analysis. Hence, the invariant and equivalence relation should be chosen such that they are compliant anyway, although it is not required by Definition 46 and the general invariant-based confluence modulo equivalence criterion (c.f. Theorem 10).

In the following, it is presented how an invariant-maintaining equivalence relation can simplify forming the set of pairs of minimal extensions and therefore simplify the \(\beta\)-property test. First of all, it is shown that the sets of minimal extensions of two equivalent states are equivalent if the equivalence relation maintains the invariant.

**Lemma 20 (Unique Minimal Extensions).** Let \(I\) be an invariant and \(\approx\) be a congruence relation with respect to \(\circ\) that maintains \(I\). For two states \([\sigma_1] \approx [\sigma_2]\), it holds that \(\mathcal{M}^I([\sigma_1]) = \mathcal{M}^I([\sigma_2])\).

**Proof.** “\(\Rightarrow\)”: Let \([\sigma_1] \approx [\sigma_2]\) and w.l.o.g. \([\sigma'_m] \in \mathcal{M}^I([\sigma_1])\). Hence, \(I([\sigma_1] \circ [\sigma'_m])\) holds. Since \(\approx\) is a congruence relation with respect to \(\circ\) and \(\approx\) maintains \(I\), it follows that

\([\sigma_1] \circ [\sigma'_m] \approx [\sigma_2] \circ [\sigma'_m]\) and \(I([\sigma_2] \circ [\sigma'_m])\) holds.
Therefore, \([\sigma_1^1] \in \Sigma^T([\sigma_2])\). We now want to show that \([\sigma_1^1] \) is also a \(\prec\)-minimal element of the set of extensions of \([\sigma_2]\), i.e. \([\sigma_1^1] \in \mathcal{M}^T([\sigma_2])\).

Assume that there is an element \([\sigma_2^2] \in \mathcal{M}^T([\sigma_2])\). Since \([\sigma_1^1] \in \Sigma^T([\sigma_2])\), there are three cases:

1. \([\sigma_1^1] \prec [\sigma_2^2]\).

   Since \([\sigma_2^2] \in \mathcal{M}^T([\sigma_2])\) by assumption, it is clear that \([\sigma_2^2] \preceq [\sigma_1^1]\) and hence it follows that \([\sigma_1^1] = [\sigma_2^2]\) by anti-symmetry of \(\prec\) and therefore \([\sigma_2^2] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_1^1] \in \mathcal{M}^T([\sigma_2])\).

2. \([\sigma_2^2] \prec [\sigma_1^1]\).

   Analogously to above, \([\sigma_2^2] \in \Sigma^T([\sigma_1])\). Since \([\sigma_2^2] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_2^2] \in [\sigma_1^1]\) are comparable with respect to \(\prec\), it holds that \([\sigma_1^1] \approx [\sigma_2^2]\). Together with \([\sigma_2^2] \prec [\sigma_1^1]\), it follows that \([\sigma_1^1] = [\sigma_2^2]\) and therefore \([\sigma_2^2] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_1^1] \in \mathcal{M}^T([\sigma_2])\).

3. \([\sigma_1^1] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_2^2] \in \mathcal{M}^T([\sigma_2])\), but neither \([\sigma_1^1] \prec [\sigma_2^2]\) nor \([\sigma_2^2] \prec [\sigma_1^1]\) hold.

   Then the proposition is already shown, since \([\sigma_1^1] \in \mathcal{M}^T([\sigma_2])\).

\(\Leftarrow\): The proof is analogous to the \(\Rightarrow\)-direction.

The following technical observation lifts the result of Lemma 20 to the set of pairs of minimal extensions (according to Definition 47):

**Lemma 21 (Unique Minimal Extensions Modulo \(\approx\)).** Let \(I\) be an invariant, \(\approx\) be a congruence relation with respect to \(\circ\) that maintains \(I\). For two states \([\sigma_1] \approx [\sigma_2]\), it holds that

\[
\mathcal{M}^{I,\approx}([\sigma_1], [\sigma_2]) = \{(\sigma_1^1, \sigma_2^2) \mid \sigma_1^1 \in \mathcal{M}^T([\sigma_1]), \sigma_2^2 \in \mathcal{M}^T([\sigma_2])\}
\]

Proof. \(\Rightarrow\):

Let \([\sigma_1] \approx [\sigma_2]\) and \((\sigma_1^1, \sigma_2^2) \in \mathcal{M}^{I,\approx}([\sigma_1], [\sigma_2])\). By Definitions 22 and 47 and the fact that \(\approx\) maintains the invariant, it follows that \([\sigma_1^1] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_2^2] \in \mathcal{M}^T([\sigma_2])\), i.e. the two components of the tuple are also minimal elements of the individual states.

\(\Leftarrow\):

Let \([\sigma_1] \approx [\sigma_2], [\sigma_1^1] \in \mathcal{M}^T([\sigma_1])\) and \([\sigma_2^2] \in \mathcal{M}^T([\sigma_2])\). Since \([\sigma_1] \approx [\sigma_2]\) and \(\approx\) is a congruence relation maintaining the invariant, it follows by Definition 47 that \((\sigma_1^1, \sigma_2^2) \in \mathcal{M}^{I,\approx}([\sigma_1], [\sigma_2])\).
The equivalence relation \( \approx \) maintains \( \mathcal{I} \) and therefore we can apply Lemma 20. Hence, \( \mathcal{M}^\mathcal{I}(\sigma_1) = \mathcal{M}^\mathcal{I}(\sigma_2) \) and therefore

\[
\{(\sigma_1^1, \sigma_2^1), (\sigma_1^2, \sigma_2^2) \mid (\sigma_1^1, \sigma_2^1) \in \mathcal{M}^\mathcal{I}(\sigma_1) \wedge (\sigma_1^2, \sigma_2^2) \in \mathcal{M}^\mathcal{I}(\sigma_2)\}
\]

\[
= \{(\sigma_1^1, \sigma_2^1), (\sigma_1^2, \sigma_2^2) \mid (\sigma_1^1, \sigma_1^2) \in \mathcal{M}^\mathcal{I}(\sigma_1)\}
\]

\[
= \{(\sigma_1^1, \sigma_2^1), (\sigma_1^2, \sigma_2^2) \mid (\sigma_1^1, \sigma_2^2) \in \mathcal{M}^\mathcal{I}(\sigma_2)\},
\]

i.e. it suffices to only consider the minimal extensions of only one of the states.

With Lemma 21, the \( \beta \)-property test of Lemma 18 can be reformulated:

**Corollary 8** (Simplified \( \beta \)-Property Test). Let \( \mathcal{P} \) be a CHR program, \( \mathcal{I} \) an invariant, \( \approx \) a \( \Sigma \)-compatible equivalence relation with respect to \( \circ \) that maintains \( \mathcal{I} \) and let \( \mathcal{M}^\mathcal{I}(\sigma) \) be well-defined for all rule states \( \sigma \) in \( \mathcal{P} \). Furthermore, let for all rule states \( \sigma \) hold that \( \sigma \in \Sigma \) and if \( \sigma_1, \sigma_2 \in \mathcal{M}^\mathcal{I}(\sigma) \), then \( \sigma_1, \sigma_2 \in \Sigma \). Then: \( \mathcal{P} \) has the \( \beta \)-property with respect to \( \mathcal{I} \) and \( \approx \) if and only if for all rule states \( \sigma \) with successor state \( \sigma_1 \), all \( \sigma_2 \) with \( \sigma \approx \sigma_2 \) and all pairs \( (\sigma_1^1, \sigma_2^1), (\sigma_1^2, \sigma_2^2) \) with \( \sigma_1 \in \mathcal{M}^\mathcal{I}(\sigma) \) such that \( \sigma_1^1 \approx \sigma_2^1 \), it holds that \( (\sigma_1 \circ \sigma_1^1) \downarrow = (\sigma_2 \circ \sigma_2^2) \).

**Proof.** This follows directly from Lemmas 18 and 21.

With this result, the invariant-based confluence modulo equivalence test can be simplified tremendously for equivalence relations that maintain the invariant. In this case it suffices to only consider the minimal elements of all rule states and extend the states equivalent to the rule state with the same minimal elements instead of finding pairs of minimal elements according to Definition 47. This insight is demonstrated in the following example.

**Example 36.** Consider the equivalence relation \( \approx_{cd} \) from Example 35 that allows to replace \( c \) constraints by \( d \) constraints in equivalent states. However, let \( \mathcal{I}_{(c|d) \geq 2} \) be an invariant that holds in a state \( \sigma := (G; B; V) \) if and only if there are at least two constraints of the form \( c \) or \( d \) in the CHR constraint store \( G \) of \( \sigma \). For instance, consider the following states:

- \( \mathcal{I}_{(c|d) \geq 2}(\{(c, c; T; \emptyset)\}) \) holds.
- \( \mathcal{I}_{(c|d) \geq 2}(\{(d, d; T; \emptyset)\}) \) holds.
- \( \mathcal{I}_{(c|d) \geq 2}(\{(c, d; T; \emptyset)\}) \) holds.
- \( \mathcal{I}_{(c|d) \geq 2}(\{(c, e; T; \emptyset)\}) \) does not hold.
- \( \mathcal{I}_{(c|d) \geq 2}(\{(d, e; T; \emptyset)\}) \) does not hold.

It is clear that the invariant \( \mathcal{I}_{(c|d) \geq 2} \) is maintained by the equivalence relation \( \approx_{cd} \).
Consider the following program with the single rule $r : d \Leftrightarrow c$. Clearly, $I_{\#(c;d)\geq 2}$ is an invariant of $P$. The rule state of $r$ is $\sigma_r := (d; \top; \emptyset)$. The only equivalent state is $\sigma'_r := (c; \top; \emptyset)$.

According to Lemma 20, the sets of minimal extensions $[\sigma_r]$ and $[\sigma'_r]$ are equivalent. The CHR constraint stores of both states can either be extended by the constraint $c$ or $d$ to satisfy the invariant, and therefore

$$M_{\#(c;d)\geq 2}^I ([\sigma_r]) = M_{\#(c;d)\geq 2}^I ([\sigma'_r]) = \{ [(c; \top; \emptyset), (d; \top; \emptyset)] \}.$$  

By Lemma 21, the set of minimal extensions with respect to $\approx_{cd}$ of $[\sigma_r]$ and $[\sigma'_r]$ is therefore

$$M_{\#(c;d)\geq 2}^{\approx_{cd}} ([\sigma_r], [\sigma'_r]) = \{ [(\langle c; \top; \emptyset \rangle, \langle c; \top; \emptyset \rangle)] \}.$$  

Let $\tau := (c; \top; \emptyset)$ with $[\sigma_r] \rightarrow^r [\tau]$ be the successor state of $[\sigma_r]$. To prove the $\beta$-property, it has to be shown that for all $([\sigma^1_m], [\sigma^2_m]) \in M_{\#(c;d)\geq 2}^{\approx_{cd}} ([\sigma_r], [\sigma'_r])$ the states $[\tau] \odot [\sigma^1_m]$ and $[\sigma'_r] \odot [\sigma^2_m]$ are joinable according to Corollary 8.

It can be seen that the program is $\mathcal{I}_{\#(c;d)\geq 2}$-confluent modulo $\approx_{cd}$. Consider, e.g., the pair of minimal extensions $\{ [(c; \top; \emptyset)], [(c; \top; \emptyset)] \}$. The two final states $[(c, c; \top; \emptyset)]$ and $[(c, c; \top; \emptyset)]$ are obviously not joinable with respect to $\approx_{cd}$, since they are even state equivalent. The same situation can be shown for all other extensions.

Note that the rule $r$ already reflects the equivalence of $c$ and $d$. In a related example for the rule $r' : d \Leftrightarrow e$ without an invariant, we would find that it lacks confluence modulo $\approx_{cd}$ because the rule state $\sigma_{r'} := (d; \top; \emptyset)$ has successor state $\tau := (c; \top; \emptyset)$. However, the state $\sigma'_{r'} := (c; \top; \emptyset)$ with $[\sigma'_{r'}] = [\sigma_r]$ is a final state with $[\sigma'_{r'}] \not\approx_{cd} [\tau]$ and, hence, the $\beta$-property is not satisfied. The program could be completed by a rule $c \Leftrightarrow e$ that again reflects the $\approx_{cd}$-equivalence of $c$ and $d$.

### 6.6 Discussion

In this section, we first discuss the implications of our results in this chapter. We then compare the updated results in this thesis to our prior work. Eventually, the notion of confluence modulo equivalence is related to the popular formalization of confluence of CHR often found in the literature.

#### 6.6.1 Implications of the Results

The work in this section shows that to prove invariant-based confluence modulo a user-defined equivalence relation, it is required to show the $\alpha$- and the $\beta$-property.
The $\alpha$-property test requires the user-defined equivalence relation to be a congruence relation with respect to $\diamond_V$. Then, for all overlap states $\sigma$ with critical pairs $(\sigma_1, \sigma_2)$, we have to show that for all minimal extensions $\sigma_m$ of $\sigma$, the extended critical pairs $(\sigma_1 \diamond \sigma_m, \sigma_2 \diamond \sigma_m)$ are joinable modulo the equivalence relation.

The $\alpha$ is decidable for terminating programs as long as the invariant and the equivalence relation (a congruence relation with respect to $\diamond_V$) are decidable and the set of minimal extensions is well-defined and finite. It coincides with the traditional invariant-based confluence test except for the relaxation that the critical pairs only have to be joinable modulo the user-defined equivalence relation instead of the stricter state equivalence.

The $\beta$-property test is far more difficult to prove: First of all, it requires to be the user-defined equivalence relation to be $\Sigma$-compatible with respect to $\diamond_V$, where $\Sigma$ contains at least all the rule states. Then, for all rule states $[\sigma]$ we have to show that for all equivalent states $[\sigma']$ of $[\sigma]$ and for all pairs $([\sigma_m], [\sigma'_m])$ of minimal extensions of $[\sigma]$ and $[\sigma']$ that extend $[\sigma]$ and $[\sigma']$ such that $[\sigma] \diamond [\sigma_m] \approx [\sigma'] \diamond [\sigma'_m]$ and the invariant holds in both extended states, the successor state $[\tau]$ of $[\sigma]$ and $[\sigma']$ are joinable modulo the user-defined equivalence relation.

Unfortunately, already the set of states equivalent to the rule state may be infinitely large in general. Furthermore, as for the $\alpha$-property it depends on the invariant if there is a finite number of minimal extensions of those states such that the invariant holds (and the extended states are equivalent).

Nevertheless, for the class of $p$-state equivalence relations, the number of equivalent states is finite. This makes the $\beta$-property test decidable. Since this class already contains many interesting equivalence relations, the invariant-based confluence modulo equivalence theorem is applicable and even decidable, although the requirement of a $\Sigma$-compatible equivalence relation with respect to $\diamond_V$ seems very restricting at first glance.

For equivalence relations that maintain the invariant, the construction of the pairs of minimal extensions is simplified significantly, since only combinations of the unique set of minimal extensions of the individual states (that are equivalent for both states) have to be considered.

Although the special cases substantially simplify reasoning about invariant-based confluence modulo equivalence, we want to point out that the most general formulation of the test in Lemma 18 still is applicable to more exotic equivalence relations, albeit at the cost of losing decidability in general.
6.6.2 Relation to Prior Work

This chapter described two criteria to prove invariant-based confluence modulo equivalence for CHR programs. The criterion in Theorem 10 is the most general form of the confluence modulo equivalence test that only restricts the equivalence relation to be $\odot$-compatible. The criterion in Section 6.5 is a specialization of Theorem 10 that shows that the $\beta$-property test can be simplified, if the equivalence relation additionally maintains the invariant (c. f. Corollary 8). Both results are published in this thesis for the first time.

A third variant of the invariant-based confluence modulo equivalence criterion has been presented in [GF18b]. It was the first description of our results on the topic. Similar to the criterion in Section 6.5, the test required the equivalence relation to be $\odot$-compatible and to maintain the invariant. To prove the $\beta$-property, the theorem from [GF18b] uses the following criterion (shortened): For all rules $r$ with rule states $[\sigma]$ and $[\sigma] \rightarrow [\tau]$, all their equivalent states $[\sigma'] \approx [\sigma]$ it suffices to show that the states $[\tau] \odot [\sigma_1^m]$ and $[\sigma'] \odot [\sigma_2^m]$ are joinable, where $[\sigma_1^m] \in \mathcal{M}^I([\sigma])$ are the minimal extensions of the rule state that reestablish the invariant $I$ and $[\sigma_2^m]$ are all states $\approx$-equivalent to $[\sigma_1^m]$.

The criterion in Theorem 10 is a generalization of the result in [GF18b], as it does not require the equivalence relation to maintain the invariant, i.e. it is applicable to a broader set of CHR programs. This is possible by forming the set of pairs of minimal extensions of the rule state and its equivalent states $\mathcal{M}^I,\approx ([\sigma], [\sigma'])$. Section 6.5 shows that for equivalence relations that maintain the invariant, the two sets of minimal extensions $\mathcal{M}^I([\sigma])$ and $\mathcal{M}^I([\sigma'])$ are equivalent. Therefore, the set $\mathcal{M}^I,\approx ([\sigma], [\sigma'])$ can be constructed by forming all tuples of minimal extensions of one of the states. This leads to the same set of states as in the general criterion in Theorem 10 and the version in [GF18b], but the work in this thesis defines a process of how those states can be found from the set of minimal extensions of the rule state $\mathcal{M}^I([\sigma])$. Furthermore, the proof is simplified as it just uses the general criterion from Theorem 10 and its formal structures. The insight that for equivalence relations that maintain the invariant the sets of minimal extensions of two $\approx$-equivalent states are equivalent (i.e. the members are state equivalent) was unknown to the time where [GF18b] was published and is a further contribution of this thesis.

Furthermore, in [GF18b], it is implicitly assumed that the equivalence relation is also $\odot_V$ monotonic in the $\beta$-property. The corrected Lemma 18 in this thesis makes this fact explicit.

The class of $p$-state equivalence relations was unknown to the time of the publication of [GF18b] and is a new contribution of this thesis. Additionally, the result of the simplified $\beta$-property test for $p$-state equivalence relations in Corollary 7 shows that for this class of equiv-
alence relations the β-property test is decidable for terminating programs (if the set of minimal extensions is finite). The insight that there exists an interesting class of equivalence relations which makes the invariant-based confluence test decidable for terminating programs is another original contribution of this thesis.

6.6.3 Comparison of Confluence Definitions

In CHR literature, confluence is often implicitly defined as confluence modulo state equivalence (c.f. Definition 16):

“A CHR program is confluent if for all states σ, τ₁, τ₂: If σ →* τ₁ and σ →* τ₂, then there exist states τ'₁, τ'₂ such that τ₁ →* τ'₁ and τ₂ →* τ'₂ and τ'₁ ≡ τ'₂” [Frü09, p. 102].

The reason that the definition of confluence modulo state equivalence and traditional confluence coincide for CHR lies in the semantic compliance of state equivalence with respect to ωva and ωe. The definition of ωe modulo state equivalence, i.e. over equivalence classes, as in Definition 13, already suggests that the definitions coincide, as already mentioned in [RBF09, p. 13].

Nevertheless, it is not directly clear why the canonical confluence test for CHR only considers the α-property. The reason is that the β-property is satisfied for state equivalence due to semantic compliance: According to Definition 45, the β-property for state equivalence requires that for all states σ, τ, τ' ∈ ΣCHR it holds that if σ →rτ and σ ≡ σ' then τ and σ' are joinable modulo ≡. Due to semantic compliance (c.f. Theorem 4), there is a state τ' ≡ τ that is reached by applying r to σ', i.e. σ' →r τ'. Hence, σ and σ' are joinable modulo state equivalence. In other words, semantic compliance of state equivalence is stronger than the β-property modulo state equivalence.

For an empty invariant and the user-defined equivalence relation of ≡, the α-property test coincides with the traditional confluence criterion from Theorem 5. It also coincides with the β-property test, since the only state equivalent to the rule state is the (equivalence class of the) state itself. The set of minimal extensions is empty, since the invariant trivially holds. The state that is obtained by applying the rule to the original state and the original state itself are trivially joinable modulo state equivalence (by applying the exact same rule). Hence, our new analysis methods coincide with the traditional criterion which supports their validity.

6.7 Related Work

The foundations of our work on confluence modulo equivalence originate from the domain of term rewriting systems [Hue80]. In this reference, the foundations of reasoning about confluence modulo
equivalence are set up. This includes the notion of local confluence modulo equivalence, i.e. the $\alpha$- and $\beta$-properties and the important Theorem of Huet (c.f. Theorem 9) that serves as a basis for our central invariant-based confluence modulo equivalence theorem (c.f. Theorem 10).

We combine those ideas with the notion of observable confluence [DSS06; DSS07] or invariant-based confluence [Rai10]. The general idea of the subsumption of states with a partial order comes from [DSS06; DSS07]. The partial order $\prec$ that improves the partial order of [DSS06; DSS07] originates from [Rai10] and is crucial for our considerations.

Invariant-based confluence modulo equivalence has first been investigated for CHR in [CK15; CK17]. One significant difference is that the results in [CK17] refer to an own operational semantics that introduces non-logical and incomplete built-in constraints, i.e. built-in constraints that are either non-monotonic or may lead to exceptions like $\text{var}/1$ or $\text{is}/2$, respectively. To reason about such states that are close to actual CHR implementations, a meta-language is developed that summarizes the states in sets. In [CK17], the central part of the work then establishes the analysis methods for this meta-language.

The delimitation of our work against [CK15; CK17] with respect to the meta-language and the general results about reasoning modulo user-defined equivalence relations in CHR is already covered in Section 5.5. We now concentrate on the invariant-based confluence modulo equivalence test.

In contrast to [CK17], our work is settled in the ecosystem of the canonical operational semantics $\omega_c$ that is equivalent to the very abstract semantics $\omega_{\text{va}}$. This makes our results directly available to all standard CHR programs.

Furthermore, the work in [CK17] establishes a pattern of user-defined equivalence relations that is similar to the class of $p$-state equivalence relations. We have formalized this pattern on the object level in Chapter 5 and have shown the important properties of this class of relations. The properties are generalized as the notion of $\Sigma$-compatibility with respect to $\Diamond$ that is necessary to maintain reasoning about CHR states by exploiting monotonicity. This makes our results available to all equivalence relations that are $\Sigma$-compatible (and not only $p$-state equivalence relations). Furthermore, we make some interesting observations about more exotic equivalence relations in Section 5.3.2 where we investigate some implications of the compatibility property.

In [CK17], the proposed pattern of equivalence relations is not further investigated. The proposed criteria are formulated on a less concrete level than our $\alpha$- and $\beta$-property: For the so-called abstract joinability diagrams it is not described how they can be proven and the proof depends on the equivalence relation. Hence, the sets of states represented by an abstract joinability diagram may be arbitrarily
complex. The examples presented in [CK17] all are fully compatible with respect to $\diamondsuit_Y$. This property is exploited by the authors in their proofs of the abstract joinability diagrams.
The question if two programs are operationally equivalent is fundamental for program analysis. For instance, for automated program repair or refactoring of code it is important that the automatic transformations maintain the semantics of a program. Hereby, operational equivalence denotes that two programs behave in the same way, i.e. yield the same result for every input. Another application is the removal of redundant rules without changing the operational semantics. This can be a useful application especially for automatically generated programs.

For CHR there is a decidable sufficient and necessary criterion for operational equivalence of terminating and confluent programs [Frü09; AF99]. The criterion generates a finite set of states – the rule states – from the source code of both programs and compares the output of programs for those input states.

However, in practice, the notion of operational equivalence is often too strict. For instance, if the constraints in a program are renamed, the resulting program is not considered equivalent to the original program, although it essentially behaves the same. Therefore, various methods have been introduced that allow to compare CHR programs with different signatures, for instance operational $c$-equivalence for a constraint $c$ [AF99; Frü09, pp. 129–132; Rai10].

Furthermore, analogously to the canonical definition of confluence, strict operational equivalence also considers states that can never be reached in practice. Therefore, invariants on CHR states have been introduced that restrict the considered states for operational equivalence to only those that are relevant in practice [Rai10]. There is a criterion for so-called invariant-based operational equivalence that is decidable for terminating and confluent programs depending on the invariant [Rai10].

In this chapter, we introduce a generalization of operational equivalence based on the idea of confluence modulo equivalence. The fundamental concept is a user-defined equivalence relation over states that weakens the conditions for state equivalence. Together with invariants, this concept of invariant-based operational equivalence modulo equivalence subsumes the other definitions, including operational equivalence for programs with different signatures. The most important extension of our definition of invariant-based operational equivalence modulo equivalence is that it introduces user-defined equivalence relations. This allows to handle more general concepts of state equivalence, for
The main contribution of this chapter is a sufficient and necessary decision criterion for our novel definition of invariant-based operational equivalence modulo an equivalence relation for programs that are terminating and confluent modulo this equivalence relation. The criterion requires the equivalence relation to maintain the invariant, i.e., that the invariant holds in a state if and only if it holds in all equivalent states. Furthermore, analogously to the confluence modulo equivalence criterion in Chapter 10, the equivalence relation has to be (fully) $\Diamond_V$-compatible to ensure that the monotonicity property of CHR can be exploited. The criterion is decidable for a subset of invariants that is common in works on invariant-based confluence as it was the case in Section 3.3.7 and Chapter 6 [DSS07; DSS06; Rai10].

The chapter is structured as follows: Section 7.1 introduces the notion of invariant-based operational equivalence modulo equivalence while Section 7.2 presents the sufficient and necessary criterion deciding it. The result is discussed in Section 7.3.

### 7.1 Definition

This section introduces the notion of invariant-based operational equivalence modulo equivalence formally for arbitrary state transition systems. Since many programs that are operationally equivalent in practice might introduce different temporary or intermediate constraints, the idea emerged that it can be useful to restrict the considered input and output states to only members of a defined interface between the programs, as already explained in Section 3.3.7.3. We generalize the definition there as follows:

**Definition 49 (Interface States [Rai10, pp. 92–93]).** The set of constraint symbols $I$ is an interface of two CHR programs $P_1, P_2$, if and only if $I \subseteq C(P_1) \cap C(P_2)$ where $C(P)$ denotes the set of constraint symbols in a program $P$. [Rai10, p. 92]

A state $\sigma = \langle G; B; V \rangle$ is called an $I$-state, i.e., an interface state for the interface $I$ if and only if all constraint symbols in $G$ occur in $I$. The definition can straightforwardly be lifted to rule states and $\equiv$-equivalence classes of states. In the latter case, the equivalence class of failed states is considered an $I$-state for all interfaces $I$. [Rai10, p. 93]

Note that compared to Definitions 24 and 25 from [Rai10], we generalize the notion of interface to a subset of shared constraint symbols instead of exactly the shared symbols, since all definitions and proofs still work. The same effect can be achieved by renaming constraints in both programs and using the original definition.

In the following, invariant-based operational equivalence modulo equivalence is defined formally. It connects the previous definitions of
append macro step A

In this section a criterion for invariant-based operational equivalence is the concatenation of \( L_1 \) and, by Lemma 8, let \( \sigma_1 \approx \sigma_2 \) and \( I([\sigma_1]) \approx I([\sigma_2]) \). The proof of the criterion is based on so-called macro step induction [AF99; Rai10].

**Definition 51 (Macro Step [Rai10, p. 94]).** Let \( \sigma_r \) be a rule state of \( r \) and let \( \sigma_r \rightarrow^+ \sigma_f \) for a final state \( \sigma_f \). Let \( \sigma \) be a state where \( r \) is applicable and, by Lemma 8, let \( \sigma = [\sigma_r] \circ \delta \) where \( \delta \) are the global variables of \( \sigma_r \). A macro step of \( \sigma \) is a computation of the form \( \sigma \rightarrow^+ [\sigma_r] \circ \delta \).

The proof technique of macro step induction is used.
The following theorem is the main theorem of this chapter. It gives a decidable sufficient and necessary criterion for operational $\mathcal{I}$-$\mathcal{I}$-equivalence modulo $\mathcal{I}$-confluent modulo $\approx$. The idea is to test if all (extended) rule states lead to equivalent final states in both programs. The set of minimal extensions $\mathcal{M}^\mathcal{I}(\{\sigma\})$ is required to be well-defined according to Definition 22 and finite for all $\mathcal{I}$-rule states $[\sigma]$. The equivalence relation $\approx$ has to be $\circ_{\mathcal{V}}$-compatible and has to maintain the invariant $\mathcal{I}$ for all $\mathcal{I}$-rule states.

**Theorem 11 (Deciding Operational Equivalence Modulo Equivalence).**

Let $\mathcal{I}$ be a set of constraints and $\mathcal{I}$ an invariant, for which $\mathcal{M}^\mathcal{I}$ is well-defined for all $\mathcal{I}$-rule states. Let $\approx$ be a fully $\circ_{\mathcal{V}}$-compatible equivalence relation that maintains $\mathcal{I}$ for all $\mathcal{I}$-rule states. Let $\mathcal{P}_1, \mathcal{P}_2$ be two terminating and $\mathcal{I}$-confluent modulo $\approx$ CHR programs with interface $\mathcal{I}$.

$\mathcal{P}_1$ and $\mathcal{P}_2$ are $\mathcal{I}$-$\mathcal{I}$-equivalent modulo $\approx$ if and only if for all $\mathcal{I}$-rule states $\sigma_r$ of $\mathcal{P}_1$ and $\mathcal{P}_2$ and all $[\sigma_m] \in \mathcal{M}^\mathcal{I}(\{\sigma_r\})$ holds that $[\sigma_r] \circ [\sigma_m] \rightarrow^{*}_{\mathcal{P}_1} [\sigma_1^1] \not\rightarrow \mathcal{P}_1$ and $[\sigma_r] \circ [\sigma_m] \rightarrow^{*}_{\mathcal{P}_2} [\sigma_2^2] \not\rightarrow \mathcal{P}_2$, $[\sigma_1^1] \approx [\sigma_2^2]$ and $[\sigma_1^1], [\sigma_2^2]$ are $\mathcal{I}$-states.

**Proof.** $\Rightarrow$:

Let $\mathcal{P}_1$ and $\mathcal{P}_2$ be two $\mathcal{I}$-terminating CHR programs that are $\mathcal{I}$-$\mathcal{I}$-equivalent modulo $\approx$ and let $\sigma_r$ be an $\mathcal{I}$-rule state of $\mathcal{P}_1$ or $\mathcal{P}_2$. There are two cases:

1. $\mathcal{I}(\{\sigma_r\})$ holds. Then $\mathcal{M}^\mathcal{I}(\{\sigma_r\}) = \{[\sigma_0]\}$ according to Lemma 9. Since $[\sigma_0]$ is the neutral element of $\circ$, it holds that $[\sigma_r] \circ [\sigma_0] = [\sigma_r]$ (c.f. Definition 14). $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\mathcal{I}$-$\mathcal{I}$-equivalent modulo $\approx$ and therefore

$[\sigma_r] \rightarrow^{*}_{\mathcal{P}_1} [\tau^1], \text{ and}$

$[\sigma_r] \rightarrow^{*}_{\mathcal{P}_2} [\tau^2]$

where $[\tau^1] \approx [\tau^2]$ according to Definition 50.

2. $\mathcal{I}(\{\sigma_r\})$ does not hold. For all $[\sigma_m] \in \mathcal{M}^\mathcal{I}(\{\sigma_r\})$, the invariant $\mathcal{I}(\{\sigma_r \circ [\sigma_m]\})$ holds. Since $\mathcal{P}_1$ and $\mathcal{P}_2$ are $\mathcal{I}$-$\mathcal{I}$-equivalent modulo $\approx$, it is clear that

$[\sigma_r] \circ [\sigma_m] \rightarrow^{*}_{\mathcal{P}_1} [\tau^1], \text{ and}$

$[\sigma_r] \circ [\sigma_m] \rightarrow^{*}_{\mathcal{P}_2} [\tau^2]$

where $[\tau^1] \approx [\tau^2]$ according to Definition 50.

$\Leftarrow$:

The proof uses induction over the number of macro steps. It has to be shown that for all $\mathcal{I}$-states $\sigma$ where the invariant $\mathcal{I}(\{\sigma\})$ holds that for all $[\sigma'] \approx [\sigma]$ the final states $[\sigma_i^i], [\sigma_i^f]$ with $[\sigma] \rightarrow^{*}_{\mathcal{P}_1} [\sigma_i^i] \not\rightarrow \mathcal{P}_1$ and $[\sigma'] \rightarrow^{*}_{\mathcal{P}_2} [\sigma_i^f] \not\rightarrow \mathcal{P}_2$, it holds that $[\sigma_i^i] \approx [\sigma_i^f]$ and the final states only use constraint symbols from $\mathcal{I}$.
Let the \( [\sigma^2] \) be a final state of \( [\sigma^1] \) in \( \mathcal{P}_1 \), i.e. \( [\sigma^1] \rightarrow^*_\mathcal{P}_1 [\sigma^2] \not\twoheadrightarrow \mathcal{P}_1 \), then \( [\sigma^1] \rightarrow^*_\mathcal{P}_1 [\sigma^2] \) with a finite number of macro steps. By induction over the number of macro steps it is proven that for all \( [\sigma^2] \) with \( [\sigma^1] \approx [\sigma^2] \): \( [\sigma^1] \approx [\sigma^2] \) where \( [\sigma^2] \rightarrow^*_\mathcal{P}_2 [\sigma^2] \not\twoheadrightarrow \mathcal{P}_2 \). This means that the final state for \( [\sigma^1] \) in program \( \mathcal{P}_1 \) is equivalent to the final state for all equivalent states \( [\sigma^2] \) in \( \mathcal{P}_2 \). In fact, the following slightly stronger hypothesis is proven:

**Induction Hypothesis** For \( |\ast\rangle \text{-states} \ [\sigma^1] \) and \( [\delta^1] \) with \( [\sigma^1] \rightarrow^*_\mathcal{P}_1 [\delta^1] \) and all states \( [\sigma^2] \approx [\sigma^1] \) with \( [\sigma^2] \rightarrow^*_\mathcal{P}_2 [\delta^2] \) and \( [\delta^1] \approx [\sigma^2] \) it holds that for all \( [\delta^1], [\delta^2] \) with \( [\sigma^1] \rightarrow^*_\mathcal{P}_1 [\delta^1] \) and \( [\sigma^2] \rightarrow^*_\mathcal{P}_2 [\delta^2] \), \( [\delta^1], [\delta^2] \) are \( |\ast\rangle \text{-states} \).

The hypothesis is stronger than the one from above, since \( [\delta^1] \) and \( [\delta^2] \) are not required to be final states.

**Base Case** In the base case with 0 macro steps, \( [\sigma^1] = [\sigma^1] \) where \( [\sigma^1] \not\twoheadrightarrow \mathcal{P}_1 \). By contradiction it is shown that \( [\sigma^1] \approx [\sigma^2] \) for all final states \( [\sigma^2] \) of equivalent states \( [\sigma^2] \) in \( \mathcal{P}_2 \), i.e. where \( [\sigma^1] \approx [\sigma^2] \) and \( [\sigma^2] \rightarrow^*_\mathcal{P}_2 [\sigma^2] \not\twoheadrightarrow \mathcal{P}_2 \).

Let \( [\sigma^1] \approx [\sigma^2] \) and \( [\sigma^2] \rightarrow^*_\mathcal{P}_2 [\tau] \) for some rule \( \tau \). This is possible, as we assume that \( [\sigma^1] \) is not a final state in \( \mathcal{P}_2 \) and hence due to confluence modulo \( \approx \) of \( \mathcal{P}_2 \), neither is \( [\sigma^2] \). By Lemma 8, \( [\sigma^2] \) can be split into the rule state \( \sigma_r \) and some rest \( [\delta] \):

\[ \exists [\delta], [\sigma^2] = [\sigma_r] \circ \mathcal{V} [\delta] \]

where \( \mathcal{V} \) are the global variables of the rule state \( \sigma_r \), i.e. all variables of rule \( r \).

As \( \mathcal{I}([\sigma^2]) \) holds, it is clear that \( [\delta] \) is in the set of states that extend \( [\sigma_r] \) such that the invariant holds, i.e. \( [\delta] \in \Sigma^\mathcal{I}([\sigma_r]) \).

Therefore, it exists a minimal element \( [\sigma_m] \) of this set of extensions \( \Sigma^\mathcal{I}([\sigma_r]) \):

\[ \exists [\sigma_m] \in \mathcal{M}^\mathcal{I}([\sigma_r]), [\sigma_m] \triangleleft [\delta]. \]

By definition of \( \triangleleft \) and Lemma 4:

\[ [\sigma^2] = [\sigma_r] \circ \mathcal{V} ([\sigma_m] \circ [\delta]) \]
\[ = ([\sigma_r] \circ [\sigma_m]) \circ \mathcal{V} [\delta] \]

By precondition, we have that \( [\sigma_r] \circ [\sigma_m] \) yield equivalent final states, i.e. \( [\sigma_r] \circ [\sigma_m] \rightarrow^*_\mathcal{P}_1 [\rho^1] \not\twoheadrightarrow \mathcal{P}_1 \) and \( [\sigma_r] \circ [\sigma_m] \rightarrow^*_\mathcal{P}_2 [\rho^2] \not\twoheadrightarrow \mathcal{P}_2 \) with \( [\rho^1] \approx [\rho^2] \). Since \( \mathcal{P}_1 \) and \( \mathcal{P}_2 \) are confluent modulo \( \approx \), this holds for all \( [\rho^1] \) and \( [\rho^2] \).

By construction \( [\sigma_r] \circ [\sigma_m] \) is already a final state in \( \mathcal{P}_1 \) and hence \( [\rho^1] \approx [\sigma_r \circ \sigma_m] \). However, \( [\sigma_r] \circ [\sigma_m] \) is not a final state in \( \mathcal{P}_2 \), i.e. \( [\rho^2] \not\approx [\rho^1] \). This is a contradiction.
**Induction Step** If there are one or more macro steps, some rule \( r \) is applicable in \([σ^1]\) and hence \([σ^1]_m\) can be split according to Lemma 8:

\[
\exists[δ].[σ^1] = [σ^1]_r \circ V [δ]
\]

where \( V \) are the global variables from the rule state \( σ_r \), i.e. all variables appearing in rule \( r \).

As \( I([σ^1]) \) holds, i.e. the invariant is satisfied in \( σ^1 \), there is a minimal extension \([σ^1_m]\) of \([σ^1]\) such that the invariant holds. Formally, this means that

\[
\exists[σ^1_m]_m \in M_2^f([σ^1]), [σ^1_m]_m \prec [δ]\text{ and}
\]

\[
[σ^1] = ([σ^1_r]_r \circ [σ^1_m]) \circ V [σ^1],
\]

using Lemma 4 as above. It is clear that all states \( σ^1, σ^1_r, σ^1_m, δ^1 \) and \( δ \) are \( It \)-states.

Let w.l.o.g. \([σ^1_r]\) be a final state of \([σ^1_r \circ σ^1_m]\) in \( P_1 \), i.e. \([σ^1_r \circ σ^1_m] \mapsto^*_1 [σ^1_r] \not\rightarrow_{P_1} \). Such a state always exists and is unique modulo \( \approx \) as \( P_1 \) is terminating and confluent modulo \( \approx \). Nevertheless, \([σ^1_r]\) might be syntactically different from other final states of \([σ^1_r \circ σ^1_m]\). Since \([σ^1_r]\) is chosen arbitrarily, we will see in the following that it suffices that it is unique modulo \( \approx \).

By Lemma 7 (monotonicity):

\[
[σ^1] \mapsto^*_1 [σ^1_r] \circ [δ^1].
\]

W.l.o.g., let \([σ^1_m]\) be a final state of \([σ^1_r \circ σ^1_m]\) in \( P_2 \). Therefore, by precondition:

\[
[σ^1_r] \circ [σ^1_m] \mapsto^*_1 [σ^1_r] \not\rightarrow_{P_1},
\]

\[
[σ^1_r] \circ [σ^1_m] \mapsto^*_2 [σ^1_m] \not\rightarrow_{P_2}, \text{ and}
\]

\[
[σ^1_r] \approx [σ^1_m].
\]

Again, \([σ^1_m]\) exists and is unique modulo \( \approx \) due to termination and confluence of \( P_2 \). It is clear that \([σ^1_r]\) and \([σ^1_m]\) satisfy the invariant, since \([σ^1_r] \circ [σ^1_m]\) does.

Due to the split property of \( \approx \) (c.f. Definitions 37 and 39), it holds that for all \([σ^2]\) with \([σ^2] \approx [σ^1]\) there exist \([σ^2_r]\), \([σ^2_m]\) and \([δ^2]\) such that

\[
[σ^2] = ([σ^2_r] \circ [σ^2_m]) \circ V [δ^2]
\]

where \([σ^2_r] \approx [σ^2_r], [σ^2_m] \approx [σ^2_m], \) and \([δ^1] \approx [δ^2] \).

Hence, \([σ^2_r] \circ [σ^2_m] \approx [σ^2_m] \circ [σ^2_m] \) and therefore \( I([σ^2_r] \circ [σ^2_m]) \) holds as \( I \) is \( \approx \)-compatible. Since \( P_2 \) is \( I \)-confluent modulo \( \approx \), it follows that \([σ^2_r] \circ [σ^2_m] \mapsto^*_2 [σ^1_m] \not\rightarrow_{P_2} \). Due to Lemma 7 (monotonicity), it is possible to apply the same rules in the larger state \([σ^2]\):

\[
[σ^2] \mapsto^*_2 [σ^1_m] \circ V [δ^2].
\]
Note that due to $I$-confluence modulo $\approx$, $[\sigma'''] \circ_V [\delta^2]$ is unique for all states $\approx$-equivalent to $[\sigma^2]$ where $I([\sigma^2])$ holds.

In the last steps, we first observed the behavior of the extended rule state $[\sigma] \circ [\sigma_m]$ in both states that lead to $\approx$-equivalent final states $[\sigma'_1]$ and $[\sigma''_1]$. Then we used the split property to dissect a state $[\sigma^2]$ $\approx$-equivalent to $[\sigma^1]$ into parts that are $\approx$-equivalent to the dissection of $[\sigma^1]$. Since $P_2$ is confluent modulo $\approx$, we know that the result of $([\sigma^2] \circ [\sigma^2_m])$ is equivalent to $([\sigma'_1] \circ [\sigma^1_m])$, i.e. $[\sigma''_1]$. Due to monotonicity, we can just replace the part $([\sigma^2] \circ [\sigma^2_m])$ by its result to obtain the computation of one macro step on $[\sigma^2]$.

It is now possible to apply the induction hypothesis on $[\sigma'_1] \circ_V [\delta^1]$ and $[\sigma''_1] \circ_V [\delta^2]$, as one macro step has been unfolded and clearly $[\sigma'_1] \circ_V [\delta^1] \approx [\sigma''_1] \circ_V [\delta^2]$. This leads to:

$$[\sigma'_1] \circ_V [\delta^1] \mapsto_{P_1} [\sigma'_1] \not\approx,$$
$$[\sigma''_1] \circ_V [\delta^2] \mapsto_{P_2} [\sigma''_1] \not\approx,$$ and

$$[\sigma'_1] \approx [\sigma''_1].$$

where $[\sigma'_1]$ and $[\sigma''_1]$ are $I$-states.

□

The idea of the only-if direction is to show that for all $I$-states $\sigma$ and states $\sigma'' \approx [\sigma]$ where the invariant $I([\sigma])$ holds that the final states in both programs are equivalent and $I$-states.

Therefore, an arbitrary state $[\sigma]$ is split into its rule state $[\sigma_r]$ and some rest. The computations of the rule state lead to equivalent final states in each program by precondition. All states $[\sigma'']$ equivalent to $[\sigma]$ can be split into parts equivalent to the splitting of $[\sigma]$ due to the split property, i.e. there is a $[\sigma''] \approx [\sigma_r]$. Since the invariant holds in $[\sigma'_1]$ by precondition, its final state leads to an equivalent final state in $P_2$ due to $I$-confluence modulo $\approx$ of $P_2$. Due to monotonicity, this also holds for the larger state $[\sigma'']$. By applying the induction hypothesis, it can be shown that both final states are equivalent.

**Example 38 (Multi-Sets).** We go back to the two programs from Examples 19 and 37. Both programs are confluent with respect to the invariant $I_5$ modulo $\approx^S$. The fully $\approx^S$-compatible equivalence relation $\approx^S$ maintains the invariant: For two states $\sigma_1, \sigma_2$ with $\sigma_1 \approx^S \sigma_2$, it is clear that they have the same number of mset $\sigma$ constraints by Definition 41. Hence, $I_5([\sigma_1]) \iff I_5([\sigma_2])$.

We now construct the set of minimal extensions. A state where the invariant does not hold cannot be extended to satisfy the invariant, therefore $M^I_5([\sigma]) = \emptyset$ for all $[\sigma]$ where $I_5([\sigma])$ does not hold. For all other states the only minimal extension is the empty state $[\sigma_0]$ (c.f. Lemma 9). Therefore, the set of minimal elements is well-defined. Hence, the preconditions of Theorem 11 are satisfied.
Both programs have only the single rule state
\[ \sigma_1 = \sigma_2 = \langle \text{mset}(L), \text{item}(A); \top; \{L, A\} \rangle. \]

The state satisfies the invariant, i.e. \( \mathcal{I}_S(\sigma_i) \) holds for \( i = 1, 2 \). Furthermore, the interface is defined as \( \mathbb{I}_S = \{ \text{mset}/1, \text{item}/1 \} \), i.e. the constraints in both states are part of \( \mathbb{I}_S \). The final states in both programs are
\[ [\sigma_1^f] = [\langle \text{mset}([A|L]); \top; \{L, A\} \rangle] \text{ and } \]
\[ [\sigma_2^f] = [\langle \text{mset}(L'); \text{append}(L, [A|L']); \{L, A\} \rangle]. \]

Both states are equivalent to the following states:
\[ [\sigma_1^f] = (\langle \text{mset}(X); \text{perm}(X, L'); \{X\} \rangle) \]
\[ \diamond \ [\langle \emptyset; X\models [A|L] \land \text{append}(L, [A|L']); \{X, L, A\} \rangle] \]
\[ \diamond_{\{X,L'\}} [\sigma_2^f], \]
\[ [\sigma_2^f] = (\langle \text{mset}(L'); \text{perm}(X, L'); \{X, L'\} \rangle) \]
\[ \diamond \ [\langle \emptyset; X\models [A|L] \land \text{append}(L, [A|L']); \{X, L, A\} \rangle] \]
\[ \diamond_{\{X,L'\}} [\sigma_2^f]. \]

Those two states are equivalent, i.e. \( [\sigma_1^f] \approx_S [\sigma_2^f] \), because \( L' \) is a permutation of \( X \). Hence, the two programs are operationally \( \mathcal{I}_S^{-}\mathbb{I}_S \)-equivalent modulo \( \approx_S \).

7.3 Discussion

The criterion for invariant-based operational equivalence modulo equivalence in this chapter is applicable for terminating and confluent programs with respect to the same invariant and equivalence relation. This allows us to only consider rule states in the criterion, instead of testing all states equivalent to rule states. Thereby, the complexity of the test is reduced and decidability is maintained.

The user-defined equivalence relation must be \( \diamond_{\mathcal{V}} \)-compatible to make use of monotonicity of CHR. Although this seems like a harsh restriction of the approach, the class of \( p \)-state equivalence relations shows that there are interesting examples of equivalence relations that are compatible to our approach. We are confident that there are other examples like equivalence relations that handle differing orders of arguments, renamed constraints, ignoring certain arguments of a constraint, etc. that can be covered by our approach.

Additionally, the invariant is required to be maintained by the equivalence relation. This restriction is only required for interface rule states and hence might be unproblematic in many cases. In fact, it might be even desirable in practice that the invariant is maintained by the equivalence relation for all states, since it seems to be counter-intuitive that some representatives of the same equivalence class are included.
in the state transition system, while their equivalent counterparts are excluded.

As for previous results for operational equivalence with respect to invariants, decidability of the approach depends on the invariant. The set of minimal extensions must be well-defined, i.e. respecting Definition 22 in a sense that there must be a minimal element $[\sigma_m] \in \mathcal{M}^I([\sigma])$ for every element $[\sigma'] \in \Sigma^I([\sigma])$, such that $[\sigma_m] \triangleleft [\sigma']$ and $[\sigma_m]$ in fact is a $\triangleleft$-minimal element of $\Sigma^I([\sigma])$. For some built-in constraints like $<$ for natural numbers, there is no $\triangleleft$-minimal element for some elements of the set of extensions as demonstrated in Example 16.

Furthermore, the set of minimal elements must be final in order that Theorem 11 is trivially decidable. In many cases, the set of minimal extensions is well-defined and empty, since the underlying invariant cannot be repaired by state extension. For instance, the invariant that there must be at most one $c$ constraint in a state cannot be fixed by extending the state.

Since we assume $I$-confluence modulo $\approx$ and therefore the test in Theorem 11 only considers the exact rule states, decidability does not depend on the equivalence relation as long as the equivalence of two arbitrary states is decidable. In contrast to the $\beta$-property in the confluence modulo equivalence criterion of Theorem 10, the set of rule states is always final, whereas in the $\beta$-property a possibly infinite amount of equivalent states has to be regarded. Recall that for $p$-state equivalence relations, this set can be reduced to a finite set due to monotonicity.

In practice, one has to be careful which constraints he chooses to be part of the interface $I$ as this by definition changes the semantics of $I$-equivalence. First of all, only constraints that are valid inputs and outputs should be part of the interface, since constraints that are only intermediate results that are removed from the output in a final state may inhibit $I$-$I$-equivalence with no need. In some cases, renaming of constraints might be necessary to ensure that no constraint symbols except for those used in input and output states are shared [Rai10, p. 97].

Nevertheless, it is important to note that all states equivalent to input and output states have to be considered when defining the interface of two programs. Although it is required neither by Definition 50 nor by Theorem 11, representatives of the same user-defined equivalence class should either all be in the class of $I$-states or none should be. States that are not $I$-states but equivalent to an $I$-state can be considered inelegant, as the intuition behind an equivalence relation of states is broken if two equivalent states lead to non-equivalent results in two programs that are operationally equivalent modulo that equivalence relation. Hence, with an inapt interface $I$-$I$-equivalence modulo $\approx$ only ensures equivalent results in both programs $I$-representatives.

Decidability depends on the invariant.

The set of rule states is finite and therefore the criterion for operational equivalence modulo equivalence is decidable for all compatible equivalence relations.

The interface has to be chosen wisely.

The interface should be compatible to the equivalence relation to avoid counter-intuitive effects.
Although the formal purpose of the definition of an interface can also be served by an appropriate user-defined equivalence relation, it is still important that the criterion allows to define both an interface and a user-defined equivalence relation. The reason is that the union of equivalence relations might introduce undesired effects. Hence, it might be that the interface equivalence relation is not compatible to the user-defined equivalence relation to a satisfactory degree.

7.4 RELATED WORK

To the best of our knowledge, our approach is the first that investigates operational equivalence modulo user-defined equivalence relations. It is based on the work of [AF99; Rai10], but has been extended by user-defined equivalence relations.

As for the confluence modulo equivalence theorem in Chapter 6, the work about invariant-based confluence for CHR from [Rai10; DSS07; DSS06] serves as a foundation for the invariant-based part of our work on operational modulo equivalence.

Certainly, our work on user-defined equivalence relations and confluence modulo equivalence is fundamental to the work on operational equivalence modulo equivalence in this chapter. Hence, the related work discussed in Sections 5.5 and 6.7 is also related to this chapter.

The first criterion for operational equivalence without invariants in CHR has been introduced in [AF99]. In [Rai10], the method has been extended by invariants and interfaces. Additionally, the theorem has been generalized such that it does not require the preconditions from the previous approach, namely confluence, termination and the condition that the set of minimal extensions of the rule states has to be finite. However, decidability is lost in those cases. For programs satisfying the preconditions, the approach is decidable.

In contrast to the more general criterion in [Rai10] (that does not consider user-defined equivalence relations), our criterion requires the two programs to be confluent and terminating and the set of minimal extensions to be finite for all rule states. We are confident that our criterion can be generalized analogously to [Rai10].
This part discusses program analysis of ACT-R. The first step is to define its formal operational semantics, overcoming the so far informal descriptions of its functionality. To use the program analysis methods of CHR extended in Part ii, an embedding of ACT-R in CHR is developed that is proven to be sound and complete with respect to the formal operational semantics of ACT-R (in particular the so-called abstract semantics). Based on this embedding, a decidable sufficient and necessary confluence criterion is presented that uses invariant-based confluence modulo equivalence. This work can be directly used to apply the operational equivalence result from Chapter 7.
As explained before, computational cognitive modeling tries to explore human cognition by building detailed computational models of cognitive processes [Sun08]. Cognitive architectures support the modeling process by providing a formal, well-investigated theory of cognition that allows for building cognitive models of specific tasks and cognitive features.

Currently, computational cognitive modeling architectures as well as the implementations of cognitive models are typically ad-hoc constructs. They lack a formalization from the computer science point of view. For instance, Adaptive Control of Thought – Rational (ACT-R) [And+04] is a widely employed cognitive architecture. It is a modular production rule system with a special architecture of the working memory that operates on data stored as so-called chunks, i.e. the unit of knowledge in the human brain. It has a well-defined psychological theory, however, its computational system is not described formally leading to implementations that are full of technical artifacts as claimed in [AW14b; SW07], for instance. Due to the many technical artifacts it is hard to merge different cognitive models in a greater context.

Formal analysis of cognitive models can support the modeling process to ensure model quality and to prove certain properties of cognitive models to validate their plausibility. However, the lack of formalization, the several implementations and the technical artifacts impede formal reasoning about the underlying languages and the programmed models. It makes it hard to compare different implementation variants of the languages. Furthermore, it complicates verifying properties of the models. These issues call for a formal semantics of cognitive modeling languages together with proper analysis techniques. This semantics should be an elegant formulation of the core features all those implementations of cognitive architectures use to explain human cognition.

In this chapter, an operational semantics of ACT-R is defined in two degrees of abstraction: In Section 8.1, the very abstract semantics is defined. It is a general description of ACT-R’s modular architecture as a production rule system setting up the frame in which different ACT-R implementations can be defined. Section 8.2 defines the abstract semantics as an instance of the very abstract semantics. For instance, it defines the matching and the general effects of production rules. Nevertheless, it still leaves room for specifying different variants of the semantics, e.g. by leaving the conflict resolution open. This means
that in the abstract semantics every rule of the conflict set is applicable
non-deterministically making it suitable for program analyses such as
confluence analysis. Sections 8.3 and 8.4 discuss the results and
related work, respectively.

8.1 Very Abstract Operational Semantics

The goal of the very abstract semantics is to capture as many ACT-R
implementations as possible leaving room for extensions and modi-
fications. Hence, it is a common theoretical foundation for different
ACT-R semantics and implementations. The very abstract semantics
describes the fundamental concepts of a production rule system that
operates on buffers and chunks like ACT-R. This work extends and
improves the definition from [AW14b]. Our work is compared to the
work from [AW14b] in Section 8.4.1. Later on, in Section 8.2, an in-
stance of this very abstract semantics is defined that is suitable for
analysis of cognitive models.

The following notions are used in the rest of this chapter: An ACT-R
architecture is a concrete instantiation of the very abstract semantics
and defines general parts of the system that are left open by the very
abstract semantics like the set of possible actions \( A \), the effect of such
an action or the selection process. In contrast to that, an ACT-R model
defines model-specific instantiations of parts like the set of types \( T \)
and the set of production rules \( R \).

8.1.1 Chunk Stores

As described before in Chapter 4, ACT-R operates on a network of
typed chunks that we call a chunk store. Therefore, we first define the
notion of types:

**Definition 52 (Chunk Types).** Let \( 2^C_A \) be the power set of all constant
symbols. A typing function \( \tau : T \rightarrow 2^C_A \) maps each type from the set of
types \( T \subseteq C_A \) to a finite set of allowed slot names. Every type set \( T \) must
contain a special type chunk \( \in T \) with \( \tau(\text{chunk}) = \emptyset \).

A chunk store is defined over a set of types and a typing function.
We abstract from chunk names as they do not add any information to
the system. In fact, chunks are defined as unique, immutable entities
with a type and connections to other chunks:

**Definition 53 (Chunk Store).** A chunk store \( \Delta \) is a set of elements of
the form \( c::(t, \text{val}) \) where \( c \in C_A \) is a unique chunk identifier, \( t \in T \) is a
chunk type and \( \text{val} : \tau(t) \rightarrow \Delta \) is a function that maps each slot of the chunk
(determined by the type \( t \)) to another chunk.

Every chunk store \( \Delta \) must contain a chunk \( \text{nil}::(\text{chunk}, \emptyset) \). Each chunk
store \( \Delta \) has an identifier function \( \text{id}_\Delta : \Delta \rightarrow C_A, c::(t, \text{val}) \mapsto c \) that
returns the chunk identifier of the chunk. The inverse of id is defined as follows:

\[
\text{id}_\Delta^{-1}(x) := \begin{cases} 
  c & \text{if } \text{id}_\Delta(c) = x, \\
  \text{nil} & \text{otherwise.}
\end{cases}
\]

The set of all chunk stores over all constant symbols \( C_\Lambda \) is denoted by \( \mathcal{D} \). Chunk identifiers can be omitted, if they do not play a role or are clear from the context, i.e. a chunk \( c :: (t, \text{val}) \) can be referred to as \( (t, \text{val}) \).

The typing function \( \tau \) maps a type \( t \) from the set of type names \( T \) to a set of allowed slots, hence the function \( \text{val} \) of chunk \( c \) has the slots of \( c \) as domain.

Chunk identifiers are unique within one chunk store. Note that two chunks are only considered equivalent, if they have the same chunk identifier, type and value functions (in that case they are the indistinguishable in the set and therefore treated as one and the same chunk). Hence, a chunk store can contain multiple elements with the same values that still are unique entities representing different concepts. This can be seen in the following example. We model our well-known example from Fig. 4.1 as a chunk store.

**Example 39 (Chunk Store of Natural Numbers).** The chunk store from Fig. 4.1 can be modeled as follows. Note that in the examples, we use chunk identifiers to define the connections in the slots (the \( \text{val} \) functions) instead of the whole chunk definition for the sake of brevity.

- The set of types is \( T_{39} = \{ \text{number}, \text{succ} \} \).
- The typing function \( \tau_{39} : T_{39} \to 2^C_\Lambda \) is defined as \( \tau_{39}(\text{number}) = \varnothing \) and \( \tau_{39}(\text{succ}) = \{ \text{number}, \text{successor} \} \).

Note that chunks of type \text{number} have no slots. A number chunk represents the mental concept of the number. To distinguish those chunks on the model level and give them a semantic meaning, their identifiers are number symbols. Hence, the chunk with identifier 1 will represent the mental concept of number 1. However, for the model itself, the identifier does not play a semantic role. The semantics comes only from the connections to other number chunks.

- We have the following chunks in our store \( \Delta_{39} \):
  - the unique entities with identifiers 1, 2, 3 that are defined as \( (\text{number}, \varnothing) \),
  - \( b :: (\text{succ}, \text{val}_b) \) with \( \text{val}_b(s) = \begin{cases} 
  1 & \text{if } s = \text{number} \\
  2 & \text{if } s = \text{successor}
\end{cases} \)
  - \( c :: (\text{succ}, \text{val}_c) \) with \( \text{val}_c(s) = \begin{cases} 
  2 & \text{if } s = \text{number} \\
  3 & \text{if } s = \text{successor}
\end{cases} \)
From the definition of a chunk store $\Delta$, we can derive a graph $(\Delta, E)$ where for each slot-value pair $\text{val}(s) = d$ of a chunk $c::(t, \text{val}) \in \Delta$ there is an edge $(c, d) \in E$ with label $s$. In the graphical representation, we can label vertices with the chunk identifiers. We can apply this to Example 39 to derive the graph illustrated in Fig. 4.1 on page 55.

Sometimes we want to build chunk stores that only have a few chunks in them that refer to chunks in other chunk stores in their slots. We call this concept a partial chunk store.

**Definition 54 (Partial Chunk Store).** A partial chunk store with reference to a chunk store $\Delta$, denoted as $\Delta^C$, is a set of elements of the form $c::(t, \text{val})$ where $c \in C_A$ is a unique chunk identifier, $t \in T$ is a chunk type and $\text{val} : \tau(t) \to \Delta \cup \Delta^C$ is a function that maps each slot of the chunk (determined by the type $t$) to another chunk from chunk store $\Delta \cup \Delta^C$. Every chunk in the partial chunk store $\Delta^C$ has a unique identifier that is disjoint from the identifiers in $\Delta$. The function $\text{id}_{\Delta^C} : \Delta^C \to C_A$ returns the chunk identifier for each chunk in $\Delta^C$.

The set of all partial chunk stores that refer to a chunk store $\Delta$ is denoted as $D^C_{\Delta}$.

**Example 40 (Partial Chunk Store).** Let $\Delta_{39}$ be the chunk store from Example 39. We define $\Delta_{40}$ as a partial chunk store that refers to $\Delta_{39}$. It contains the chunk $x::(\text{succ}, \text{val}_x)$ with the following slots:

$$
\text{val}_x := \begin{cases} 
2 & \text{if } s = \text{number} \\
3 & \text{if } s = \text{successor}
\end{cases}
$$

We define an operation that merges two (partial) chunk stores. In an abstract way it can be considered as a special set union that merges two elements of a chunk store, if they have the same chunk identifiers. However, since there are many different implementations of ACT-R, we do not want to limit our formulation to this special type of set union, but define a more general operator $\circ$. For the general understanding of this thesis and its proofs it is sufficient to think of it as set union that maintains uniqueness of chunk identifiers and might merge some chunks to one.

**Definition 55 (Chunk Merging).** Let $\mathcal{D}$ be the set of all chunk stores, $\Delta \in \mathcal{D}$ a chunk store and $D^C_{\Delta}$ the set of all partial chunk stores that refer to $\Delta$. Then $\circ : (\mathcal{D} \cup D^C_{\Delta}) \times (\mathcal{D} \cup D^C_{\Delta}) \to (\mathcal{D} \cup D^C_{\Delta})$ is called the chunk merging operator that merges two (partial) chunk stores to one (partial) chunk store.

We require the following properties for $\circ$. For all chunk stores $\Delta \in \mathcal{D}$ and all (partial) chunk stores $\Delta', \Delta'', \Delta''' \in \mathcal{D} \cup D^C_{\Delta}$:

1. The chunk merge operator is only defined for two (partial) chunk stores that have disjoint chunk identifiers, except for syntactically identical chunks. This means that for all $c' \in \Delta'$ and $c'' \in \Delta''$ with
id_A(c') = id_A(c''), it must hold that c' = c'', i.e. both chunks have same types and value functions.

2. The merging is closed, i.e. if Δ' and Δ'' are (partial) chunk stores, then Δ' \circ Δ'' is a (partial) chunk store.

3. (Δ' \circ Δ'') \circ Δ''' = Δ' \circ (Δ'' \circ Δ'''), i.e. \circ is associative.

4. \emptyset is the neutral element, i.e. Δ' \circ \emptyset = \emptyset \circ Δ' = Δ'.

5. If c' ∈ Δ', then c' ∈ Δ' \circ Δ'' and id_{Δ'\circ Δ''}(c') = id_{Δ'}(c'), i.e. Δ' ⊆ Δ' \circ Δ'''. This means that all elements of the left chunk store Δ' are also part of the merged chunk store.

6. There is a mapping map_{Δ',Δ''} : Δ' \cup Δ'' → Δ' \circ Δ''' that maps chunks from the original chunk stores to the merged chunk store. For all chunks c := i::(t, val) ∈ Δ' \cup Δ'': If there is a chunk c' ∈ Δ \circ Δ' in the merged chunk store with the same identifier id_{Δ'\circ Δ''}(c') = i, then c = c'. This means that the mapping function represents the actual merging of the chunk store and just returns the new identifiers for the chunks that have been removed due to the merging. All other chunks remain untouched.

Note that this also means that map_{Δ',Δ''}(c) = c if c ∈ Δ', i.e. the chunks from Δ' remain untouched by the merging (c.f. Definition 55.5).

From the axioms it is clear that (D ∪ D^C, \circ) is a monoid, since the structure is closed under \circ, associative and has a neutral element.

The simplest definition of a chunk merging operator is the set union ∪. For chunk stores that obey Definition 55.1 (i.e. that have disjoint identifiers except for syntactic equivalent chunks), set union is closed, associative and has neutral element \emptyset. It also obeys Definition 55.5. The mapping function is defined as the identity function as no elements are lost in the merging process.

Example 4.1 (Set Union As \circ). Let Δ := \{c_1, c_2\} and Δ := \{c_2, c_3\} with id_A(c_i) = i for i = 1, 2 and id_A(c_i) = i for i = 2, 3. Then Δ \circ Δ = \{c_1, c_2, c_3\}. Due to the same identifiers, types and value functions of the two appearances of c_2, the merged store only keeps one version of c_2. The mapping just returns the original chunks from both stores.

In most implementations, chunks that have the same structure are merged to one chunk regardless of identifier, i.e. for a chunk store Δ and two partial chunk stores Δ^C_1 and Δ^C_2 that refer to Δ: If c_1::(t, val) ∈ Δ^C_1 and c_2::(t, val) ∈ Δ^C_2, then c_1 and c_2 is merged to c_1 in Δ^C_1 \circ Δ^C_2. This means that only c_1 is kept in the merged store. The mapping function returns

\text{map}_{Δ^C_1,Δ^C_2}(c_1) = c_1 \quad \text{and} \quad \text{map}_{Δ^C_1,Δ^C_2}(c_2) = c_1.
8.1.2 States

ACT-R states are tuples that consist of several parts:

- a chunk store,
- a cognitive state that defines which chunks from the chunk store are currently in which buffer. Those chunks from the chunk store are the only ones that are visible to the production system, since it can only match chunks in buffers.
- A conjunction of additional information that can be used to store information representing the sub-symbolic level, and
- a time component.

In the following the missing parts of an ACT-R state are defined formally.

**Definition 56 (Cognitive State).** A cognitive state $\gamma$ is a function $B \rightarrow \Delta \times \mathbb{R}_0^+$ that maps each buffer to a chunk and a delay. The delay decides at which point in time the chunk in the buffer is available to the production system. A delay $d > 0$ indicates that the chunk is not yet available to the production system. This implements delays of the processing of requests. The set of all cognitive states is denoted as $\Gamma$, whereas $\Gamma_{\text{part}}$ denotes the set of partial cognitive states, i.e. cognitive states that are partial functions and do not necessarily map each buffer to a chunk.

ACT-R adds a sub-symbolic level to the symbolic concepts that have been defined so far. The sub-symbolic level adds implicit information to the symbolic structures that account for neural processes usually associated with neural network models of cognition [TLA06]. This distinguishes ACT-R from purely symbolic architectures like SOAR [TLA06].

To gather information from the sub-symbolic layer, we add a component to the state that contains (sub-symbolic) additional information needed to calculate sub-symbolic values. This information can be altered by an abstract function as can be seen in Section 8.1.3. The information will be expressed as conjunctions of predicates from first-order logic. The additional information is also used to manage data used in ACT-R’s modules.

Additionally, modules other than the procedural module hold their data in the additional information.

ACT-R states are defined as follows:

**Definition 57 (Very Abstract State).** A very abstract state is a tuple $\langle \Delta; \gamma; v; t \rangle$ where $\gamma$ is a cognitive state in the sense of Definition 56, $v$ is a conjunction of ground, atomic first order predicates (called additional information), $t \in \mathbb{R}_0^+$ is a time. The state space is denoted with $\Sigma_{\text{Av}}$. 
The chunk store $\Delta$ contains a type that is denoted by a constant and a valuation function that connects slot names (constants) to other elements from $\Delta$. The cognitive state $\gamma$ connects buffer names (constants) with chunks from $\Delta$ and a delay in $R^+_0$.

The additional information component $\upsilon$ is a conjunction of ground, atomic predicates over a signature $\Upsilon$ and the time is also a number. The set of allowed predicates for additional information is denoted as $\Upsilon$ and must contain at least $\top$. Additionally, the allowed additional information $\Upsilon$ is closed under conjunction. Additional information holds data of ACT-R’s modules as well as sub-symbolic information. It can be considered as a bag of structured information that can be used to store values needed for the computation process as in a database. The notation of a conjunction is of only syntactical nature at first, as we do not define a logical reading for the expressions. The semantics of additional information comes from its use in the operational semantics for instance in the rule selection function or the interpretation of actions as it will be defined in Section 8.1.3. However, there is a close relation to additional information and built-in constraints in the translation of ACT-R to CHR as shown in Chapter 9 that is made explicit by this syntactic choice. Nevertheless, it is always possible to think of additional information as a (multi-)set of relational data represented as first-order predicates.

Note that in an initial state, the chunk store contains at least the chunks that appear in the rules. Additionally, a very abstract state cannot contain variables from $V_A$, but is only compound from terms, sets and functions over constants from $C_A$.

We continue our running example by defining a very abstract state with one of the chunks defined in Example 39.

**Example 42 (ACT-R States).** We want to model the counting process of a little child that has learned the sequence of the natural numbers from one to ten as declarative facts and can retrieve those facts from declarative memory. Therefore, we add a chunk of type $g$ with a current slot that memorizes the current number in the counting process.

The following state has a chunk of type $g$ in the goal buffer that has the current number 1. The retrieval buffer is currently retrieving the chunk $b$ with number 1 and successor 2. The retrieval is finished in one second as denoted by the delay. Fig. 8.1 illustrates the state. The formal definition is:

- In addition to Example 39, we add a chunk type $g$ with a current slot that takes track of the current number in the counting process.

  - $T_{42} = T_{39} \cup \{g\}$ where $T_{39}$ is the set of types from Example 39.
  
  - $\tau_{42}(t) = \begin{cases} 
  \{\text{current}\} & \text{if } t = g \\
  \tau_{39}(t) & \text{otherwise.}
  \end{cases}$
• We define an initial goal chunk \( c_g \) of this new type \( g \) as \( c_g := (g, \text{val}_{\text{goal}}) \) where 
  \[
  \text{val}_{\text{goal}}(\text{current}) = 1.
  \]

• This chunk and the number chunks from Example 39 are part of a chunk store 
  \[ \Delta_{42} := \Delta_{39} \uplus \{ c_g \} \].

• The initial state in this example is defined as \( \sigma_0 := (\Delta_{42}; \gamma_0; \top; 1) \). It is a very abstract ACT-R state with the chunk store \( \Delta_{42} \), a cognitive state \( \gamma_0 \), empty additional information and with current time \( o \).

• The cognitive state \( \gamma_0 \) of the very abstract state \( \sigma_0 \) is defined as follows:
  
  – \( \gamma_0(\text{goal}) = (c_g, 0) \), i.e. the goal buffer holds the initial goal chunk \( c_g \). It has the delay \( o \), i.e. it is directly available to the procedural system.
  
  – \( \gamma_0(\text{retrieval}) = (b, 1) \) where \( b \) is defined as in Example 39. The delay 1 indicates that this chunk is not yet available to the procedural system. This represents the situation where the chunk \( b \) has been requested from the declarative module and the request is still processed.

The current time in the state is \( 1 \). The delay of the goal buffer is \( o \), i.e. its chunk is visible to the procedural system. The delay of the retrieval buffer is \( 1 \), i.e. the chunk will be visible when the global time of the state is \( 2 \).

![Figure 8.1: Visual representation of the very abstract state defined in Example 42. The dashed arrow signifies that the chunk in the retrieval buffer is not yet visible (as indicated by the delay right of the buffer's name) [GF15b, p. 118; GF18a, 22:11].](image)

In Example 42, the additional information component was empty. This is not completely accurate, since the declarative module has its own chunk store – called declarative memory – that is a subset of the global chunk store. To resolve requests, only the chunks in this separate chunk store are considered. In the following example, additional information for the declarative memory is defined.
Example 43 (Additional Information). Chunks in the declarative memory can be represented by predicates of the form \( \text{dmchunk}(c, t) \) where \( c \) is a chunk identifier and \( t \) a type. The slot connections can be represented by predicates of the form \( \text{dmslot}(c, s, c') \) that models the connection of slot \( s \) in chunk \( c \) to chunk \( c' \).

The declarative memory consisting of the chunks \( b, 1 \) and \( 2 \) can be represented as follows:

\[
\text{dmchunk}(1, \text{number}) \land \text{dmchunk}(2, \text{number}) \land \\
\text{dmchunk}(b, \text{successor}) \land \text{dmslot}(b, \text{number}, 1) \land \text{dmslot}(b, \text{successor}, 2).
\]

In practice, the chunks in the declarative memory are complemented with some sub-symbolic information, namely activation levels that influence the latency of a retrieval request and that can model learning and forgetting.

8.1.3 Operational Semantics

In this section, the state transition system of the very abstract semantics is defined. First of all, the set of applicable rules has to be defined. Therefore, a selection function \( S \) is introduced that is defined by the architecture and maps a state to a set of applicable rules and the variable bindings implied by the application of the rule.

Definition 58 (Selection Function). Let \( \Theta(V_A, C_A) \) be the set of possible ground substitutions over variables \( V_A \) and constants \( C_A \) that replace every variable from \( V_A \) with at most one value from \( C_A \). The set \( 2^{\mathcal{R} \times \Theta(V_A, C_A)} \) denotes the power set of all tuples \((r, \theta)\) where \( r \in \mathcal{R} \) is a rule and \( \theta \in \Theta(V_A, C_A) \) is a ground substitution of variables from \( V_A \) with constants from \( C_A \) as described before.

A selection function is a function \( S : \Sigma_{\text{Ava}} \rightarrow 2^{\mathcal{R} \times \Theta(V_A, C_A)} \) that maps a state to a set of pairs \((r, \theta)\) where \( r \in \mathcal{R} \) is a production rule and \( \theta \in \Theta(V_A, C_A) \) is a substitution of variables from \( V_A \) with constants from \( C_A \), such that all variables from the rule \( r \) are substituted, i.e. \( \text{dom}(\theta) = \text{vars}(r) \).

The function \( S \) usually defines a notion of matching and makes sure that only rules that match visible information in the buffers can fire (i.e. only chunks that are not delayed are considered). In implementations the set of applicable rules typically is restricted to none or one element. The reason is that the set of syntactically matching rules is filtered using sub-symbolic information to get rid of non-determinism. Abstract definitions of ACT-R can allow more than one rule to be applicable.

To define the modification of a state by a transition, we define interpretation functions of actions that determine the possible effects of an action.

Definition 59 (Interpretation of Actions). Let \( \Sigma_{\text{Ava}} \) be the set of all very abstract states and \( 2^{\mathcal{D}_\Delta} \) the power set of all partial chunk stores that refer to \( \Delta \).
An interpretation of an action is a function \( I : A \times \Sigma_{\text{ava}} \rightarrow 2^{D_{\Delta} \times \Gamma_{\text{part}} \times Y} \) that maps a tuple of an action and a very abstract state to a set of tuples. Each tuple consists of a partial chunk store, a partial cognitive state and additional information. The following conditions must hold: \( (\Delta^*, \gamma^*, \upsilon^*) \in I(a, \sigma) \) if

1. \( I(a, \sigma) \neq \emptyset \), i.e. the interpretation of an action has at least one effect,
2. \( I(a, \langle \Delta, \gamma, \top \land \upsilon, t \rangle) = I(a, \langle \Delta, \gamma, \upsilon, t \rangle) \), i.e. \( \top \) is the neutral element of additional information and has no effect on the processing of actions,
3. the resulting chunk store \( \Delta^* \) is a partial chunk store that refers to \( \Delta \) and whose chunk identifiers are disjoint from \( \Delta \),
4. the co-domain of \( \gamma^* \) is \( \Delta^* \times \mathbb{R}_0^+ \), i.e. the cognitive state can only refer to chunks in the resulting chunk store \( \Delta^* \), and
5. if the action \( a \) has the buffer \( b \) in its scope, i.e. \( a := (b, t, p) \), then the resulting partial cognitive state \( \gamma^* \) has only \( b \) in its domain, i.e. \( \text{dom}(\gamma^*) = \{b\} \).

An interpretation maps each state and action of the form \( a(b, t, P) \) – where \( a \in A \) is an action symbol, \( b \in C_A \) a constant denoting a buffer, \( t \in C_A \) a type, and \( P \subseteq C_A \times (C_A \cup V_A) \) is a set of slot-value pairs – to a tuple \( (\Delta^*, \gamma^*, \upsilon^*) \). Thereby, \( \Delta^* \) is a partial chunk store that refers to \( \Delta \), \( \gamma^* \) is a partial cognitive state, i.e. a partial function that assigns only the buffer \( b \) from the action to a chunk and a delay. The partial cognitive state \( \gamma^* \) will be taken in the operational semantics to overwrite the changed buffer contents, i.e. it contains the new contents of the changed buffers. Analogously, the additional information \( \upsilon^* \) defines the additions to the sub-symbolic level induced by the action.

Note that the interpretation of an action can return more than one possible effect that can be chosen non-deterministically. This is used in the abstract semantics where due to the lack of sub-symbolic information all possible effects have to be considered. For example, the declarative module can find more than one chunk matching the retrieval request. Usually, by comparing activation levels of chunks, one chunk will be returned. However, in the abstract semantics all matching chunks are possible. In the refined semantics of our prior work in \[GF15b\] that formalizes the behavior of actual ACT-R implementations, the selection is restricted to one possible effect as proposed by the ACT-R reference manual \[Bot\]. This is achieved by defining an interpretation function that returns a set of at most one effect.

**Example 44 (Interpretation of an Action).** In this example we define a neutral effect. We will see later that if this effect is applied to a state, the state does not change modulo time.

Let \( a := a(b, t, p) \) be our action that produces the neutral effect and \( \sigma := \langle \Delta; \gamma; \upsilon; t \rangle \) an ACT-R state. We define

\[
I(a, \sigma) := \{ (\{c\}, \gamma', \top) \}
\]
then the combination partial chunk stores that refer to the same chunk store \( \Delta \) where information from both effects. The merged store.

However, the second chunk store might have lost some members. The mapping function assigns every chunk from the second store one from \( \Delta \). Note that from the definition of \( \gamma \) of the effects \( e \) and \( f \) with respect to a chunk store \( \Delta \) and \( \gamma \) are partial chunk stores with disjoint chunk identifiers that refer to \( \Delta \) and \( \gamma' : B' \rightarrow \Delta' \circ \Delta' \times R^+_\Delta \), \( \gamma'' : B'' \rightarrow \Delta'' \circ \Delta'' \times R^+_\Delta \) are partial cognitive states with disjoint domains, i.e. \( B', B'' \subseteq B \) and \( B' \cap B'' = \emptyset \) and \( v \) is a conjunction of first-order predicates. Then the combination of the effects \( e \) and \( f \) with respect to a chunk store \( \Delta \) is defined as

\[
e \sqcup f := (\Delta' \circ \Delta''', \gamma, v' \land v'')
\]

where \( \gamma : B' \cup B'' \rightarrow (\Delta' \circ \Delta''') \times R^+_\Delta \) with

\[
\gamma(b) := \begin{cases} 
    \text{map}_{\Delta', \Delta'''}(\gamma'(b)) & \text{if } b \in \text{dom}(\gamma'), \\
    \text{map}_{\Delta', \Delta'''}(\gamma''(b)) & \text{if } b \in \text{dom}(\gamma'').
\end{cases}
\]

The intuition behind this definition is that two effects of an action, i.e. two triples of chunk store, cognitive state and additional information, are merged to one effect that combines them. Hence, we get a merged partial cognitive state that has the combined buffer-chunk mappings of the two original cognitive states. This is possible, since the domains of the partial cognitive states are required to be disjoint.

The partial chunk stores are merged to one partial chunk store referring to the same total chunk store. Note that from the definition of \( \circ \), the chunk store of the first effect is a subset of the merged store. However, the second chunk store might have lost some members. The mapping function assigns every chunk from the second store one from the merged store.

The merged additional information is a conjunction of the additional information from both effects.

The combination is well-defined: \( \Delta' \circ \Delta'' \) exists since \( \Delta' \) and \( \Delta'' \) are partial chunk stores that refer to the same chunk store \( \Delta \). Additionally, the combination of a rule, we first define how two interpretations can be combined. Therefore, we introduce the following set operator, that combines two sets of sets:

**Definition 60** (Combination Operator \( \sqcup \) for Effects). Let \( e, f \) be effects of two actions, i.e. results of an interpretation function of an action in a state \( \sigma \) with chunk store \( \Delta \), i.e. \( e \in I(a', \sigma) \) and \( f \in I(a'', \sigma) \). Let \( e := (\Delta', \gamma', v') \) and \( f := (\Delta'', \gamma'', v'') \) where \( \Delta', \Delta'' \) are partial chunk stores with disjoint chunk identifiers that refer to \( \Delta \) and \( \gamma' : B' \rightarrow \Delta' \circ \Delta' \times R^+_\Delta \), \( \gamma' : B'' \rightarrow \Delta'' \circ \Delta'' \times R^+_\Delta \) are partial cognitive states with disjoint domains, i.e. \( B', B'' \subseteq B \) and \( B' \cap B'' = \emptyset \) and \( v \) is a conjunction of first-order predicates. Then the combination of the effects \( e \) and \( f \) with respect to a chunk store \( \Delta \) is defined as

\[
e \sqcup f := (\Delta' \circ \Delta'', \gamma, v' \land v'')
\]

where \( \gamma : B' \cup B'' \rightarrow (\Delta' \circ \Delta'') \times R^+_\Delta \) with

\[
\gamma(b) := \begin{cases} 
    \text{map}_{\Delta', \Delta''}(\gamma'(b)) & \text{if } b \in \text{dom}(\gamma'), \\
    \text{map}_{\Delta', \Delta''}(\gamma''(b)) & \text{if } b \in \text{dom}(\gamma'').
\end{cases}
\]

The actions of a rule have to be combined to compute the effects of a rule.
they have disjoint identifiers because they are results of an interpretation function (see Definition 59) and therefore their merging cannot fail due to different chunks with same identifiers.

The cognitive state is valid, since it has the combined domain of $\gamma'$ and $\gamma''$ and just maps the chunks from those original partial cognitive states to their versions in the merged chunk store by map, i.e. the co-domain of $\gamma$ is just the merge product of the co-domains of $\gamma'$ and $\gamma''$ and keeps their connections of buffers to chunks.

The definition of combinations of effects can be lifted to sets of effects by the following definition. Let $E$ and $F$ be two sets of effects in some state $\sigma$ with chunk store $\Delta$, then their combination is defined as all possible pairwise combinations of their elements:

$$E \sqcup F := \{e \sqcup f \mid e \in E \land f \in F\}.$$  

Since the interpretation of an action is possibly non-deterministic, i.e. might have more than one effect triple, the combination of such sets of effect triples is a set that combines each effect from the first set with each effect from the second set. This leads to a set of combined effects from which the transition system will be able to choose one non-deterministically. However, since every effect set is required to have at least one effect, the same applies for their combination.

The interpretation function $I : \mathcal{R} \times \Sigma_{Ava} \to 2^{\mathcal{D} \times \Gamma_{part} \times \Upsilon}$ that maps a rule to all its possible effects (i.e. chunk store, cognitive state and additional information) in a given state is defined as follows:

**Definition 61 (Interpretation of Rules).** In a state $\sigma := (\Delta; \gamma; v; t)$, a rule $r := L \Rightarrow R$ is interpreted by an interpretation function $I : \mathcal{R} \times \Sigma_{Ava} \to 2^{\mathcal{D} \times \Gamma_{part} \times \Upsilon}$. It maps a rule and a state to a set of tuples of partial chunk stores, partial cognitive states and additional information, i.e. a set of all possible effects of the rule in the given state. The interpretation function is defined as follows: $I(r, \sigma)$ applies the function apply, to all tuples in the result set when combining the individual actions of the rule:

- **apply** : $\mathcal{D}_\Delta \times \Gamma_{part} \times \Upsilon \to \mathcal{D}_\Delta \times \Gamma_{part} \times \Upsilon$ is a function that applies some more effects at the end of the rule application and is defined by the architecture. It maps a tuple of a partial chunk store, a partial cognitive state and a conjunction of additional information to another tuple of the same kind.

- For all actions $\alpha \in R$, the resulting chunk stores are disjoint (i.e. chunk identifiers are renamed apart).

- The interpretation has the result

$$I(r, \sigma) = apply_r\left(\bigcup_{\alpha \in R} I(\alpha, \sigma)\right)$$

where the combination operator $\sqcup$ refers to $\Delta$ and the apply function is applied to each member of the combination set. Hence, all possible effects
of the rules are combined and each of the resulting partial cognitive states is then modified by the apply function that is defined by the architecture.

The apply function can apply additional changes to the state that are not directly defined by its actions. For instance, it can change some sub-symbolic values that depend on the rule application like the utility of the rule itself. Note that by definition of the ACT-R syntax it is ensured that each of the \( \gamma_{\text{part}} \) in the combination of the individual actions is still a function, since only one action per buffer is allowed as defined in Section 4.2.

Note that there are two types of non-determinism in the interpretation of a rule:

1. The first non-determinism comes from the non-deterministic nature of interpretations of an action. Each action can lead to different results (depending on their definition). This is why all interpretation functions have power sets of effects as co-domain.

2. The second type of non-determinism comes from the definition of the combination operator that merges chunk stores by using the chunk merging operator \( \circ \). Since \( \circ \) is not required to be commutative, the result of the merged chunk stores may vary. This leads to possibly differing chunk identifiers. It is possible to abstract from this kind of non-determinism by introducing the concept of (graph) isomorphism on chunk stores.

The operational semantics is defined as the state transition system \((\Sigma_{\text{Av}a}, \rightarrow)\):

**Definition 62** (Very Abstract Operational Semantics). In the very abstract operational semantics of ACT-R, the transition relation \( \rightarrow \) of \( \Sigma_{\text{Av}a} \times \Sigma_{\text{Av}a} \) over very abstract states is defined as follows:

**Apply** For a rule \( r \) the following transitions are possible:

\[
(\sigma, (\Delta^*, \gamma^*, v^*)) \in I(\sigma) \rightarrow \langle \Delta; \gamma; v; t \rangle \xrightarrow{\text{apply}} \langle \Delta \circ \Delta^*; \gamma'; v \wedge v^*; t' \rangle
\]

where

- \( \gamma' : \mathcal{B} \rightarrow \Delta \circ \Delta^* \),

\[
\gamma'(b) := \begin{cases} 
(map_{\Delta, \Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\
(map_{\Delta, \Delta^*}(c), d \ominus \delta) & \text{otherwise, if } \gamma(b) = (c, d),
\end{cases}
\]

- \( x \ominus y := \begin{cases} 
 x - y & \text{if } x > y \text{ for two numbers } x, y \in \mathbb{R}_0^+, \text{ and} \\
 0 & \text{otherwise}
\end{cases} \)

- \( t' = t + \delta \) for a delay \( \delta \in \mathbb{R}_0^+ \) defined by the concrete instantiation of ACT-R.
When applying the rule, the resulting partial chunk store $\Delta^*$ is merged with the chunk store $\Delta$ from the state. Hence, $\Delta \circ \Delta^* = \Delta \circ \Delta_{\Delta^*} \circ \Delta \ldots \circ \Delta_{\Delta_{\Delta^*}}$ for all actions $a_i$ on the right-hand side of the rule. Note that $\Delta \subseteq \Delta \circ \Delta^*$, i.e. all chunks in $\Delta$ also appear in the merged chunk store with preservative chunk identifiers by Definition 55 of the chunk merging.

The partial cognitive state that comes from the interpretation of the rule replaces all positions in the original cognitive state where it is defined, otherwise the original cognitive state remains untouched. Note that for all buffers $b \in B$ and $b \notin \text{dom}(\gamma^*)$ with $\gamma(b) = (c, d)$ we can also write $\gamma'(b) := (c, d \ominus \delta)$ instead of $\gamma'(b) := (\text{map}_{\Delta \Lambda^*}(c), d \ominus \delta)$ since the chunk merging guarantees that $\Delta \subseteq \Delta \circ \Delta^*$ with preservative identifiers as mentioned before.

The delays are taken from the partial cognitive state $\gamma^*$ or are reduced by a constant amount that models progression of time.

When it is clear from the context, we just use $\rightarrow^r$ to denote that the transition applies rule $r$.

**NO-RULE**

$$C(\sigma)$$

$\sigma := \langle \Delta; \gamma; \upsilon; t \rangle \rightarrow_{\text{no}} \langle \Delta; \gamma'; \upsilon; \vartheta(\sigma) \rangle$

where

- $C \subseteq \Sigma_{\text{Ava}}$ is a side condition in form of a logical predicate,
- $\text{update} : \Sigma_{\text{Ava}} \rightarrow \Gamma_{\text{part}}$ a function that describes how the cognitive state should be transformed,
- $\vartheta : \Sigma_{\text{Ava}} \rightarrow \mathbb{R}^+$ a function that describes the time adjustment in dependency of the current state, and
- $\gamma'(b) := \begin{cases} \text{update}(\sigma)(b) & \text{if defined} \\ \gamma(b) & \text{otherwise} \end{cases}$

is the updated cognitive state.

An apply transition applies a rule that satisfies the conditions of the selection function $S$ by overwriting the cognitive state $\gamma$ with the result from the interpretations of the actions of rule $r$. Thereby, one possible combination of all effects of the actions is considered. Note that the transition is also possible for all other combinations. Only the buffers with a new chunk are overwritten, the others keep their contents. The same applies for parameters: They keep their value except for those where $\upsilon^*$ defines a new value. Additionally, the rule application can take a certain time $\delta$ that is defined by the architecture. Time is forwarded by $\delta$, i.e. the time in the state is incremented by $\delta$ and the delays in the cognitive state that determine when a chunk becomes visible to the system are decremented by $\delta$ (with a minimal delay of 0).
The no-rule transition defines what happens under certain conditions without involving a rule application. In typical instances of the semantics, the no-rule transition defines the behavior of the system, when there are still effects of e.g. requests that can be applied. This means that there are buffers \( b \in B \) with \( \gamma(b) = (c,d) \) and \( d > 0 \), i.e. information that is not visible to the production rule system.

Note that the no-rule transition is important, because sometimes the processing of requests may block the application of rules because there is no information in the buffers of the busy modules and hence, some rules may not be applicable and the system gets stuck. In this case, the system waits until the requests are completed in most semantics. This can be realized by setting the delay of a buffer to 0 in the cognitive state. Note that the formulation of \([AW\,14b]\) does not include any transitions that take care of this case.

Neutral effect

The no-rule transition is formulated abstractly. It can be instantiated by the concrete architecture.

Importance of the No-Rule-Transition

Note that our definition of the no-rule transition is a generalization of the above described typical application when no rule is applicable. The very abstract semantics allows for an arbitrary side condition \( C(\sigma) \), that can be defined by the particular ACT-R instantiation, to update cognitive state according to the function \( \text{update} \) and the current time of the system is set to a specified time \( \vartheta(\sigma) \) without involving a rule application. Both functions are also defined by the concrete architecture.

This makes new information visible to the production system and hence new rules might fire. In typical ACT-R implementations, the side condition \( C(\sigma) \) is that \( S(\sigma) = \emptyset \), i.e. that no rule is applicable, and \( \vartheta(\sigma) := t + d^* \) where \( \sigma \) has the time component \( t \) and \( d^* \) is the minimum delay in the cognitive state of \( \sigma \). This means that time is forwarded to the minimal delay in the cognitive state and makes for instance pending requests visible to the production rule system. It can be interpreted like if the production rule system waits with the next rule application until there is new information present that leads to a rule matching the state. This behavior coincides with the specification from the ACT-R reference implementation \([Bot]\). If no transition is applicable in a state \( \sigma \), i.e. there is no matching rule and no invisible information in \( \sigma \), then \( \sigma \) is a final state and the computation stops.

The definition of our very abstract semantics leaves parts to be defined by the actual architecture and the model. Table 8.1 summarizes what has to be defined by an architecture and a model.

In the following, it is shown that the neutral effect from Example 44 is a neutral element of the rule application except for the time component.

Example 45 (Neutral Element of Rule Applications). Let \( \sigma := (\Delta; \gamma; v; t) \) be an ACT-R state and \( r := L \Rightarrow \{a\} \) an ACT-R rule with only one action \( a := a(b^*, t^*, P^*) \). Let \( \gamma(b^*) := (c^*, d^*) \).

The neutral effect \( e = \{(c^*), \gamma^*, \top\} \) from Example 44, is defined as the only effect of \( a \), i.e. \( I(a, \sigma) := e \). By definition of the neutral effect,
Table 8.1: Parameters of the very abstract semantics that must be defined by the architecture or the cognitive model respectively.

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>( C_A )</td>
<td>( \mathcal{T} ) set of types</td>
</tr>
<tr>
<td>( V_A )</td>
<td>( \tau ) typing function</td>
</tr>
<tr>
<td>( B )</td>
<td>( \mathcal{R} ) set of rules</td>
</tr>
<tr>
<td>( A )</td>
<td>( \sigma_S ) start state</td>
</tr>
<tr>
<td>( \delta )</td>
<td>rule delay</td>
</tr>
<tr>
<td>( Y )</td>
<td>allowed additional information</td>
</tr>
<tr>
<td>( \circ )</td>
<td>chunk merging operator</td>
</tr>
<tr>
<td>( S )</td>
<td>rule selection function</td>
</tr>
<tr>
<td>( I )</td>
<td>interpretation functions</td>
</tr>
<tr>
<td>update</td>
<td>transformation of the cognitive state after no-rule transition</td>
</tr>
<tr>
<td>( \vartheta )</td>
<td>progress of time after no-rule transition</td>
</tr>
</tbody>
</table>

\( \gamma^*(b^*) := (c^*, d^*) \), i.e. the same chunk and delay as in the original cognitive state \( \gamma \) and undefined for all other inputs. Then

\[
I(r, \sigma) := \{(\{c^*\}, \gamma^*, \top)\}.
\]

Let \( \sigma \rightarrow^r \sigma' \). Then \( \sigma' := (\Delta \circ \{c^*\}; \gamma'; \nu \land \top; \tau') \). The follow-up cognitive state \( \gamma' \) is

\[
\gamma'(b) := \begin{cases} 
(map_{\Delta, \{c^*\}}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\
(map_{\Delta, \{c^*\}}(c), d \odot \delta) & \text{otherwise, if } \gamma(b) = (c, d). 
\end{cases}
\]

This can be reduced to

\[
\gamma'(b) := \begin{cases} 
(c^*, d) & \text{if } b = b^* \\
(c, d \odot \delta) & \text{otherwise, if } \gamma(b) = (c, d), 
\end{cases}
\]

Since \( c^* \) has the same type and value function, \( \gamma'(b) \) can be considered equivalent to \( \gamma(b) \) modulo delays for all \( b \in B \). The action only adds the predicate \( \top \) to the conjunction of additional information which is ignored when actions are processed, i.e. effectively no information is added. Hence, \( \sigma \) is equivalent to \( \sigma' \) modulo delays and the time component that comes from the pure rule application.

### 8.2 Abstract Operational Semantics

The abstract semantics is defined as an instance of the very abstract semantics. It is suitable for the analysis of procedural core of cognitive...
Matching is an important part of all ACT-R variants.

8.2 abstract operational semantics

models because it abstracts from timings and conflict resolution by leaving parts of the transition system to non-deterministic choices and still giving room for extensions and modifications. The idea is to define the minimal core of all implementations of ACT-R’s procedural system disregarding parameter choices, timings, sub-symbolic information and module configuration. The abstract semantics captures all possible state transitions the procedural system can make.

Since it is a central part of the procedural system of ACT-R, we first define the notion of matchings:

**Definition 63 (Matching).** A buffer test $\beta := (b, t, P)$ for a buffer $b \in B$ testing for a type $t \in T$ and slot-value pairs $P \subseteq C_A \times (C_A \cup \Gamma_A)$ matches a state $\sigma := \langle \Delta; \gamma; v; x \rangle$, written $\beta \sqsubseteq \sigma$, if and only if there is a substitution $\theta : \Gamma_A \rightarrow C_A$ such that $\gamma(b) = (i_b :: (t, val), 0)$ and for all $(s, v) \in P : id_A(val(s)) = v\theta$. We define $\text{Bindings}(\beta, \sigma) := \theta$ as the function that returns the smallest substitution that satisfies the matching $\beta \sqsubseteq \sigma$.

This definition can be extended to rules: A rule $r := L \Rightarrow R$ matches a state $\sigma$, written as $r \sqsubseteq \sigma$, if and only if for all buffer tests $t \in L$ match $\sigma$. The function $\text{Bindings}(r, \sigma)$ returns the smallest substitution that satisfies the matching $r \sqsubseteq \sigma$.

This means that a buffer test matches a state, if the tested buffer contains a chunk of the tested type and all slot tests hold in the state, i.e. the variables in the test can be substituted by values consistently such that they match the values from the state. The values in the test are denoted by the identifiers of the chunks. Note that a test can only match chunks in the cognitive state that are visible to the system, i.e. whose delay is zero. A test cannot match chunks with a delay greater than zero.

We give the architectural parameters that are left open in the very abstract semantics:

**States** We set the time in every state to $t := 0$ (or any other constant) because abstract states are not timed. Hence, each abstract state is a tuple $\langle \Delta; \gamma; v; 0 \rangle$ where $\gamma \in \Gamma$ is a cognitive state. We sometimes project an abstract state $\langle \Delta; \gamma; v; 0 \rangle$ to $\langle \Delta; \gamma; v \rangle$ for the sake of brevity.

**Selection function** The rule selection in the abstract semantics is simply defined as $S_{abs}(\sigma) := \{ (r, \text{Bindings}(r, \sigma)) \mid r \in R \land r \sqsubseteq \sigma \}$. Hence we select all matching rules in state $\sigma$ and replace the variables from the rules by their actual values from the matching, since in the state transition system the substitution $\theta$ is applied to the rule when calculating the effects.

**Effects** For a state $\sigma = \langle \Delta; \gamma; v; 0 \rangle$ with $\gamma(b) = (i_b :: (t, val_\gamma), d)$, the interpretation function $I_{abs}$ for actions $A := \{=, +\}$ in the abstract semantics is defined as follows:
\[-c := i_{c::(t, val_b)} \text{ for a fresh } i_c \in C_A,\]
\[-\Delta^* := \{c\},\]
\[-\gamma^*(b) := (c, 0), \text{ and}\]
\[-\text{the new slot values are:}\]
\[
val_b(s) := \begin{cases} 
\text{id}_\Delta^{-1}(v) & \text{if } (s, v) \in P \\
val_\gamma(s) & \text{otherwise.}
\end{cases}
\]

This means that a modification creates a new chunk that modifies only the slots specified by \(P\) and takes the remaining values from the chunk that has been in the buffer. Note that the type cannot be modified, since the resulting chunk always has the type derived from the chunk that has previously been in the buffer. Modifications are deterministic, i.e. that there is only one possible effect.

The slot-value function \(val_b\) is well-defined, since it appears in a partial chunk store that references \(\Delta\), it has \(\Delta\) as co-domain by Definition 54 of a partial chunk store.

If the action contains a slot-value pair \((s, v)\) that modifies \(s\) to a chunk that was not existent in the original state \(\sigma\), this chunk is not magically constructed, since we do not know its type or values. Instead, we map this slot to \(\text{nil}\). This comes from the definition of \(id_\Delta^{-1}\), which is \(\text{nil}\) for \(v\), since the chunk referenced by \(v\) does not exist in \(\Delta\).

\[-(\Delta^*, \gamma^*, \upsilon^*) \in I_{abs}(+(b, t, P), \sigma) \text{ for requests}\]
\[-\text{if}\]
\[-\text{request}_b : T \times 2^{C_A \times (C_A \cup V_A)} \times V \rightarrow 2^{\Delta \times R_+} \times \Upsilon \text{ is a function}\]
\[-\text{defined by the architecture for each buffer. It calculates the set of possible answers for a request that is specified by a type and a set of slot value pairs. Possible answers are tuples } (c, d, v) \text{ of a chunk } c, \text{ delay } d \text{ and parameter valuation function } v.\]
\[-\text{For all } (c^*, d^*, \upsilon^*) \in \text{request}_b(t, P, v) \text{ we set } \gamma^*(b) := \begin{cases} 
(c^*, 1) & \text{if } d^* > 0 \\
(c^*, 0) & \text{otherwise,}
\end{cases}\]
\[-\text{and } \Delta^* := \{c^*\}.\]
Note that the additional information in the result of the request is directly added to the result of the interpretation function to update the internal state of the requested module.

- The function \textit{apply} from Definition 61 that adds additional changes to the state when a rule is applied is defined as the identity function, i.e. no changes to the state are introduced by the rule application itself but only by its actions.

**Rule Application Delay** The delay of a rule application is set to \( \delta := 0 \), since the abstract semantics does not care about timings.

**No-Rule Transition** In the no-rule transition, there are three parameters to be defined by the actual ACT-R instantiation: The side condition \( C \), the state update function \( \text{update} \) and the time adjustment function \( \vartheta \). We define them for a state \( \sigma := (\gamma; v; t) \) as follows:

- \( \sigma \in C \) if and only if there is a \( b^* \in B \) such that \( \gamma(b^*) = (c, d) \) with \( d > 0 \), i.e. there is a buffer with a chunk that is not visible to the system. Those are the cases where there is a pending request. This means that the no-rule transition is possible as soon as there is at least one pending request. We call the buffer of one such request \( b^* \).

- \( \text{[update}(\sigma)\text{]}(b^*) := (c, 0) \) if \( \gamma(b^*) = (c, d) \) and \( d > 0 \) for one \( b^* \in B \). This means that one pending request is chosen to be applied (the one appearing in \( C \)). Since this is a rule scheme and \( b^* \) can be chosen arbitrarily, the transition is possible for all assignments of \( b^* \). This coincides with the original definition of our abstract semantics where one request is chosen from the set of pending requests.

- The function \( \vartheta \) that determines how the time is adjusted after a chunk has been made visible is defined as \( \vartheta(\sigma) := t \), i.e. the time is not adjusted.

Table 8.2 shows the parameters of the abstract semantics that have to be defined by the architecture.

We summarize the transition scheme of the abstract semantics:

**Rule Transition**

\[
\begin{align*}
\sigma & \sqsubseteq \sigma \wedge \theta = \text{Bindings}(r, \sigma) \wedge (\Delta^*, \gamma^*, \nu^*) \in I(r \sigma, \nu) \\
\sigma := (\Delta; \gamma; v) & \xrightarrow{\text{apply}} (\Delta \circ \Delta^*; \gamma'; v \wedge \nu^*)
\end{align*}
\]

where \( \gamma' : B \rightarrow \Delta \circ \Delta^* \),

\[
\gamma'(b) := \begin{cases} 
(map_{\Delta, \Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\
(map_{\Delta, \Delta^*}(c), d) & \text{otherwise, if } \gamma(b) = (c, d).
\end{cases}
\]
Table 8.2: Parameters of the abstract semantics that must be defined by the architecture.

<table>
<thead>
<tr>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \mathcal{C}_A )</td>
</tr>
<tr>
<td>( \mathcal{V}_A )</td>
</tr>
<tr>
<td>( B )</td>
</tr>
<tr>
<td>( A )</td>
</tr>
<tr>
<td>( \delta )</td>
</tr>
<tr>
<td>( Y )</td>
</tr>
<tr>
<td>( \circ )</td>
</tr>
<tr>
<td>( \text{request}_b )</td>
</tr>
</tbody>
</table>

Again, since \( \Delta \subseteq \Delta \circ \Delta^* \) with preservative chunk identifiers, we can also write

\[
\gamma'(b) := \begin{cases} 
(map_{\Delta \circ \Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\
(c, d) & \text{otherwise, if } \gamma(b) = (c, d).
\end{cases}
\]

**No-Rule Transition**

\[
\sigma := (\Delta; \gamma; v; t) \rightarrow_{\text{no}} (\Delta; \gamma'; v)
\]

where \( \gamma'(b) := \begin{cases} 
(c^*, 0) & \text{if } b = b^* \\
\gamma(b) & \text{otherwise}.
\end{cases} \)

The running example is extended by a derivation in the abstract semantics:

**Example 46 (Abstract Semantics).** We begin with the state

\[ \sigma_0 = (\Delta; \gamma_0; \top; 1) \]

from Example 42. It is visualized in Fig. 8.1. We extend the state with additional information as described in Example 43. Thereby, the same chunks as in the global chunk store \( \Delta \). Additionally, we change the time to 0 according to the definition of the abstract semantics. This results in the following state:

\[ \sigma'_0 = (\Delta; \gamma_0; \text{DM}_0; 0) \]

where \( \text{DM}_0 \) is the declarative memory containing all chunks from \( \Delta \). Then, the following derivations are possible:

\[
\sigma'_0 \rightarrow_{\text{no}} (\gamma_1; \text{DM}_0; 0) \\
\rightarrow_{\text{inc}} (\gamma_2; \text{DM}_0; 0) =: \sigma_2,
\]

where
In $\sigma_0$ no-rule is applicable, but there is a pending request whose result is not visible for the production system. Hence, we can apply the no-rule transition which makes the chunk $b$ visible. Then the rule inc from Example 18 is applicable.

If we assume that $\text{request}_{\text{ret}}(\text{succ}, \{(\text{number}, 2)\}, v) = (c_{\text{req}}, 1, \emptyset)$ for all additional information $v$, where

$$c_{\text{req}} = (\text{succ}, \{(\text{number}, 2), (\text{successor}, 3)\}),$$

i.e. a chunk of type succ with the number 2 in the number slot and 3 in the successor slot, we reach the state $\sigma_2$ that is illustrated in Fig. 8.2.

Note that in this state $\sigma_2$ the no-rule transition is possible again. This transition just chooses one invisible chunk non-deterministically and makes it visible. It can always be applied non-deterministically. This means that if there was another rule applicable in $\sigma_2$, either the rule can be applied or the no-rule transition is used.

The additional information does not change in the rewriting process. This comes from the definition of the no-rule transition that never changes the additional information component. Furthermore, modifications do not change the additional information according to their definition. Requests to the declarative memory do not change additional information as well in the setting of our example as we only consider symbolic chunk structures in the declarative memory. As no new chunks are created or added to the declarative memory in a retrieval request, no additional information is added.

In general, requests can add additional information according to their definition. If sub-symbolic information of the structures in the declarative memory is considered, then a request to it will add information about the retrieval of a chunk to consider it when calculating activation levels. Since we abstract from those features, a retrieval request does not add any additional information.

### 8.3 Discussion

In this section, the results are discussed. First, the findings and their implications are summarized in Section 8.3.1 and then they are related to prior work in Section 8.3.2.

#### 8.3.1 Summary

The abstract semantics formalizes the symbolic part of the ACT-R architecture. This means, that the procedural system and its interaction
The abstract semantics describes transitions from actual implementations.

The semantics are designed to thoroughly interpret the informal descriptions of ACT-R.

The relation of the very abstract semantics to an independent formulation of ACT-R supports the soundness of the abstract semantics.

Figure 8.2: Visual representation of state $\sigma_2$ from Example 46 [GF15b, p. 120; GF18a, p. 22:21].

with the modules are in focus. Additionally, it captures all transitions that could be possible in an ACT-R model without the knowledge of actual values of sub-symbolic information. By concentrating on the core elements of ACT-R, our work makes analysis of cognitive models possible.

An important result that supports the suitability of the abstract semantics for program analysis has been presented in our prior work from [GF15b], where we have defined a refined semantics that formalizes some details of actual ACT-R implementations. The refined semantics aims for thoroughly describing the core of the reference implementation according to the reference manual [Bot] and is additionally closely related to our CHR implementation from [GF14]. In [GF15b], we have shown that the refined semantics is sound with respect to the abstract semantics, i.e. every computation of the refined semantics is a computation of the abstract semantics. This means that the abstract semantics captures all computations possible in ACT-R implementations regardless of actual sub-symbolic components. Therefore, we consider the abstract semantics as suitable for program analysis of the procedural system of ACT-R.

Since the semantics of ACT-R and its reference implementation is only given informally (c.f. [Bot; And+04; TLA06]), it is not possible to formally verify our definition of the semantics. We closely followed the descriptions in the reference manual and designed the semantics with the reference implementation in mind. The soundness result for the refined semantics supports the close relation of our formalization and implementations. In [GF14], an older version of our ACT-R implementation that is closely related to the formalization has been evaluated to match the results of typical ACT-R implementations with different conflict resolution mechanisms.

Furthermore, in [AW14b; AGW14] a formal semantics for ACT-R is presented that has emerged independently from our work (c.f. Section 8.4.1). Our abstract semantics that – to the best of our knowledge – was the first formal operational semantics for ACT-R, can be ex-
pressed in terms of the very abstract semantics. Since the very abstract semantics is an improved version of the independently developed semantics of Albrecht and Westphal, it is shown that the abstract semantics is a valid ACT-R formalization that can be expressed in terms of the semantics from [AW14b; AGW14]. This indicates that we have found some common understanding of what constitutes an ACT-R architecture.

Our semantics can be easily extended by new modules by changing the signature of the allowed additional information and by adding an interpretation of the corresponding request in the interpretation function of actions (c.f. Definition 59). The very abstract and abstract semantics ignore different conflict resolution mechanisms and represent rule conflicts by non-determinism. By specifying a different selection function (c.f. Definition 62) than in the abstract semantics (c.f. Section 8.2), different conflict resolution mechanisms can be represented. Since the selection function of the abstract semantics simply defines matching of production rules, it can be easily extended to filter the matching rules according to some conflict resolution mechanism. For instance, in the refined semantics of [GF15b] restricts the selection function such that it only selects at most one applicable rule, which resembles conflict resolution. The conflict resolution mechanism in the refined semantics itself still remains abstract and can therefore be exchanged, as we have demonstrated for our CHR implementation in [GF14].

8.3.2 Relation to Prior Work

This chapter is a revised version of parts of the formally published papers [GF15b; GF18a]. The very abstract semantics and abstract semantics have already been published in [GF15b; GF18a]. The refined semantics with the corresponding soundness result only appeared in [GF15b]. Since this thesis concentrates on program analysis of ACT-R models, we have only included the very abstract and abstract semantics that are designed for analysis. However, we want to point out the close relation of the refined semantics to the abstract semantics and to actual ACT-R implementations as discussed in Section 8.3. This supports the applicability of our work in this thesis for program analysis. The definition and the soundness result are documented in the formal publication [GF15b].

The abstract semantics has first been published in [GF15a], but in a less general fashion and without the theoretical foundation of the very abstract semantics. However, it was the first abstract formulation of ACT-R’s operational semantics. In [GF15b] we have expressed our abstract semantics in terms of an improved version of [AW14b] that we call the very abstract semantics.
The current definition of the abstract semantics and the definition in [GF15a] are closely related. The main difference is that in the new formulation results of requests are put directly into the requested buffer but are made invisible by the delay. Additionally, chunks in the old definition could be modified in-place. However, this would also be possible in the current definition by adding an in-place modification action symbol. In [GF15b] the differences of the formulations are discussed in detail.

8.4 Related work

We want to highlight the contribution of [AW14b] that we have used as a starting point to improve our work by unifying and extending it by our needs. We discuss this line of work in detail in Section 8.4.1. In Section 8.4.2 we summarize other work related to this thesis.

8.4.1 Semantics of Albrecht and Westphal

Our abstract semantics from [GF15a] has first been presented at the 24th International Symposium on Logic-Based Program Synthesis and Transformation (LOPSTR 2014) in Canterbury that took place from September 9–11 in 2014. The work of Albrecht and Westphal has been presented at the 12th Biannual Conference of the German Cognitive Science Society that took place from September 29 – October 2 in 2014 [AW14b] and has hence been developed independently from our work in [Gal13; GF15a]. Note that we refer to the revised selected version of the conference proceedings that appeared in 2015. The aim of our work in [GF15b] was to express our abstract semantics in terms of the semantics of [AW14b] to show its compliance with other approaches of formalizing ACT-R. The result then was an improved, but still closely related, version of the semantics of [AW14b].

In the following, we discuss the differences between our semantics in its different degrees of abstraction and the semantics of Albrecht and Westphal that appeared in [AW14b]. Albrecht and Westphal basically define a general production rule system that works on sets of buffers and chunks without specifying actual matching, actions and effects for the sake of modularity and reusability. We briefly summarize the differences between the semantics in [AW14b] and our very abstract semantics. For details, we refer to the original papers. The nomenclature in this work differs in some points from the original paper [AW14b] to unify it with our previous work.

The sets of buffers $B$ and action symbols $A$ are defined as in Section 4.2. For the sake of brevity, we have omitted the so-called buffer queries in our definition of the very abstract semantics. Queries are an additional type of test on the left-hand side of a rule. The very abstract semantics can be easily extended by queries. We have adopted
the definition of chunk types, chunks and cognitive states from the formalization of [AW14b], although the set of chunks in [AW14b] should be a multi-set-like structure as Example 39 shows. However, we have reduced the definition in our very abstract semantics by omitting the notion of a finite trace, which is a sequence \( \gamma_0, \gamma_1, \ldots \in \Gamma^+ \) of cognitive states. Those traces are used to compute the effects of an action. This definition seems inaccurate as the information of a finite trace that only logs the contents of the buffers at each step does not suffice to calculate sub-symbolic information. In typical definitions the calculation of production rule utilities needs the times of all rule applications that are not part of the trace. In other implementations and instantiations of ACT-R, there can be more additional information that is needed for sub-symbolic calculations. That is why we have extended the states by a parameter valuation function that abstracts from the information needed and leaves it to the architecture to define which information is stored.

In [AW14b], effects of actions with action symbol \( \alpha \in A \) are defined by an interpretation function \( I_\alpha : \Pi \to 2^{\Gamma_{\text{part}} \times 2^\Delta} \) (we have omitted queries as stated before). Similarly to the very abstract semantics, it assigns to each finite trace the possible effects of an action. Effects are a partial cognitive state that overwrites the contents of the buffers as in the very abstract semantics and a set \( C \subseteq \Delta \) that defines the chunks that are removed. In typical implementations of ACT-R, the chunks in \( C \) are moved to the declarative module which explains the need to define such a set. We have generalized this information by the notion parameter valuations that can be manipulated by an interpretation function. This enables us to abstract from the specific concept of moving chunks to declarative memory in our abstract semantics for example. Note that in [AW14b], the combination of interpretation functions to a rule interpretation is only stated informally. Additionally, we have extended the domain of an action interpretation function to actions, i.e. terms over the actions symbols in \( A \), and states instead of only action symbols, since more information is needed to calculate, like the parameters of the actions (i.e. the slot-value pairs) and information from the state.

The production rule selection function \( S : \Pi \to 2^R \) maps a set of applicable rules to each finite trace. In the very abstract semantics we have extended the domain from traces to a whole state since again additional information might be needed to resolve rule conflicts. With parameter valuations, we abstract from the information that is actually needed and leave it to the architecture definition. Additionally, our definition of selection function adds the notion of variable bindings that are not considered in [AW14b].
The operational semantics in [AW14b] is defined as a labeled, timed transition system with the following transition relation $\leadsto$ over time-stamped cognitive states from $\Gamma \times \mathbb{R}_0^+$:

$$(\gamma, t) \stackrel{r, d, \omega}{\pi} (\gamma', t')$$

for a production rule $r \in \mathcal{R}$, an execution delay $d \in \mathbb{R}_0^+$, a set of chunks $\omega \subseteq \Delta$ and a finite trace $\pi \in \Pi$, if and only if $r \in S(\pi, \gamma)$, i.e. $r$ is applicable in $\gamma$, the actions of $r$ according to the interpretation functions yield $\gamma'$ and $t' = t + d$.

Note that the set of chunks $\omega$ has been used but never defined in the original paper [AW14b]. We suspect that it represents an equivalent to the chunk store from our abstract semantics, i.e. the used subset of all possible chunks (which is how $\Delta$ is defined according to the paper). Although we consider it an integral part of ACT-R, the matching of rules – and particularly binding of variables by the matching – is completely hidden in $S$ or even not defined. On the one hand this simplifies exchanging the matching, on the other hand the function $S$ should then be defined slightly different to enable proper handling of variable bindings and conflict resolution as we discuss in Section 8.1.

In the original semantics according to [AW14b] there is no definition of what happens if there is no-rule applicable, but there are still effects of e.g. requests that can be applied. We have treated this case by adding the no-rule transition to the very abstract semantics.

### 8.4.2 Other Work

In this section, we give an overview of other approaches that try to formalize ACT-R.

**Declarative/Logic-based Cognitive Modeling** The framework of declarative/logic-based computational cognitive modeling (LCCM) [Bri08] tries to model human cognition by using a declarative knowledge representation and inference as the central process to manipulate the knowledge. The framework builds up a generalization of a logical system and uses logic-based computer programs that can be seen as a generalization of logic programming for its computations. Hence, oversimplified, LCCM models human cognition as the execution of a logic program [Bri08, p. 1].

Thereby, LCCM can be understood as a formal basis of typical cognitive architectures. It tries to extract the common foundations like representations and reasoning techniques – i.e. the building blocks – of those architectures in a top-down approach. In particular, this means that the structures are declarative and not numerical like in artificial neural networks [Bri08, p. 127].

The aim is not to compete with existing cognitive architectures: “These systems are all pitched at a level well above LCCM; they can
all be derived from LCCM. The formal umbrella used for the systematization herein is to offer a way to understand and rationalize all computational cognitive architectures that are declarative, that is, that are, at least in part, rule-based, explicitly logic-based, predicate-and-argument-based, propositional, and production-rule-based.” [Bri08, p. 128]

In that regard, LCCM is similar to our efforts. However, our work tries to formalize cognitive architectures (and in particular ACT-R) more concretely. Instead of a declarative semantics, our work consists of an operational semantics that is as abstract as possible, but still as concrete as necessary to analyze actual ACT-R models for computational features and then use that knowledge to improve the models. Hence, the degree of abstraction even of the very abstract semantics is way more concrete than for LCCM.

**Implementations** The reference implementation of ACT-R is described in a technical document [Bot] that defines the operational semantics mostly verbally and determines various technical details that are important for this exact implementation but not the architecture itself. In [GF15b], we have defined a semantics that describes the core of the reference implementation of ACT-R and show that every transition possible in this refined semantics is also possible in the abstract semantics. This shows that formal reasoning about our abstract semantics is meaningful to actual implementations.

There are approaches of implementing ACT-R in other languages, for example a Python implementation [SW07] or (at least) two Java implementations [Har08; Sal]. All those approaches do not concentrate on formalization and analysis, but only introduce new implementations. In [SW07] it is stated that exchanging integral parts of the ACT-R reference implementation is difficult due to the need of an extensive knowledge of technical details. They propose an architecture that is more concise and reduced to the fundamental concepts (that they also identify in their paper). However, their work still lacks a formalization of the operational semantics.

In [AGW14], the authors summarize the work on semantics in the ACT-R context. They also come to the conclusion that there are only new implementations available that sometimes try to formalize parts of the architecture, but no formal definition of ACT-R’s operational semantics. The authors use this result as a motivation for their work in [AW14b].

We describe an adaptable implementation of ACT-R using CHR in [Gal13; GF14; GF15c] that is based on our formalization. Due to the declarativity of CHR, the implementation is very close to the formalization and can therefore be considered as a preliminary version of an operational semantics for ACT-R. The implementation is easy to extend. This has been proven by exchanging the conflict resolution
mechanism (that is an integral part of typical implementations) with very low effort [GF14]. Even the integration of refraction, i.e. inhibiting rules to fire twice on the same (partial) state, has been exemplified and can be combined with other conflict resolution strategies. The translation presented there is close to both the core of the reference implementation and the abstract semantics whose abstract parts are defined such that they match the reference implementation and some of its extensions.
After formalizing the operational semantics of ACT-R in an abstract, adaptable fashion, the goal is to apply various analysis techniques to the semantics. Since the declarative programming language CHR features many theoretical analysis results and practical tools and has a clean logical framework due to its sound declarative semantics, it is shown in this chapter, how ACT-R can be embedded into CHR.

For this purpose, a formal translation scheme of ACT-R models to CHR programs is presented. The translation is then shown to be sound and complete with respect to the formal operational semantics of ACT-R defined in Section 8.2. This allows to transfer analysis results from the translated model back to the original ACT-R model and therefore forms the basis of the confluence test presented in Chapter 10.

The chapter is structured as follows: Section 9.1 covers the formal definition of the translation scheme, whereas in Section 9.2 the translation is shown to be sound and complete with respect to the operational semantics of ACT-R as defined in Section 8.2.

9.1 TRANSLATION TO CONSTRAINT HANDLING RULES

In this section, the translation of ACT-R models to CHR programs is defined. We start with the definition of a normal form of ACT-R rules that simplifies the translation process and proofs in Section 9.1.1 and clarify some notational aspects in Section 9.1.2. In Section 9.1.3 we show the translation of ACT-R states to CHR states and in Section 9.1.4 we define the translation scheme for rules.

9.1.1 SET NORMAL FORM

To simplify the translation scheme, we assume the ACT-R production rules to be in set normal form. The idea is that each buffer test contains each slot exactly once. In the general ACT-R syntax it is possible to specify more than one slot-value pair for each slot or none at all. If we assume set normal form, we can reduce the cases to consider in the translation and soundness and completeness proofs.

In the following, the set normal form is defined formally and it is shown that every ACT-R production rule can be transformed to set normal form preserving semantics.
DEFINITION 6.4 (Set Normal Form). An ACT-R rule $r := L \Rightarrow R$ is in set normal form, if and only if for all buffer tests $= (b, t, P) \in L$ and $s \in \tau(t)$ there is exactly one $v \in V_A \cup C_A$ such that $(s, v) \in L$.

THEOREM 12 (Set Normal Form). For all ACT-R models with rules $\mathcal{R}$, there is a set of rules $\Sigma'$ with the following properties: For all rules $r \in \Sigma$ with $r := L \Rightarrow R$ that are applicable in at least one state there is a rule $r' \in \Sigma'$ with $r' := L' \Rightarrow R$ in set normal form such that for all ACT-R states $\sigma, \sigma'$ it holds that $\sigma \Rightarrow \sigma'$ if and only if $\sigma \Rightarrow \sigma'$ (in the abstract semantics).

Proof. The set $\Sigma'$ that contains the rules of $\Sigma$ in set normal form can be constructed as follows. All rules in $\Sigma$ that are not applicable in any state do not appear in $\Sigma'$. For all remaining rules $r \in \Sigma$, we first transform $r$ to a rule $r'$ that is in set normal form. For every buffer test $= (b, t, P) \in L$ there is a test $= (b, t, P'\theta) \in L'$ for a substitution $\theta$. Thereby, $P'$ has the following slot-value pairs:

1. For all $s \in \tau(t)$ where there is exactly one $v \in V_A \cup C_A$ such that $(s, v) \in P$, $(s, v) \in P'$.
2. For all $s \in \tau(t)$ where there is no $v \in V_A \cup C_A$ such that $(s, v) \in P$, there is a slot-value pair $(s, V) \in P'$ for a fresh variable $V \in V_A$.
3. For all $s \in \tau(t)$ where there are are $v_1, \ldots, v_n \in V_A \cup C_A$ such that $(s, v_i) \in P$ for $1 \leq i \leq n$ we produce a substitution $\theta_b := \{v_1/v_1, \ldots, v_n/v_n\}$ and add a slot-value pair $(s, v) \in P'$ for a $v \in V_A \cup C_A$. There are the following cases:
   1. $v_1, \ldots, v_n \in V_A$: Then $v \in V_A$ is a fresh variable. Intuitively, we just introduce a new variable and replace all other variables by this new variable.
   2. $v_1, \ldots, v_n \in C_A$: Then $v = v_1$. Intuitively, since $r$ is applicable in at least one state, $v_1 = v_2 = \ldots = v_n$. The slot-value pairs are redundant and therefore we only add one of them.
   3. $v_1, \ldots, v_n \in V_A \cup C_A$: W.l.o.g., $v_1 \in C_A$. Then $v = v_1$.

We define the substitution $\theta$ as the composition of all substitutions $\theta_b$. The rule $r'$ is obviously in set normal form.

The proof idea is that if $r \subseteq \sigma$ all variables or constants that appear in the same slot test in the same buffer have the same values. If there is a buffer test $= (b, t, P)$ with two slot-value pairs $(s, v) \in P$ and $(s, v')$, then $\theta = v'\theta$ for $\theta = \text{Bindings}(r, \sigma)$. Hence, we can replace all occurrences of $v$ and $v'$ by $v$ and remove $(s, v')$ from $P$. This is exactly what our construction of the set normal form does.

Theorem 12 allows to assume that rules are in set normal form without loss of generality.
9.1.2 Notational Aspects

The following symbols and notations are special to this chapter and complement our list of Section 2.1:

- As usually, lists are denoted by enumerating their elements, e.g. \([a, b, c]\) for the list with elements \(a, b\) and \(c\). \([\,]\) denotes the empty list. Remember that we use \([H|T]\) for a list with head element \(H\) and tail list \(T\). We also use the following notation for list comprehension: \([x : p(x)]\) is the list with all elements \(x\) that satisfy \(p(x)\). Typically, \(p(x)\) limits the elements \(x\) to a finite set \(M\), e.g. \(p(x) \leftrightarrow x \in M \wedge \ldots\)

- For a function \(f : A \to B\) with finite domain \(A\), we define \([f] := [(a, b) : a \in A \land b \in B \land f(a) = b]\) sorted by an order on \(A\) and \(B\), i.e. the (sorted) enumerative list notation of the function \(f\). It can be understood as the list representation of the relational representation of \(f\) as a set of tuples \(f \subseteq A \times B\).

9.1.3 Translation of States

To translate an ACT-R state to CHR, we have to define translations for the individual components of such a state.

**Definition 65** (Translation of Chunk Stores). A (partial) chunk store \(\Delta\) can be translated to a first order term as follows:

\[
\text{[chunk}(c, t, [\text{val}]) : \text{c}: (t, \text{val}) \in \Delta]
\]

Thereby, \([f]\) denotes the explicit relational notation of the function \(f\) as a sorted list of tuples and \([x : p(x)]\) is the list comprehension as defined in Section 9.1.2.

We denote the translation of a (partial) chunk store \(\Delta\) with \(\text{chr}(\Delta)\).

Each chunk in a chunk store is translated to a term \text{chunk} that is member of a list. Note that we do not have defined the order of chunks in the list that represents the chunk store. The order is only necessary to make the translation unique. This allows to show soundness and completeness based on pure CHR state equivalence. However, as in the abstract semantics of ACT-R, we consider all permutations of such chunk store lists as equivalent. For the proofs the actual choice of order will also not make any difference. Hence, to simplify the proofs (where chunk stores are sometimes split into a part of interest and a rest), we just assume that list concatenation, chunk store merging, etc. reestablish the assumed order implicitly. Where necessary, we comment on that issue in our proofs in Section 9.2. The slot-value pairs in the chunk terms are sorted as defined in \([\,]\).

The cognitive state \(\gamma\) will be represented by \text{gamma} constraints that map a buffer \(b \in B\) to a chunk identifier and a delay. Since

The chunk store is represented by a user-defined constraint that contains a list of terms describing the chunks.

The cognitive state is represented by a user-defined constraint for each buffer.
additional information is already represented as logical predicates, we represent them as built-in constraints in the CHR store. We can now define the translation of an abstract state.

**Definition 66** (Translation of Abstract States). An abstract ACT-R state $\sigma := (\Delta; \gamma; v)$ can be translated to the following CHR state:

$$\langle \{\text{delta}(\text{chr}(\Delta))\} \cup \{\gamma(b, \text{id}_\Delta(c), d) \mid b \in B, c \in \Delta \land \gamma(b) = (c, d); v; \emptyset\} \rangle.$$

We denote the translation of an ACT-R state $\sigma$ by $\text{chr}(\sigma)$.

The chunk store is represented by a *delta* constraint that contains the translated chunk store as defined in Definition 65. Hence, a valid translation of an ACT-R state can only contain exactly one *delta* constraint.

For every buffer of the given architecture, a constraint *gamma* with buffer name, chunk id and delay is added to the state. Since $\gamma$ is a total function, every buffer has exactly one *gamma* constraint. Additionally, the chunk identifier in the *gamma* constraint must appear in exactly one *chunk* constraint because the co-domain of $\gamma$ refers to $\Delta$ and the chunk identifiers are unique.

Additional information is used directly as built-in constraints.

### 9.1.4 Translation of Rules

In our translation scheme, ACT-R rules are translated to corresponding CHR rules.

#### 9.1.4.1 Auxiliary Functions for Variable Names

To manage relations between newly introduced variables, we define some auxiliary functions. The functions all produce variable names from the set of variables $V_A$ for a set of arguments that are from the set of constants $C_A$ (or a subset of it) and are applied during the translation, i.e. they do not appear in the generated CHR code.

**Definition 67** (Variable Functions). Let $V_A, C_A$ be the set of variables and constants of an ACT-R architecture respectively, $B \subseteq C_A$ the set of buffers of this architecture and $V_{A,i} \subseteq V_A$ for $i = 1, \ldots, 6$ are disjoint subsets of the set of variables. Then the following auxiliary functions are defined:

**Chunk Variable Function** $CVar : B \rightarrow V_{A,1}, b \mapsto C_b$ that returns a fresh, unique variable $C_b$ for each buffer $b$. It identifies the chunk of a particular buffer in the translation.

**Delay Variable Function** $DVar : B \rightarrow V_{A,2}, b \mapsto D_b$ that returns a fresh, unique variable $D_b$ for each buffer $b$. It identifies the delay of a particular buffer in the translation.
RESULT VARIABLE FUNCTIONS $\text{ResStore}, \text{ResId}, \text{ResDelay} : \mathcal{B} \rightarrow \mathcal{V}_{\lambda}^i$

for $i = 3, 4, 5$ are defined as $b \mapsto X^i_b$ and return a fresh, unique variable $X^i_b$ for each buffer $b$. They are needed to memorize the results of an action.

MERGE VARIABLE FUNCTION $\text{MergeId} : \mathcal{B} \times \mathcal{V}_{\lambda}^b \rightarrow \mathcal{V}_{b,s}$ that returns a fresh, unique variable $X^s_{b,s}$ for each buffer $b$. It is needed to memorize the new chunk identifier after merging the chunk in $b$ with the existing chunk store.

COGNITIVE STATE VARIABLE FUNCTION $\text{CogState} : 2^\mathcal{B} \rightarrow 2^{\mathcal{B} \times \mathcal{V}_{\lambda}^1}$ is defined as $\text{CogState}(\mathcal{B}) := \{ (b, \text{CVar}(b)) : b \in \mathcal{B} \}$ (where the list is sorted by the $b$). The function returns a set of buffer-variable pairs for a set of buffers that connects each buffer with the chunk variable of its chunk. When equating $\text{CVar}(b)$ with $\text{id}_A(\gamma(b))$ for all buffers $b \in \mathcal{B}$, $\text{CogState}(\mathcal{B})$ can be considered as a pattern for the cognitive state.

The functions start with capital letters to emphasize that they represent CHR variables (or tuples of CHR variables in the case of $\text{CogState}$) in the translated rules.

9.1.4.2 Built-in Constraints for Actions

We define some built-in constraints that are needed for the translation. The idea is that they calculate the results of actions, chunk merging and chunk mapping as defined in the operational semantics of ACT-R. For actual instantiations of the abstract semantics (i.e. with a defined set of actions and chunk merging and mapping mechanisms), it has to be shown that their CHR implementations obey the properties that we define in the following.

Note that for all built-in constraints defined in this section, the notation $X = f(\cdot)$ for a logical variable $X$ and a function $f$ denotes mathematical/algebraic equivalence, i.e. the result of $f(\cdot)$ is syntactically equivalent to $X$ and not the function term $f(\cdot)$ itself. This algebraic equivalence then implies syntactic equivalence. More precisely, we can also write $X \equiv \text{res}$, if $f(\cdot) = \text{res}$.

DEFINITION 68 (Action Built-In Constraints). Let $\alpha := a(b, t, P)$ be an action and $\sigma := \langle \Delta; \gamma; \upsilon \rangle$ an ACT-R state. Let $D := \text{chr}(\Delta)$ be the CHR representation of the chunk store in $\sigma$ and $G := [\gamma]$ be the enumerative list representation of the cognitive state $\gamma$. For $(\Delta^*, \gamma^*, \upsilon^*) \in I(\alpha, \sigma^*)$, where $\gamma^*(b) := (c^*_b, d^*_b)$, the action constraint is defined as follows:

$$
\text{action}(\alpha, D, G, D_{\text{res}}, C_{\text{res}}, E_{\text{res}}) \land \upsilon
$$

$$
\leftrightarrow D_{\text{res}} = \text{chr}(\Delta^*) \land C_{\text{res}} = \text{id}_A(c^*_b) \land E_{\text{res}} = d^*_b \land \upsilon^*,
$$

This means that the action constraint is equivalent to the situation where the triple $(\Delta^*, \gamma^*, \upsilon^*) \in I(\alpha, \sigma^*)$ is a result of the interpretation of action $\alpha$ in the ACT-R state $\sigma$ with $\gamma^*(b) := (c^*_b, d^*_b)$ and the result variables for the
chunk store $D_{res}$, the chunk of the cognitive state $C_{res}$, and the delay of the cognitive state $E_{res}$ are syntactically equivalent to the translations of their ACT-R counterparts.

Intuitively, the action built-in constraint represents a function that gets the action $\alpha$, CHR representations of the constraint store $\Delta$ and the cognitive state $\gamma$ as input and returns by the help of the additional information $\upsilon$ the CHR representation of $I(\alpha, \sigma)$. The constraint theory of the action constraint is well-defined, since in Definition 59, $\gamma^*$ has the domain $\{b\}$ and co-domain $\Delta^* \times \mathbb{R}^+_0$, hence $\gamma^*(b)$ is defined. Since $c_b^* \in \Delta^*$, $id_{\Delta^*}(c_b^*)$ is defined.

9.1.4.3 Built-in Constraints for Chunk Merging

In the abstract semantics, chunk stores are merged by the operator $\circ$. In the following, built-in constraints are defined that implement $\circ$ and the corresponding mapping function $map$.

**Definition 69 (Merge Built-In Constraint).** For a set of chunk stores $\{\Delta_1, \ldots, \Delta_n\}$ with $D_i := chr(\Delta_i)$ for $i = 1, \ldots, n$, the built-in merge is defined as follows:

$$merge([D_1, D_2, \ldots, D_n], D) \leftrightarrow D = chr(\Delta_1 \circ \Delta_2 \circ \cdots \circ \Delta_n).$$

**Definition 70 (Map Built-In Constraint).** For two CHR representations of chunk stores $D := chr(\Delta)$ and $D' := chr(\Delta')$, the built-in constraint $map$ is defined as:

$$map(D, D', C, C') \leftrightarrow \begin{cases} 
C' = id_{\Delta \circ \Delta'}(map_{\Delta \circ \Delta'}(id_{\Delta}^{-1}(C))) & \text{if chunk}(C, T, P) \text{ in } D, \\
C' = id_{\Delta \circ \Delta'}(map_{\Delta \circ \Delta'}(id_{\Delta}^{-1}(C))) & \text{otherwise.}
\end{cases}$$

The main difference of this definition from the definition of the function $map$ from the abstract semantics is that the built-in constraint operates on chunk identifiers, whereas the function operates directly on chunks. Note that in the case that the chunk with identifier $C$ appears in neither in $D$ nor in $D'$, $C'$ is bound to $\text{nil}$ by definition of $id$ (see Definition 53).

9.1.4.4 List Operations

We use the built-in constraint $in$ to denote that a term is member of a list.

**Definition 71 (Member of a List).** For a term $c$ and a list $l$, the constraint $c$ in $l$ holds, if and only if there is a term $c'$ that is member of $l$ and $c = c'$.

Note that variables in $c$ are bound to the values in $l$ by this definition.
9.1.4.5 Translation Scheme for Rules

We can now define the translation scheme for rules.

**Definition 72 (Translation of Rules).** An ACT-R rule in set normal form \( r := L \Rightarrow R \) can be translated to a CHR rule of the following form:

\[
\begin{align*}
& \text{\( r : \{ \delta(D) \} \uplus \{ \gamma(b, CVar(b), DVar(b)) \mid b \in B \} \)} \\
\iff & \ \bigwedge_{(b,t,P) \in L} \text{chunk(CVar(b), t, P) in D} \land \text{DVar(b)=0} \\
& \uplus \{ \gamma(b, Merged(b), ResDelay(b)) \mid a(b, t, P) \in R \} \\
& \uplus \{ \gamma(b, CVar(b), DVar(b)) \mid a(b, t, P) \notin R \}, \\
& \bigwedge_{a=a(b, t, P) \in R} \text{action(a, D, CogState(B)}, \\
& \text{ResStore(b), ResId(b), ResDelay(b))} \\
& \land \text{merge([ResStore(b) : a(b, t, P) \in R], D')} \\
& \land \bigwedge_{a(b, t, P) \in R} \text{map(D, D', ResId(b), Merged(b)).}
\end{align*}
\]

Note that ACT-R constants and variables from \( C_A \) and \( V_A \) are implicitly translated to corresponding CHR variables.

We denote the translation of a rule \( r \) by \( chr(r) \) and the translation of an ACT-R model \( \Sigma \) that is a set of ACT-R rules by \( chr(\Sigma) \). Thereby, \( chr(\Sigma) := \{ chr(r) \mid r \in \Sigma \} \).

The intuition behind the translation can be described as follows:

The CHR rule tests the state for a \( \delta/1 \) constraint representing the chunk store and \( \gamma/3 \) constraints that come from the buffer tests of the rule. In the \( \gamma/3 \) constraints a variable for the chunk identifier is introduced. In the guard, the built-in constraint \( \text{in}/2 \) checks, if the chunk store represented as a list contains a term \( \text{chunk}/3 \) with the same type and slot-value pairs as specified in the buffer tests. The connection to the buffer is realized by the same variable for the chunk identifier (through the variable function \( CVar \)). Since the rule is in set normal form, the buffer tests are already completed (i.e. all slots are tested) and represented as a sorted list of slot-value pairs as in the state.

In the body of the rule, the built-in constraint calculates the result of each action from the right-hand side of the ACT-R rule. The resulting chunk stores are merged to one store \( D' \) by the built-in constraint \( \text{merge} \). Note that the order of merging is not specified by the translation scheme (as it is not specified by the ACT-R abstract semantics).

The built-in constraint \( \text{map}/4 \) implements the \( \text{map} \) function of the ACT-R semantics and gives access to the possibly modified chunk
identifier of all elements in \( D' \). By definition of \( \text{map} \), only chunk identifiers in \( D' \) are modified by merging. The resulting chunk identifiers are bound to a variable specified by the variable function \( 	ext{MergeId}/1 \).

The resulting chunk store \( \text{delta} \) and \( \text{gamma} \) constraints for all buffers are added. If the buffers have been modified, the \( \text{gamma} \) constraints points to the resulting chunk of the action, if not it shows to the chunk that has been in the buffer before.

**Example 47 (Translation of Rules).** In this example, the rule inc : \( L \Rightarrow R \) from Example 18 is translated to CHR. Let \( \alpha_g \in R \) and \( \alpha_r \in R \) be the actions for the goal (\( g \)) and the retrieval (\( r \)) buffer, respectively.

The following rule is the result of the translation chr(inc). For readability, multi-set brackets are omitted and built-in constraints and CHR constraints are mixed and separated by comma in the body.

\[
\text{inc} : \text{delta}(D), \\
\quad \text{gamma}(\text{goal}, C_g, E_g), \text{gamma}(\text{retrieval}, C_r, E_r) \\
\Leftrightarrow \\
\quad \text{chunk}(C_g, g, [(\text{current}, X)]) \text{ in } D \land E_g = 0 \land \\
\quad \text{chunk}(C_r, \text{succ}, [(\text{number}, X), (\text{successor}, Y)]) \text{ in } D \land E_r = 0 | \\
\quad \text{action}(\alpha_g, D, [(\text{goal}, C_g), (\text{retrieval}, C_r)], RD_g, RC_g, RE_g), \\
\quad \text{action}(\alpha_r, D, [(\text{goal}, C_g), (\text{retrieval}, C_r)], RD_r, RC_r, RE_r), \\
\quad \text{merge}([RD_g, RD_r], D'), \\
\quad \text{merge}([D, D'], D^*), \\
\quad \text{map}(D, D', RC_g, M_g), \\
\quad \text{map}(D, D', RC_r, M_r), \\
\quad \text{delta}(D^*), \\
\quad \text{gamma}(\text{goal}, M_g, RE_g), \\
\quad \text{gamma}(\text{retrieval}, M_r, RE_r).
\]

### 9.1.5 No-Rule Transition

In addition to transitions by rule applications, ACT-R can also have state transitions without rule applications. This is useful for instance, if no rule is applicable (i.e. computation is stuck in a state) but there are pending requests, then simulation time can be forwarded to the point where the next request is finished and its results are visible to the procedural system. This may trigger new rules and continue the computation.

The no-rule transition can be modeled in CHR by one individual generic rule:

\[
\text{no} : \text{gamma}(B, C, D) \Leftrightarrow D > 0 | \text{gamma}(B, C, 0)
\]
This transition is possible for all requests that are pending (i.e. that have a delay $D > 0$). Hence, the system chooses one request non-deterministically.

9.2 Soundness and Completeness of the Translation

In this section, we show that the translation from Section 9.1 is sound and complete w.r.t. the abstract semantics of ACT-R as defined in Section 8.2. This means that every transition that is possible in the abstract ACT-R semantics is also possible in CHR and vice versa. This allows for transferring proofs from program analysis in the translated CHR program back to the original ACT-R model.

The first step is to show that the results of the built-in constraints in the body of a translated ACT-R rule are equivalent to the interpretation of the right-hand side of the ACT-R rule. Therefore, we use that by definition the built-in constraints $\text{action}$, $\text{merge}$ and $\text{map}$ are equivalent to their ACT-R counterparts. Additionally, we have to show that the combination of the results of individual actions leads to the same result as the built-in constraints. We use induction to show that in the following lemma.

**Lemma 22 (Equivalence of Effects).** For an ACT-R rule $r := L \Rightarrow R$ in set normal form, a state $\sigma := (\Delta, \gamma, \upsilon, t)$ and $D = \text{chr}(\Delta)$. Let $I(r, \sigma) := (\Delta^*, \gamma^*, \upsilon^*)$ with $\lambda_{b \in \text{dom}(\gamma^*)} \gamma^*(b) = (c_b^*, d_b^*)$. Then the following two propositions are equivalent:

1. $\bigwedge_{a \in R} \text{action}(a_b, D, G, \text{ResStore}(b), \text{ResId}(b), \text{ResDelay}(b)) \land 
\upsilon \land
\text{merge}([\text{ResStore}(b) : a(b, t, P) \in R], D^*) \land
\text{merge}([D, D^*], D') \land
\bigwedge_{a \in R} \text{map}(D, D^*, \text{ResId}(b), \text{MergeId}(b))$

2. $D^* = \text{chr}(\Delta^*) \land D' = \text{chr}(\Delta \circ \Delta^*) \land \upsilon^* \land \upsilon \land
\bigwedge_{b \in \text{dom}(\gamma^*)} (\text{MergeId}(b) = \text{id}_{\Delta \circ \Delta^*}(\text{map}_{\Delta \Delta^*}(c_b^*) ) \land \text{ResDelay}(b) = d_b^*)$

**Proof.** We use induction over the number of actions in $R$.

**Base Case** ($|R| = 1$) Let $r := L \Rightarrow R$ be an ACT-R rule in set normal form with one action $a \in R$ and $\sigma := (\Delta, \gamma, \upsilon, t)$. Let $D = \text{chr}(\Delta)$ and $G = [\gamma]$. Let $I(r, \sigma) := (\Delta^*, \gamma^*, \upsilon^*)$ with $\lambda_{b \in \text{dom}(\gamma^*)} \gamma^*(b) = (c_b^*, d_b^*)$ and $I(a, \sigma) := (\Delta_a, \gamma_a, \upsilon_a)$ with $(c_a, d_a) = \gamma_a(b)$. The effects of the rule are equivalent to the final result in the body of the CHR rule.
We start with:
\[
\text{action}(\alpha, D, G, D_{res}, C_{res}, E_{res}) \land v \lor \\
\text{merge}([D_{res}], D^*) \land \\
\text{merge}([D, D^*], D') \land \\
\text{map}(D, D^*, C_{res}, C'_{res}).
\]

First of all, we reduce the \(\text{action}\) constraint by use of Definition 68:
\[
\text{action}(\alpha, D, G, D_{res}, C_{res}, E_{res}) \land v
\leftrightarrow D_{res} = \text{chr}(\Delta_\alpha) \land C_{res} = \text{id}_{\Delta_\alpha}(c_\alpha) \land E_{res} = d_\alpha \land v \lor v_\alpha.
\]

By Definition 69, we can reduce the \(\text{merge}\) constraints to
\[
D^* = \text{chr}(\text{chr}^{-1}(D_{res})) \land D' = \text{chr}(\text{chr}^{-1}(D) \circ \text{chr}^{-1}(D^*))
\]
which is by definition of \(D := \text{chr}(\Delta)\) from the assumptions and \(D_{res} = \text{chr}(\Delta_\alpha)\) from the last step equivalent to
\[
D^* = \text{chr}(\Delta_\alpha) \land D' = \text{chr}(\Delta \circ \Delta_\alpha).
\]

Since \(C_{res} = \text{id}_{\Delta_\alpha}(c_\alpha)\) and therefore a \textit{chunk} term with identifier \(C_{res}\) appears in \(D^*\) (and not in \(D\)), we can now reduce the \(\text{map}\) built-in by Definition 70 and get together with the definitions of \(D\) and \(D^*\) to
\[
\text{map}(D, D^*, C_{res}, C'_{res}) \leftrightarrow C'_{res} = \text{id}_{\Delta \circ \Delta_\alpha}(\text{map}_{\Delta_\alpha}(\text{id}_{\Delta_\alpha}^{-1}(C_{res}))).
\]

Since we have that \(C_{res} = \text{id}_{\Delta_\alpha}(c_\alpha)\), this is equivalent to
\[
C'_{res} = \text{id}_{\Delta \circ \Delta_\alpha}(\text{map}_{\Delta_\alpha}(c_\alpha)).
\]

By Definition 61, we have that for one action \(\Delta_\alpha = \Delta^*\) and \(c_\alpha = c^*\). Hence, this is equivalent to
\[
C'_{res} = \text{id}_{\Delta \circ \Delta^*}(\text{map}_{\Delta^*}(c^*)).
\]

All in all, this proves the proposition for \(|R| = 1\).

\textbf{Induction step} \((|R| \rightarrow |R| + 1)\) Let \(\sigma := \langle \Delta; \gamma; v \rangle\) be an ACT-R state and \(D := \text{chr}(\Delta)\) the CHR representation of the chunk store and \(G := [\gamma]\) the enumerative list representation of the cognitive state.

Let \(r' := L \Rightarrow R'\) with \(R' := R \cup \{\alpha\}\) be a rule that has been constructed from a rule \(r := L \Rightarrow R\). Let \(I(r, \sigma) := (\Delta^*, \gamma^*, v^*)\) with \(\bigwedge_{b \in \text{dom}(\gamma^*)} \gamma^*(b) = (c_b^*, d_b^*)\) be the interpretation of the smaller rule \(r\) with \(|R|\) actions.

Let \(I(r', \sigma) := (\Delta^{**}, \gamma^{**}, v^{**})\) with \(\bigwedge_{b \in \text{dom}(\gamma^{**})} \gamma^{**}(b) = (c_b^{**}, d_b^{**})\) be the interpretation of the rule \(r'\) that has \(|R| + 1\) actions.
We begin with

$$\bigwedge_{a_b \in R'} \text{action}(a_b, D, G, D_b, C_b, E_b)$$

$$\land \nu$$

$$\land \text{merge}([D_b : a(b, t, P) \in R'], D^{**})$$

$$\land \text{merge}([D, D^{**}], D'') \land$$

$$\bigwedge_{a_b \in R'} \text{map}(D, D^{**}, C_b, C_b'').$$

We need to apply the induction hypothesis. Therefore, we split the conjunction of actions over \(R' = R \cup \{a\}\) into a conjunction over \(R\) and an individual action constraint for \(a\). Additionally, the lists are split (w. l. o. g. since the order of chunks does not play a role and the lists can be merged again in the right order) and get the equivalent formula

$$\bigwedge_{a_b \in R} \text{action}(a_b, D, G, D_b, C_b, E_b)$$

$$\land \text{action}(a, D, G, D_a, C_a, E_a)$$

$$\land \nu$$

$$\land \text{merge}([D_b : a(b, t, P) \in R] + [D_a], D^{**})$$

$$\land \text{merge}([D, D^{**}], D'') \land$$

$$\bigwedge_{a_b \in R} \text{map}(D, D^{**}, C_b, C_b'') \land \text{map}(D, D^{**}, C_a, C_a').$$

By definition of \text{merge} (c. f. Definition 68) and associativity of \(\circ\) and neutral element \([\ ] = \text{chr}(\emptyset)\), we can split the merging as follows: We first merge the actions in \(R\) to \(D^*\) and then merge \(D^*\) with the result chunk of action \(a\) to \(D^{**}\):

$$\bigwedge_{a_b \in R} (\text{action}(a_b, D, G, D_b, C_b, E_b)$$

$$\land \text{action}(a, D_a, T_a, P_a, D_a)$$

$$\land \nu$$

$$\land \text{merge}([D_b : a(b, t, P) \in R], D^*)$$

$$\land \text{merge}([D^*, D_a], D^{**})$$

$$\land \text{merge}([D, D^{**}], D'') \land$$

$$\bigwedge_{a_b \in R} \text{map}(D, D^{**}, C_b, C_b'') \land \text{map}(D, D^{**}, C_a, C_a').$$

We can now introduce an intermediate result chunk store that merges the original store \(D\) with the results from \(r\), i.e. \(D^*\), to a chunk store \(D'\). We introduce some auxiliary variables \(C_b'\) that
map the chunk identifiers of the intermediate chunk store $D^*$ to the resulting chunk store $D^*$:

$$\bigwedge_{a_b \in R} \text{action}(a_b, D, G, D_{a_b}, C_{b}, E_b)$$

$$\land \text{action}(a, C_a, T_a, P_a, D_a) \land v$$

$$\land \text{merge}([D_b : a(b, t, P) \in R], D^*)$$

$$\land \text{merge}([D, D^*], D') \land$$

$$\bigwedge_{a_b \in R} \text{map}(D, D^*, C_{b}, C_{b}')$$

$$\land \text{merge}([D^*, D_{a}], D^{**})$$

$$\land \text{merge}([D, D^{**}], D'') \land$$

$$\bigwedge_{a_b \in R} \text{map}(D, D^{**}, C_{b}, C_{b}') \land \text{map}(D, D^{**}, C_a, C_a').$$

We can now apply the induction hypothesis:

$$(\ast) := v \land v^* \land$$

$$D^* = \text{chr}(\Delta^*) \land D' = \text{chr}(\Delta \circ \Delta^*) \land$$

$$\bigwedge_{b \in \text{dom}(\gamma^*)} (C_b = \text{id}_{\Delta \circ \Delta^*}(\text{map}_{\Delta \circ \Delta^*}(c_b^*)) \land E_b = d_b^*) \land$$

$$\text{merge}([D^*, \text{chunk}(C_a, T_a, P_a)], D^{**}) \land$$

$$\text{merge}([D, D^{**}], D'') \land$$

$$\bigwedge_{a_b \in R} \text{map}(D, D^{**}, C_{b}, C_{b}') \land \text{map}(D, D^{**}, C_a, C_a').$$

Thereby, it holds by definition of $I(r, \sigma)$ that $\text{dom}(\gamma^*) = \{ b \mid a_b \in R \}.

If we apply Definition 68 to the remaining action constraint of action $a$, we get

$$\text{action}(a, D, G, D_a, C_a, E_a) \land v$$

$$\leftrightarrow D_a = \text{chr}(\Delta_a) \land C_a = \text{id}_{\Delta_a}(c_a) \land E_a = d_a \land v \land v^*.$$

Hence, we have that

$$\text{merge}([D^*, D_a, D^{**}]) \leftrightarrow D^{**} = \text{chr}(\Delta \circ \Delta_a).$$

Thus, $D^{**}$ is the CHR version of the merging of the results from the actions in $R$ merged with the results from $a$. By Definition 61 of the interpretation of rules, we have that

$$I(r', \sigma) = (\Delta^* \circ \Delta_a, \gamma' \cup \gamma_a, v^* \land v_a).$$

This is equivalent to

$$I(r', \sigma) = (\Delta^{**}, \gamma^{**}, v^{**})$$
by definition of \( r, r' \) and the interpretation of rules (Definition 61).

The last merge constraint can be reduced to:

\[
\text{merge}([D, D^{**}], D'') \leftrightarrow D'' = \text{chr}(\Delta \circ \Delta^{**})
\]

The map constraints can be reconnected which yields

\[
\bigwedge_{a_s \in R} \text{map}(D, D^{**}, C_b, C''_b) \land \text{map}(D, D^{**}, C_a, C'_a)
\]

\[
\leftrightarrow \bigwedge_{a_s \in R'} \text{map}(D, D^{**}, C_b, C''_b)
\]

By Definition 62, for all defined buffers \( \gamma^{**}(b) = \text{map}_{\Delta^{**}}(c_b) \) and we have assumed that \( \gamma^{**}(b) = c^{**}_b \). By Definition 70, this yields

\[
C''_b = \text{id}_{\Delta \circ \Delta^{**}}(\text{map}_{\Delta^{**}}(c^{**}_b)).
\]

\( \text{id}_{\Delta^{**}}(c_b) \) exists, since the co-domain of \( \gamma^{**} \) is \( \Delta^{**} \times \mathbb{R}^+_0 \) by Definition 54 and the \( C_b \) are identifiers for the chunks in \( \gamma^{**} \).

If we apply all this to the conjunction in \((*)\), we get

\[
D^{**} = \text{chr}(\Delta^{**}) \land D'' = \text{chr}(\Delta \circ \Delta^{**}) \land
\bigwedge_{b \in \text{dom}(\gamma^{**})} (C''_b = \text{id}_{\Delta \circ \Delta^{**}}(\text{map}_{\Delta^{**}}(c^{**}_b))) \land E_b = d^{**}_b)
\land v^{**} \land v
\]

The next lemma proves that rule application transitions are sound.

**Lemma 23 (Soundness of Rule Applications).** For all ACT-R rules \( r \) and ACT-R states \( \sigma, \sigma' \in \Sigma_{\text{Aabs}} \): if \( \sigma \rightarrow^r \sigma' \) then \( \text{chr}(\sigma) \rightarrow^r \text{chr}(\sigma') \).

**Proof.** Let \( r := L \Rightarrow R \) and \( \sigma := \langle \Delta; \gamma; v; \emptyset \rangle \). Since \( r \subseteq \sigma \), we know that for every buffer test \( \beta := \langle b, t, P \rangle \) there is a substitution \( \theta \) such that \( \gamma(b) = (c::(t, \text{val}), 0) \) and for all \( (s, v) \in P : id_{\Delta}(\text{val}(s)) = v \theta \).

Let \( \rho := \text{chr}(\sigma) \) the CHR translation of the ACT-R state \( \sigma \). By Definition 66, we have that

\[
\rho \equiv \langle \{ \text{delta}((\text{chunk}(i_c, t, [\text{val}])) : c \in \Delta \land c = i_c::(t, \text{val})) \} \rangle
\]

\[
\forall \{ \text{gamma}(b, id_{\Delta}(c), d) \mid b \in B \land c \in \Delta \land \gamma(b) = (c, d) \} ;
\]

\[
v; \emptyset \rangle
\]

In the next step we split the state to only concentrate on the parts we are interested in for the rule application, i.e. gamma constraints for buffers that occur in a test. W.l.o.g., to simplify the visual representation of the proof, we do the same by splitting the representation of
the constraint store to the chunks that occur in buffers and all other chunks. Formally, this is an abuse of notation that does not harm the result as the order of chunks does not actually play a role and can be reestablished in the end without loss of information.\footnote{Furthermore, \textit{chr}(\Delta) does not define the particular order on the \textit{chunk} terms in the list in the \textit{delta} constraint.} We just assume that all operations like list concatenation simply reestablished the assumed order.

\[
\rho \equiv \langle \{ \text{delta}(\text{chunk}(i_c, t, \llbracket \text{val} \rrbracket)) : = (b, t, P) \in L \land \gamma(b) = (c, 0) \\
\quad \land c = i_c::(t, \text{val}) \} \cup \Delta \rangle \\
\quad \cup \{ \text{gamma}(b, \text{id}(c), d) | = (b, t, P) \in L \land \gamma(b) = (c, d) \} \cup \Gamma; v; \emptyset
\]

Note that by now we only have rewritten the CHR state without requiring that the tests we refer to match.

Due to the fact that \( r \subseteq \sigma \), we can apply this knowledge to the translated state:

\[
\rho \equiv \langle \{ \text{delta}(\text{chunk}(i_c, t, \llbracket \text{val} \rrbracket)) : = (b, t, P) \in L \land \gamma(b) = (c, 0) \\
\quad \land c = i_c::(t, \text{val}) \land \text{for all } (s, v) \in P : \text{id}(\text{val}(s)) = v\theta \} \cup \Delta \rangle \\
\quad \cup \{ \text{gamma}(b, \text{id}(c), 0) | = (b, t, P) \in L \land \gamma(b) = (c, 0) \} \cup \Gamma; v; \emptyset
\]

Since \( r \) is in set normal form, we can assume that there is only one test for each buffer in \( L \). Hence, we find a \textit{gamma} constraint in \( \rho \) for each such test without violating Definition \ref{def:gamma}, that only produces one \textit{gamma} constraint for each buffer.

Due to set normal form of \( r \) and hence totality of \( P \) w.r.t. the domain of \( \text{val} \), it is clear that \( P\theta = \llbracket \text{val} \rrbracket \). Hence, we can reduce the state to:

\[
\rho \equiv \langle \{ \text{delta}(\text{chunk}(\text{id}(c), t, P\theta)) : = (b, t, P) \in L \land \gamma(b) = (c, 0) \} \cup \Delta \rangle \\
\quad \cup \{ \text{gamma}(b, \text{id}(c), 0) | = (b, t, P) \in L \land \gamma(b) = (c, 0) \} \cup \Gamma; v; \emptyset
\]

We introduce a substitution with fresh variables to replace the chunk identifiers in the state, i.e. \( \theta' := \{ \text{CVar}(b)/\text{id}(c) | \gamma(b) = c \} \).

\[
\rho \equiv \langle \{ \text{delta}(\text{chunk}(\text{CVar}(b)\theta', t, P\theta)) : = (b, t, P) \in L \} \cup \Delta \rangle \\
\quad \cup \{ \text{gamma}(b, \text{CVar}(b)\theta', 0) | = (b, t, P) \in L \} \cup \Gamma; v; \emptyset
\]

Let \( \Theta := \theta^\uparrow \land (\theta')^\uparrow \) be the conjunction of syntactic equality constraints that can be derived from the substitution \( \theta \cup \theta' \) according to
Definition 29, i.e. each substitution $x/t \in \theta \cup \theta'$ appears in $\Theta$ as $x \equiv t$.
By Definition 7.1, we can move the substitution to the built-in store:
\[
\rho \equiv \{\delta([\text{chunk}(\text{CVar}(b), t, P)) : = (b, t, P) \in L] + D\} \\
\cup \{\gamma(b, \text{CVar}(b), 0) | = (b, t, P) \in L \} \cup \Theta; \Theta \cup v; \emptyset
\]

We introduce a fresh variable $D$ and add
\[
D = [\text{chunk}(\text{CVar}(b), t, P) : = (b, t, P) \in L] + D
\]
to the built-in store. Note that again we denote mathematical equality by $=\equiv$, which means that the result of evaluating $[\cdot] + D$ is bound to $D$ leading to a syntactic equivalence constraint $D = [\ldots]$ where $[\cdot]$ is the result list of the evaluated concatenation and list comprehension. This leads to the following equivalent state:
\[
\rho \equiv \{\delta(D)\} \cup \{\gamma(b, \text{CVar}(b), 0) | = (b, t, P) \in L \} \cup \Theta; \Theta \wedge D = [\text{chunk}(\text{CVar}(b), t, P) : = (b, t, P) \in L] + D \wedge v; \emptyset
\]

By Definition 7.1, we have that $D = d \leftrightarrow \{\text{chunk}(\text{CVar}(b), t, P) \in D | = (b, t, P)\}$. We can replace the corresponding built-ins by Definition 7.2:
\[
\rho \equiv \{\delta(D)\} \cup \{\gamma(b, \text{CVar}(b), 0) | = (b, t, P) \in L \} \cup \Theta; \Theta \wedge \{\text{chunk}(\text{CVar}(b), t, P) \in D | = (b, t, P)\} \wedge v; \emptyset
\]

Let $\text{chr}(r) := H \leftrightarrow G \mid B_c, B_b$. By Definition 7.2, we have that
\[
\rho \equiv \{H \cup \Theta \wedge G \wedge v; \emptyset\}
\]

By Definition 12, $\text{chr}(r)$ is applicable in $\rho \equiv \gamma(\sigma')$ and $\rho \rightarrow_r \rho'$ with
\[
\rho' \equiv \{B_c \cup \Theta \wedge G \wedge B_b \wedge v; \emptyset\}
\]

Due to Definition 7.2 of $\text{chr}(r)$, we have that
\[
B_b := \bigwedge_{a = \delta(b, t, P) \in R} (\text{action}(a, \text{ResId}(b), \text{restype}(b), \text{resslots}(b), \text{ResDelay}(b))) \wedge
\text{merge}([\text{chunk}(\text{ResId}(b), \text{restype}(b), \text{resslots}(b))]) : a(b, t, P) \in R, D') \wedge
\text{merge}(D, D', D') \wedge
\bigwedge_{a = \delta(b, t, P) \in R} (\text{map}(D, D', \text{ResId}(b), \text{MergId}(b)))
\]

$B_c := \{\delta(D')\} \cup
\{\gamma(b, \text{MergId}(b), \text{ResDelay}(b)) | a(b, t, P) \in R\}$
Since \( r \) is in set normal form, all buffers appearing in \( L \) also appear in \( R \). Hence, all \( \text{gamma} \) constraints removed by \( \text{chr}(r) \) are added in \( B_c \).

There is exactly one \( \text{delta} \) constraint in \( \rho \). It remains to show that the \( \text{delta} \) and \( \text{gamma} \) constraints are the ones describing the state \( \sigma' \).

By Lemma 22, we have that for all \( I(r, \sigma) \supseteq (\Delta^*, \gamma^*, \upsilon^*) \) with \( \gamma^*(b) = (c_b^*, d_b^*) \) for all \( b \in \text{dom}(\gamma^*) \)

\[
B_b \leftrightarrow D^* = \text{chr}(\Delta^*) \land D' = \text{chr}(\Delta \circ \Delta^*) \land \\
\bigwedge_{b \in \text{dom}(\gamma^*)} (\text{Merged}(b) = \text{id}_{\Delta \circ \Delta^*}(\text{map}_{\Delta \circ \Delta^*}(c_b^*))) \\
\land \text{ResDelay}(b) = d_b^* \\
\land \upsilon^* \land \upsilon
\]

Finally, when reestablishing the correct order on the chunk stores, we get by Definition 66 that \( \rho' \equiv \text{chr}(\sigma') \). Hence, the translation of rule applications is sound w.r.t. the abstract operational semantics of ACT-R.

We have now shown that our translation is sound regarding rule applications, i.e. a rule that can be applied in the abstract semantics of ACT-R can also be applied in the translated CHR program and leads to the same result. The state transition system of ACT-R has a second type of transitions: the \textit{no-rule transitions}, that can be applied if there is a buffer with a delay \( > 0 \), i.e. an invisible chunk. In that case, the no-rule transition allows us to make this chunk visible to the production system by setting the delay to zero. In the following it is shown that our translation is sound and complete regarding the no-rule transition.

**Lemma 24 (Soundness and Completeness of the No-Rule Transition).** For an ACT-R model \( M \), ACT-R states \( \sigma \) and \( \sigma' \) and their CHR counterparts \( \text{chr}(M), \text{chr}(\sigma) \) and \( \text{chr}(\sigma') \) the following propositions are equivalent:

1. \( \sigma \xrightarrow{\text{no}} \sigma' \) in the model \( M \)
2. \( \text{chr}(\sigma) \xrightarrow{\text{no}} \text{chr}(\sigma') \)

**Proof.** We start with Proposition 1. Let \( \sigma := \langle \Delta; \gamma; \upsilon \rangle \) and \( \sigma' := \langle \Delta; \gamma'; \upsilon \rangle \). In the abstract semantics, the no-rule transition is defined as

\[
\frac{\gamma(b) = (c^*, d^*) \land d^* > 0}{\langle \Delta; \gamma; \upsilon \rangle \xrightarrow{\text{no}} \langle \Delta; \gamma'; \upsilon \rangle}
\]

where \( \gamma'(b) = \begin{cases} (c^*, 0) & \text{if } b = b^*, \\ \gamma(b) & \text{otherwise} \end{cases} \)

The \textit{no-rule} transition in CHR is represented by the following CHR rule in \( \text{chr}(M) \):

\[
\text{no} : \gamma(B, C, D) \iff D > 0 \mid \gamma(B, C, 0)
\]
Since the no-rule transition is applicable in $\sigma$ for some arbitrary but fixed $b^*$, it holds that
\[ \gamma(b^*) = (c^*, d^*) \land d^* > 0. \]

Therefore, $\text{chr} (\sigma)$ must have the following form by Definition 66:
\[ \text{chr} (\sigma) \equiv \langle \gamma (B, C, D) \cup G; B=b^* \land C=id_B (c^*) \land D=d^* \land C; \emptyset \rangle. \]

Since $d^* > 0$, this is equivalent to
\[ \text{chr} (\sigma) \equiv \langle \gamma (B, C, D) \cup G; D > 0 \land B=b^* \land C=id_B (c^*) \land D=d^* \land C; \emptyset \rangle, \]

and therefore the rule no can be applied to $\text{chr} (\sigma)$ by Definition 12. This leads to the state $\rho'$ with
\[ \rho' \equiv \langle \gamma (b^*, c^*, 0) \cup G; C; \emptyset \rangle. \]

By Definition 66, we have that $\rho' \equiv \text{chr} (\sigma)$. The other direction is analogous. \qed

After proving soundness of rule applications and soundness and completeness of the no-rule transition, it remains to show completeness of rule applications.

**Lemma 25** (Completeness of Rule Applications). Let $\rho, \rho'$ be CHR states that have been translated from ACT-R states, i.e. there are ACT-R states $\sigma, \sigma'$ such that $\rho := \text{chr} (\sigma)$ and $\rho' := \text{chr} (\sigma')$. Let $r$ be a CHR rule translated from an ACT-R rule $s$. If $\rho \rightarrow_r \rho'$ then $\sigma \rightarrow_s \sigma'$.

**Proof.** The CHR rule $r := H \Leftarrow G \mid B_c, B_b$ has been translated from an ACT-R rule $s$. Let $s := L \Rightarrow R$ be an ACT-R rule in set normal form. Then $r$ has the following form by Definition 72:

\[
\begin{align*}
\delta (D) \cup \{ \gamma (b, CVar (b), DVar (b)) \mid b \in B \} \\
\land \bigwedge_{a = a(b,t,P) \in R} \text{chunk} (CVar (b), t, P) \text{ in } D \land DVar (b) = 0 \\
\{ \delta (D') \} \cup \\
\{ \gamma (b, Mergetype (b), ResDelay (b)) \mid a(b,t,P) \in R \} \\
\{ \gamma (b, CVar (b), DVar (b)) \mid a(b,t,P) \notin R \}, \\
\bigwedge_{a = a(b,t,P) \in R} (\text{action} (a, D, CogState (B), \\
\text{ResStore} (b), \text{ResId} (b), \text{ResDelay} (b))),
\end{align*}
\]

\[ \land \text{merge} ([\text{ResStore} (b) : a(b,t,P) \in R], D') \]
\[ \land \text{merge} ([D, D'], D^*) \]
\[ \land \bigwedge_{a = a(b,t,P) \in R} (\text{map} (D, D', \text{ResId} (b), \text{Merged}) \). \]
where $B_B$ are the built-in constraints of the body and $B_c$ the CHR constraints.

Since $r$ is applicable in $\rho$ by Definition 12, $\rho$ must have the following form:

$$\rho \equiv \langle H \uplus G; G \land C; V \rangle$$

Thereby, $G$ is a multi-set of user-defined constraints and $C$ a conjunction of built-in constraints. Since by definition of ACT-R states, $H \uplus G$ must be ground and therefore there must be some built-in constraints $\Theta$ in $C$ that bind those variables to the values from the state:

$$\rho \equiv \langle H \uplus G; \Theta \land G \land C; V \rangle$$

$\Theta$ is a conjunction of constraints of the form $V \equiv c$ for a variable $V$ and a term $c$, that binds the variables from $H$ and $G$ to the values from the state, i.e. the variables $CVar(b), DVar(b)$ (for all $b \in B$) and the variables appearing in the $P$ from the guard are bound to some values from the state. We denote $\theta$ as the substitution that follows from $\Theta = \theta^t$ according to Definition 29. Furthermore, $\rho$ must contain the additional information $\upsilon$:

$$\rho \equiv \langle H \uplus G; \Theta \land G \land \upsilon \land C'; V \rangle$$

In the following, the ACT-R state $\sigma$ is constructed. Let $\sigma := \langle \Delta; \gamma; \upsilon' \rangle$.

Thereby, $\text{chr}(\Delta) = D$ and for all $b \in B$:

$$\gamma(b) = (id_{\Delta}^{-1}(CVar(b)),DVar(b))\theta$$

by Definition 66. Additionally, $\upsilon' := \upsilon$. We can now apply the same equivalences as in Lemma 23 in reverse order and get

$$\rho \equiv \langle \{\delta \text{e}ta((\text{chunk}(i_{ic}, t, [val]) : (= (b, t, P) \in L \land \gamma(b) = (c, 0) \land c = i_{ic}(t, val) \land \text{for all } (s, v) \in P : id_{\Delta}(val(s)) = v\theta)\}
\uplus \{\gamma \text{amma}(b, id_{\Delta}(c), 0) \mid = (b, t, P) \in L \land \gamma(b) = (c, 0) \} \uplus G; \\
\Theta \land G \land \upsilon \land C'; \emptyset\rangle$$

From there it is clear that the ACT-R rule $s \sqsubseteq \sigma$ by Definition 63. The rule $s$ can be applied to $\sigma$ according to Definition 62 which yields

$$\sigma' := \langle \Delta \circ \Delta^*; \gamma'; \upsilon \land \upsilon^* \rangle$$

$(\Delta^*, \gamma^*, \upsilon^*) \in I(s\theta, \sigma)$ and

$$\gamma'(b) := \begin{cases} 
    (\text{map}_{\Delta^*}(c), d) & \text{if } \gamma^*(b) = (c, d) \text{ is defined} \\
    (\text{map}_{\Delta^*}(c), d) & \text{otherwise, if } \gamma(b) = (c, d).
\end{cases}$$
By Definition 12, the application of \( r \) in \( \rho \) leads to the state \( \rho' \) with
\[
\rho' \equiv (B_c \lor G; G \land B_b \land \Theta \land v; \emptyset).
\]

By Lemma 22, we get that \( B_b \) is equivalent to
\[
D^* = chr(\Delta^*) \land D' = chr(\Delta \circ \Delta^*) \land \\
\bigwedge_{b \in \text{dom}(\gamma^*)} (\text{MergeId}(b) = \text{id}_{\Delta \circ \Delta^*}(\text{map}_{\Delta \circ \Delta^*}(c^*))) \land \text{ResDelay}(b) = d_b^*
\]
\[
\land v^* \land v
\]

Hence, \( \rho' \) and \( \sigma' \) correspond directly to each other, i.e. \( \rho' \equiv chr(\sigma') \) and therefore rule transitions in the translation are complete w.r.t. the abstract operational semantics of ACT-R.

The results can be summarized in the following central soundness and completeness result of the embedding.

**Theorem 13 (Soundness and Completeness of the Embedding).** For an ACT-R model \( M \), ACT-R states \( \sigma \) and \( \sigma' \) and their CHR counterparts \( chr(M), chr(\sigma) \) and \( chr(\sigma') \) the following propositions are equivalent:

1. \( \sigma \rightarrow \sigma' \) in the model \( M \)
2. \( chr(\sigma) \rightarrow chr(\sigma') \)

**Proof.** This follows directly from Lemmas 23 to 25.

### 9.3 Discussion

In this section, the embedding presented in this chapter is summarized in Section 9.3.1 and then related to our prior work on the topic in Section 9.3.2.

#### 9.3.1 Summary

In this chapter, an embedding of ACT-R in CHR is presented. Therefore, a source-to-source transformation is described that translates ACT-R states and rules to CHR states and rules. The translation is then shown to be sound and complete with respect to the abstract operational semantics defined in Section 8.2. This means that every ACT-R transition step has a direct correspondence to a transition in the embedding and vice versa. This is a strong correspondence between the embedding and the semantics, since for soundness and completeness usually only the reflexive-transitive closures of the transition relations are required to correspond to each other (c.f. for instance the soundness and completeness results for CHR in [Frü09, p. 74; FA03, p. 47]). This is particularly interesting, because the abstract semantics is a formal description of ACT-R that features some non-deterministic
We rely on the order of a chunk store, because the list representations in the embedding have an order, whereas the sets in the ACT-R semantics does not.

We rely on the order of a chunk store, because the list representations in the embedding have an order, whereas the sets in the ACT-R semantics does not.

Note that the translation of states relies on an arbitrary order of the chunks in the chunk store. The reason for this is that in the operational semantics, chunk stores are represented as sets and therefore the order and multiplicities do not play a role. Our soundness and completeness results are formulated with respect to the order. Since the order can be arbitrarily chosen, the states can also be considered as equivalent modulo order (and multiplicity) of the chunk stores. In Chapter 10, we will define an equivalence relation over CHR states that ignores the order of the chunks in the chunk store and therefore allows to consider them as sets in the CHR embedding on a formal basis.

The modules of ACT-R are represented as built-in constraints in the embedding. In particular, the additional information of the state contains the built-in constraints that store the information of the modules including the sub-symbolic layer. The requests to the modules are then defined by a theory that solves the corresponding request given the constraints in the built-in store. This resembles the modular architecture of ACT-R, since our embedding only defines the procedural core of ACT-R, i.e. the procedural module and the interface to the modules. The modules themselves can then be added by adding an appropriate constraint theory.

By this approach, even model-specific adaptations of the modules can be integrated into the constraint theory. This can be used to improve program analysis, i.e. when reasoning about states with incomplete information represented by unbound variables which are not allowed in other ACT-R implementations.

Note that the constraint theory itself can be implemented in CHR again [Duc+03; Sch+06].

9.3.2 Relation to Prior Work

This chapter is a revised version of parts of the formally published papers [GF16; GF18a]. The translation of ACT-R models to CHR has been published first in [GF16]. However, that translation lacked a formal proof of soundness and completeness. Those properties have only been made plausible by an informal argumentation on the construction of the translation. The soundness and completeness results in Section 9.2 of this thesis first appeared in [GF18a].

Compared to [GF16], the translation presented in this thesis and in [GF18a] has been revised and improved significantly to make it suitable for proofs. The former translation from [GF16] has relied on macro-steps that simulate one ACT-R transition in CHR. In our current formulation in [GF18a] and in this thesis, we have changed this to a direct correspondence of ACT-R and CHR steps, since this simplifies the components like rule application. Due to the CHR embedding, an execution model for the translated ACT-R models can be derived directly.
proofs and leads to a stronger formal correspondence result. Furthermore, chunks are represented as individual constraints in the former translation, whereas the current formulation represents a complete chunk store in one constraint. The first chunk representation has advantages for implementations, whereas the latter simplifies the proofs. Both representations are equally powerful and can be interchanged. In general, the former translation can be regarded as an optimization for implementations of the current formulation that is more suitable for the analytical results. Nevertheless, both formulations are closely related semantically.

9.4 RELATED WORK

In this section, we discuss the relation of our embedding to other approaches.

OTHER EMBEDDINGS CHR embeddings of other rule-based approaches exist like embeddings of term rewriting systems, functional programming, event-condition-action (ECA) rules, production rules, General Abstract Model for Multi-set Manipulation (GAMMA) or logical algorithms (LA). An overview of those embeddings is given in [Frü09, pp. 141–170]. Those embeddings typically follow a similar motivation as our work: Since “CHR allows for their analysis, comparison, and cross-fertilization using a common platform” [Frü09, p. 143]. For logical algorithms, the CHR embedding even serves as the first known executable implementation of the theoretical formalism [Frü09, p. 169]. Furthermore, the embedding defines a relation to a logical theory due to the declarative semantics.

CHR allows to execute programs even though “some arguments are unknown or partially known as described by the built-in constraints” [Frü09, p. 129]. This feature is directly available for our embedding, where certain arguments, chunks, chunk stores or module data can be described by corresponding built-in constraints without actually defining ground values for all those components. Monotonicity of CHR then allows to reason from those incomplete states about all other states that extend this state, i.e. that consist of the same CHR constraints and satisfy the built-in constraints. This can directly be used in program analysis to reason from a finite amount of states about all larger states. This is not possible in other ACT-R implementations.

Due to ACT-R’s origins in production rule systems, the corresponding embedding is closest to our embedding of ACT-R. However, the resemblance is only marginal due to the special architecture and data structures of ACT-R. Furthermore, the production rule embedding summarized in [Frü09] is quite ad-hoc and lacks a soundness and completeness result.
As discussed in Section 8.4.2, the declarative/logic based framework LCCM described by Bringsjord tries to subsume cognitive architectures with a declarative, top-down approach [Bri08]. LCCM can be seen as a framework that consists of a declarative description of the building blocks of human cognition and cognitive architectures – namely declarative or propositional knowledge representation and inference over this knowledge [Bri08]. Hence, LCCM can be seen as the most general declarative semantics for all (at least partially symbolic and rule-based) cognitive architectures.

CHR has various well-formalized declarative semantics, e.g. based on first-order logic [Frü09, pp. 69–65] or linear logic [BF05; Bet14, pp. 73–104]. Due to the corresponding soundness and completeness results of those declarative semantics with respect to the operational semantics of CHR, our embedding automatically gives rise to a declarative semantics of the translated cognitive models. Due to soundness and completeness of our embedding, the declarative semantics is also valid for the original cognitive model.

Apart from our embedding, there are many implementations of ACT-R in different languages that reach from the Lisp reference implementation [Bot] to certain Java (e.g. [Sal]) or even Python implementations [SW07]. All of those approaches are efforts of getting rid of many technicalities that have been incorporated over time in the reference implementation, but none of them deal with formal analysis of ACT-R.
ACT-R introduces non-determinism to model learning and competing strategies. This resembles the applicability of more than one strategy in many situations. The strategy is chosen depending on information learned from situations in the past.

The inherent non-determinism leads to a desired non-confluence as soon as different strategies and concurring rules are involved. Nevertheless, a model typically contains parts that should not interfere with each other and that lead to deterministic results, i.e., usually derivations in cognitive models include states that are confluent. The deterministic and purposely non-deterministic parts of a cognitive model are interwoven and call each other mutually. This makes the behavior of a model often unpredictable and inhibits a clear understanding of the model.

By identifying the rules that lead to non-confluence, model quality can be improved: It allows to check if the model has the desired behavior regarding competing strategies and e.g., identify rules that interfere with each other unintentionally. Hence, although the human mind usually is considered to be non-confluent – supported by the implications of the conflict resolution strategies in the ACT-R theory – confluence analysis still can improve understanding of cognitive models and the relations between different strategies, how they interact and when they come into play. Furthermore, undesired rule conflicts can be uncovered.

This chapter gradually develops a decidable criterion for confluence of ACT-R using the CHR embedding. The general idea is to apply the confluence analysis methods from Chapters 5 and 6. Since the CHR embedding of ACT-R does not include arbitrary CHR states, plain confluence analysis cannot be applied since it leads to overlap states that do not represent valid ACT-R states. For this purpose, an ACT-R invariant A over CHR states is defined. It is satisfied if the CHR state has been derived from an ACT-R state, i.e., is a valid translated state. Then a decidable criterion for the invariant is presented and it is shown that the invariant is maintained in translated ACT-R models, i.e., it actually is an invariant according to Definition 19.

Furthermore, the proposed confluence analysis in this chapter makes use of a generalization of the CHR embedding from Chapter 9. In the embedding, an order over chunk stores is introduced that ensures
The embedding is generalized to allow for arbitrary orders of chunk stores.

The unambiguosity of the translation (c.f. Definition 65). The order allows for a direct correspondence of the CHR embedding and the ACT-R model only modulo plain CHR state equivalence without any further relaxations.

However, in further analytical considerations such an order on chunk stores may be bulky, in particular considering that the definition of the actual order does not play a role. Therefore, to better correspond to typical implementations and to simplify notations in this chapter, a generalization of the embedding is proposed by introducing a user-defined equivalence relation that discards the requested order over chunk stores. It is shown that the resulting confluence analysis (modulo equivalence) is equivalent to the corresponding invariant-based confluence test. This further substantiates that the order on chunk stores can be chosen arbitrarily and does not play a role. Hence, the equivalence relation can be introduced naturally to the embedding.

The generalization of the embedding is both a formal tool to simplify reasoning about ACT-R models and a significant example that demonstrates the power of the proposed analysis methods in Part ii.

Eventually, a decidable confluence test for ACT-R is proposed that is based on invariant-based confluence modulo equivalence analysis in the CHR embedding from Chapter 9. Note that both the proposed ACT-R invariant and equivalence relation can be used for other contexts than confluence analysis and therefore significantly contribute to our goal of enabling computational analysis of ACT-R models.

The chapter is structured as follows: First of all, in Section 10.1, the generalization of the embedding from Chapter 9 is presented and the corresponding equivalence relation over translated states is introduced. In Section 10.2, the ACT-R invariant is presented together with a decidable decision criterion and proof that is maintained by the transition system of the embedding. Based on the generalization of the embedding, the corresponding equivalence relation and the invariant, the confluence criterion is defined in Section 10.3 using invariant-based confluence modulo equivalence. The results are discussed in Section 10.4 followed by a comparison to related work in Section 10.5.

Note that in this chapter, we usually use $\sigma, \sigma_1, \ldots, \sigma', \ldots$ to denote ACT-R states and $\rho, \rho_1, \ldots, \rho', \ldots$ to denote CHR states.

### 10.1 Generalization of the Embedding

In the translation of ACT-R states, chunk stores are represented as lists with respect to a certain order as it can be seen in Definition 65. As already discussed, the order does not play a role for the soundness and completeness proofs in Chapter 9, and hence, chunk stores can be represented as lists without any order in the CHR states more conveniently.
However, when comparing two CHR states that represent ACT-R states as it is necessary for confluence, the actual first-order representation of the chunk store becomes important. Hence, to reason about the translated ACT-R models with unordered chunk store representations in a formally sound way, it is necessary to include the program analysis tools with user-defined equivalence relations from Part ii.

In this section, the generalized translation method is defined. Then, a user-defined equivalence relation for CHR representations of ACT-R states is defined that allows for reasoning about confluence of ACT-R models.

**Definition 73 (Generalization of the Embedding).** The generalized translation of (partial) chunk stores is equivalent to the translation in Definition 65 with the exception that no particular order is required for the resulting list of chunk terms and the relational representation of the slot-value function val. We denote the generalized translation of a (partial) chunk store $\Delta$ with $\text{chr}'(\Delta)$.

The generalized translation of an ACT-R state $\sigma$ is equivalent to the translation in Definition 66 except for the translation of the chunk store $\Delta$ that uses the function $\text{chr}'(\Delta)$ instead of $\text{chr}(\Delta)$. We denote the generalized translation of an ACT-R state $\sigma$ by $\text{chr}'(\sigma)$.

Note that only the translation of chunk stores is adapted in the generalized translation, the translation of rules remains unmodified. In Chapter 9, the built-in constraints for list concatenation, merging of chunk stores, etc. are assumed to implicitly respect the order of the chunk stores. This requirement is obsolete for the generalized embedding.

Based on the generalized translation of chunk stores and states, we define a notion of state equivalence that ignores the ambiguity of chunk store representations.

**Definition 74 (ACT-R State Equivalence $\approx^A$).** Let set be a function that returns the set representation of a list, i.e. that is defined as $\text{set}(L) := \{x | \text{member}(x, L)\}$.

The ACT-R state equivalence relation $\approx^A$ is the smallest $\models$-state equivalence relation with

$$\langle \text{delta}(\Delta); \Delta \models \Delta'; \{\Delta\} \rangle \approx^A \langle \text{delta}(\Delta'); \Delta \models \Delta'; \{\Delta, \Delta'\} \rangle,$$

where for two lists $L_1, L_2$ and their set representations

$$S_i := \{\text{chunk}(c, t, \text{set}(V)) | \text{chunk}(c, t, V) \in \text{set}(L_i)\} (i = 1, 2)$$

it holds that $L_1 \models L_2$ if and only if $S_1 = S_2$.

This means that two translated ACT-R states are equivalent, if the lists in their $\text{delta}$ constraint interpreted as sets are equivalent, i.e. they represent equivalent chunk stores. Thereby, the slot-value pairs in the chunk-terms are also interpreted as sets.
EXAMPLE 48 ($\approx^A$-Equivalent States). Consider the following two translated ACT-R states:

- $\sigma_1 := \langle \delta([\text{chunk}(c_1, t, [(s_1, v_1), (s_2, v_2)]),
\text{chunk}(c_2, t, [(s_1', v_1'), (s_2', v_2')])), \gamma(b, c_1); \top, \emptyset \rangle$,

- $\sigma_2 := \langle \delta([\text{chunk}(c_2, t, [(s_2', v_2'), (s_1, v_1')])),
\text{chunk}(c_1, t, [(s_2, v_2), (s_1, v_1)])), \gamma(b, c_1); \top, \emptyset \rangle$.

It is clear that $\sigma_1 \approx^A \sigma_2$.

Note that $\approx^A$ does not only discard the order on chunks but also multiplicities, although there are no multiplicities greater than 1 in the translation.

The following Lemma shows that the generalized translation is unique modulo the equivalence relation $\approx^A$, i.e. modulo order of chunk stores. This means that the equivalence relation complements the generalized translation such that it replaces the role of plain state equivalence from the original translation. Hence, $\approx^A$ captures the difference between the two translations.

**Lemma 26.** For all ACT-R states $\sigma, \sigma'$ it holds that $\sigma = \sigma'$ if and only if $\text{chr}'(\sigma) \approx^A \text{chr}'(\sigma')$.

**Proof.** This follows directly from Definitions 73 and 74.

We now show that the proposed equivalence relation is compatible and therefore our analysis tools from Part II are applicable.

**Lemma 27 (Compatibility of $\approx^A$).** The ACT-R state equivalence relation $\approx^A$ is $\Sigma$-compatible, where $\Sigma := \{\langle \delta(\Delta); \top; \{\Delta\}\rangle\}$.

**Proof.** Since $\approx^A$ is a $\approx$-state equivalence relation, this follows directly from Definition 42 and Lemma 15.

10.2 ACT-R INTEGRATION

To reason about confluence of ACT-R models in CHR, we need an invariant that restricts the CHR state space to states that stem from a valid ACT-R state. In the following example, we show how overlapping translated ACT-R rules can lead to overlap states that do not describe a valid ACT-R state.

**Example 49.** Consider the CHR rule

\[
\delta(D), \gamma(B, C, 0) \Leftrightarrow \text{chunk}(C, T, P) \text{ in } D | \ldots
\]

that has been obtained from an ACT-R rule. By overlapping the rule with itself, the following overlap state can be constructed:

\[
\sigma := \langle \delta(D), \gamma(B, C, 0), \gamma(B, C', 0); \text{chunk}(C, T, P) \text{ in } D \land \text{chunk}(C', T', P') \text{ in } D; \emptyset \rangle,
\]
where $V$ are the variables in $\sigma$. However, this state does not stem from a valid ACT-R state, since $\gamma$ is a function with only one value for each buffer and therefore the translation of an ACT-R state can never contain two gamma constraints for the same buffer $B$.

In the following, we define the ACT-R invariant $A$ on CHR states that limits the state space to states that stem from valid ACT-R states. We show that the invariant is decidable by breaking it down to five fine grained invariants. We also show that it actually defines an invariant for translated ACT-R models.

**Definition 75 (ACT-R Invariant).** Let $[\rho]$ be a CHR state. The ACT-R invariant $A$ holds if and only if there is an ACT-R state $\sigma$ such that $\rho \equiv \text{chr}(\sigma)$.

Basically, this means that $A([\rho])$ holds if $[\rho]$ is the valid translation of an ACT-R state. However, by this definition it is hard to decide if a CHR state satisfies the invariant.

We now show some decidable sub-invariants on CHR states and prove that their conjunction is equivalent to $A$. For this purpose, we define an auxiliary function $\text{ids}$ that returns the set of chunk identifiers for a list of chunk/3 terms.

**Definition 76 (Chunk Identifiers).** Let $d$ be a list. Then

$$\text{ids}(d) := \{ c \mid \text{chunk}(c, t, p) \text{ in } d \}$$

is the set of chunk identifiers of the list $d$.

The sub-invariants mainly consist of uniqueness invariants, i.e. they require that there is only one constraint of a certain kind for a class of arguments, and functional dependency invariants, i.e. that certain sets that represent relations appearing in constraints are functions. Eventually, the constraints that can be be used in a state are restricted.

**Theorem 14 (ACT-R Invariants).** Let $\rho := (G; B; V)$ be a CHR state. We define the following sub-invariants:

1. **Unique Chunk Store**

   $A_1([\rho]) \leftrightarrow$ There is exactly one constraint $\text{delta}(d) \in G$ for some ground list $d$. For all elements $e$ in list $d$, it holds that there exist $c \in C_A, t \in T, s \in \tau(t), v \in C_A$ such that $e = \text{chunk}(c, t, p)$ and $p$ is a sub-list of $[\{(s, v) : s \in \tau(t) \land v \in \text{ids}(d)\}]$, such that it is total in the sense that for all $s \in \tau(t)$ there is a pair $(s, v)$ in $p$.

2. **Functional Dependency of Cognitive State**

   $A_2([\rho]) \leftrightarrow$ For all buffers $b \in B$ there is exactly one $\text{gamma}(b, c, e) \in G$ where $c \in \text{ids}(d)$ for some $\text{delta}(d) \in G$ and $e \in \mathbb{R}_0^+$. 

The ACT-R invariant $A$ restricts the CHR state space to only valid embeddings of ACT-R states. The invariant $A$ can be divided into five sub-invariants.
3. **Unique Chunk Identifiers**

$$\mathcal{A}_3([\rho]) \iff \text{For all chunk identifiers } c \in \mathcal{C}_A \text{ and constraints } \delta(d) \in \mathcal{G}, \text{ if there is a term } \text{chunk}(c, t, p) \text{ in the chunk store list } d, \text{ then there is no other term } \text{chunk}(c', t', p') \text{ in } d.$$ 

4. **Functional Dependency of Slot-Value Pairs**

$$\mathcal{A}_4([\rho]) \iff \text{For all constraints } \delta(d) \in \mathcal{G}, \text{ terms } \text{chunk}(c, t, p) \text{ in list } d \text{ and } (s, v) \text{ in list } p, \text{ there is no other term } (s, v') \text{ in } p.$$ 

5. **Allowed Constraints**

$$\mathcal{A}_5([\rho]) \iff \text{In } \mathcal{G} \text{ there are only } \delta(1) \text{ and } \gamma(3) \text{ constraints, only syntactic equality } \equiv/2 \text{ and the allowed constraints defined by the ACT-R architectures appear in } \mathcal{B} \text{ and } [\rho] \text{ is ground.}$$

For all CHR states $[\rho]$ it holds that $\mathcal{A}([\rho]) \iff \bigwedge_{i=1}^{5} \mathcal{A}_i([\rho]).$

**Proof.** "\(\Rightarrow\)" If $\mathcal{A}([\rho])$, then $[\rho]$ is the product of the translation of an ACT-R state. It follows directly from definition 66 that in that case, $\mathcal{A}_1([\rho]), \mathcal{A}_2([\rho]), \mathcal{A}_3([\rho]), \mathcal{A}_4([\rho])$ and $\mathcal{A}_5([\rho])$ hold.

"\(\Leftarrow\)" We have to show that for all CHR states $[\rho]$ where the invariants $\mathcal{A}_1([\rho]), \mathcal{A}_2([\rho]), \mathcal{A}_3([\rho]), \mathcal{A}_4([\rho])$ and $\mathcal{A}_5([\rho])$ hold, there is an ACT-R state $\sigma$ such that $\rho \equiv \text{chr}(\sigma)$. Let $[\rho] := ([\mathcal{G}; \mathcal{B}; \mathcal{V}]).$

We construct the ACT-R state $\sigma := (\Delta; \gamma; v)$. Since $\mathcal{A}_1([\rho])$, there is exactly one $\delta(d)$ constraint for a list $d$ and all elements in $d$ are of the form $\text{chunk}(c, t, p)$ where $c \in \mathcal{C}_A$, $t \in \mathcal{T}$ and $p$ is a list of elements $(s, v)$ with $s \in \tau(t)$ and $v \in \text{ids}(d)$. The list $p$ is total with respect to $s$ and the $v$ are chunk identifiers that appear in $d$. Due to $\mathcal{A}_4$, there is exactly one $(s, v)$ in the list $p$ for each $s \in \tau(t)$, hence $p$ is the relational list representation of a value function. The invariant $\mathcal{A}_3$ guarantees that the chunk identifiers are unique.

We define $\Delta := \{(t, p) \mid \text{chunk}(c, t, p) \text{ in } d\}$ with the identifier function $\text{id}_d := \{(t, p), c \mid \text{chunk}(c, t, p)\}$.

Due to invariant $\mathcal{A}_2$, the cognitive state can then be defined for all $b \in \mathcal{B}$ such that $\gamma(b) := (\text{id}_d^{-1}(c), e)$ for each $\gamma(c, b, e) \in \mathcal{G}$.

Since $\mathcal{A}_5([\rho])$, $[\rho]$ is ground. Hence, we can find another representative of the state with $\rho \equiv (\mathcal{G}'; \mathcal{B}'; \mathcal{V})$, that applies all equality constraints $\equiv t$ in $\mathcal{B}$ such that only constants appear in $\mathcal{G}'$ and $\mathcal{B}'$ and $\mathcal{B}'$ only consists of allowed predicates defined by the ACT-R architecture. Therefore, we can set $\nu := \mathcal{B}'$.

From the construction of $\sigma$ it is clear that $\rho \equiv \text{chr}(\sigma)$. \(\square\)

The invariants $\mathcal{A}_1, \ldots, \mathcal{A}_5$ are obviously decidable. Since they are equivalent to the ACT-R invariant $\mathcal{A}$, Theorem 14 gives us a decidable criterion for the ACT-R invariant $\mathcal{A}$.

In the next step, we show that the ACT-R invariant $\mathcal{A}$ is maintained by transitions that come from a translated ACT-R program, i.e. that it actually is an invariant.
10.3 Invariant-Based Confluence Test

We want to use the invariant-based confluence modulo equivalence criterion from Chapter 6 to prove confluence modulo $\approx^A$ of all states $[\rho]$ that satisfy the ACT-R invariant $A$ modulo order of chunk store representations. We first construct the criterion for the $\alpha$-property in Section 10.3.1. Then, we show that the $\beta$-property is satisfied for all valid embeddings in Section 10.3.2. Hence, it is not required to prove for every individual model that the $\beta$-property is satisfied. The two results are combined to a confluence criterion for ACT-R Section 10.3.3.

10.3.1 Criterion for the $\alpha$-Property

To prove the $\alpha$-property of a model, we have to construct the set $\Sigma^A([\rho])$ for each overlap state $[\rho]$ that does not satisfy $A$. It contains all states that can be merged to $[\rho]$ such that they satisfy $A$. The minimal elements in this set have to be considered in the confluence test.

We will see that for all such states $[\rho]$ that do not satisfy $A$, the set of minimal elements is empty. Intuitively, this means that there are no states that can extend $[\rho]$ such that it satisfies $A$.

**Lemma 29 (Minimal Elements for $A$).** Let $A$ be the ACT-R invariant as in Definition 75. For all overlap states $[\rho]$ such that $A([\rho])$ does not hold, $\Sigma^A([\rho]) = \emptyset$ and therefore $M^A([\rho]) = \emptyset$.

**Proof.** Let $[\rho] := ([G; B; V])$. We use Theorem 14 that allows us to analyze the individual sub-invariants:

1. If $A_1$ is violated, there are the following cases:
   - There are two constraints $\delta(d), \delta(d') \in G$. We cannot extend $[\rho]$ (i.e. add constraints) to satisfy $A_1$.
   - There is only one unique $\delta(d) \in G$, with elements that do not have the required form. Again, no constraints can be added to satisfy $A_1$.

2. If $A_2$ is violated, there are two constraints $\gamma(b, c, e), \gamma(b', c', e') \in G$.

We cannot satisfy $A_2$ for such a state.
3. The proof for $\mathcal{A}_3$ and $\mathcal{A}_4$ is analogous to the previous arguments.

4. If $\mathcal{A}_5$ is violated, there are other constraints then delta or gamma in $\mathcal{G}$ or other than the allowed constraints defined by the architecture in $\mathcal{B}$. This cannot be repaired by extending $\mathcal{G}$ or $\mathcal{B}$.

\[\square\]

Note that for states $\rho$, where $\mathcal{A}([\rho])$ holds, it follows from Lemma 9 that $\mathcal{M}^\mathcal{A}([\rho]) = \{[\sigma_0]\}$. Hence, Lemma 29 simplifies the $\alpha$-property test for $\mathcal{A}$. We only have to consider the overlap states without any extensions.

10.3.2 Criterion for the $\beta$-Property

In this section, the preliminaries for a $\beta$-property test are discussed. Recall that the $\beta$-property requires all equivalent states that satisfy the invariant $\mathcal{A}$ to be joinable modulo $\approx^\mathcal{A}$, if in one of the states a rule is applicable. However, for the combination of invariant $\mathcal{A}$ and equivalence relation $\approx^\mathcal{A}$, we can show that the $\beta$-property is satisfied for all translated models and hence is not required to be considered in the confluence test.

We begin with the observation that the equivalence relation $\approx^\mathcal{A}$ maintains the invariant.

**Lemma 30 ($\approx^\mathcal{A}$ Maintains $\mathcal{A}$).** The ACT-R state equivalence relation $\approx^\mathcal{A}$ maintains the ACT-R invariant $\mathcal{A}$.

**Proof.** It has to been shown for all $\rho_1, \rho_2$ that if $\rho_1, \rho_2$ then $\mathcal{A}([\rho_1]) \leftrightarrow \mathcal{A}([\rho_2])$.

Let $\rho_1, \rho_2$. Since the two states are $\approx^\mathcal{A}$-equivalent, we can derive ordered variants $\rho'_1$ and $\rho'_2$ of $\rho_1$ and $\rho_2$ with respect to some order on the chunk stores such that $[\rho'_1] = [\rho'_2]$. Since $\rho'_1$ and $\rho'_2$ are state equivalent, it follows that $\mathcal{A}([\rho'_1]) \leftrightarrow \mathcal{A}([\rho'_2])$. Since no sub-invariant of Theorem 14 depends on the order of chunks, it follows that $\mathcal{A}([\rho_1]) \leftrightarrow \mathcal{A}([\rho_2])$.

\[\square\]

Recall that due to Lemma 29, the set of minimal extensions consists only of the empty state, i.e. $\mathcal{M}^\mathcal{A}([\rho]) = \{[\sigma_0]\}$, if the invariant $\mathcal{A}([\rho])$ holds. Otherwise, it is empty, i.e. $\mathcal{M}^\mathcal{A}([\rho]) = \emptyset$ otherwise. From the fact that the invariant is maintained by $\approx^\mathcal{A}$ and Lemma 29 it then follows directly that for two $\approx^\mathcal{A}$-equivalent states $\rho_1$ and $\rho_2$ the set $\mathcal{M}^\mathcal{A,\approx^\mathcal{A}}([\rho_1], [\rho_2])$ necessary for the invariant-based confluence modulo equivalence criterion only contains empty extensions or is empty itself.

**Corollary 9 (Minimal Elements for $\mathcal{A}$ and $\approx^\mathcal{A}$).** For two equivalent states $[\rho_1] \approx^\mathcal{A} [\rho_2]$, it holds that

$$
\mathcal{M}^{\mathcal{A,\approx^\mathcal{A}}}([\rho_1], [\rho_2]) = \begin{cases} 
\{([\sigma_0], [\sigma_0])\} & \text{if } \mathcal{A}([\rho_1]) \text{ and } \mathcal{A}([\rho_2]) \text{ hold,} \\
\emptyset & \text{otherwise.}
\end{cases}
$$
Proof. This follows directly from Lemmas 21, 29 and 30.

Note that since $\mathcal{A}$ is maintained by $\approx^A$ (c.f. Lemma 30), it is clear that either both $\mathcal{A}([\rho_1])$ and $\mathcal{A}([\rho_2])$ are satisfied, or both are not. Hence, the case distinction above is complete.

Lemma 29 and Corollary 9 allow us to ignore the minimal extensions both in the $\alpha$- and the $\beta$-property. In the next step, we even show that the entire $\beta$-property test becomes obsolete for the ACT-R embedding (and hence no minimal elements have to be considered there anyways).

**Lemma 31 (β-Property Satisfied in Embedding).** For all embedded states $\rho, \rho'$ with $[\rho] \approx^A [\rho']$: If $[\rho] \rightarrow [\tau]$, then $[\tau] \downarrow \approx^\delta [\rho']$.

Proof. Let $\rho, \rho'$ be two embedded states with $[\rho] \approx^A [\rho']$. This means that both states have been derived from (equivalent) ACT-R states, i.e. $\text{chr'}(\cdot) \equiv \rho$ and $\text{chr'}(\cdot) \equiv \rho'$.

Let $[\rho] \rightarrow_r [\tau]$ for some rule $r$. Let

$$[\rho] = [\langle \text{delta}(\Delta); \top; \{\Delta}\rangle] \circ_V [\delta].$$

This split of $\rho$ exists, because $\rho$ is an embedded state and therefore must contain a $\text{delta}$/1 constraint due to Definition 66. For all constraint stores $\Delta'$ with $\Delta \equiv \Delta'$ it holds that

$$[\rho] = [\langle \text{delta}(\Delta); \Delta \equiv \Delta'; \{\Delta}\rangle] \circ_V [\delta],$$

due to Definition 7.2. According to Definition 74, there is a state $\rho'$ such that

$$[\rho] = [\langle \text{delta}(\Delta); \Delta \equiv \Delta'; \{\Delta}\rangle] \circ_V [\delta]$$

$$[\rho'] = [\langle \text{delta}(\Delta'); \Delta \equiv \Delta'; \{\Delta, \Delta'}\rangle] \circ_V [\delta]$$

for some set of variables $V$ and some state $\delta$. Recall that $\equiv$ roughly denotes that the chunk stores interpreted as sets are equivalent (and the chunk terms represent the same chunks modulo order and multiplicity of the slot-value pairs). Hence, the two states only differ in the $\text{delta}$/1 constraints such that $\Delta \equiv \Delta'$.

According to Definition 72, the only constraints over the argument of the $\text{delta}$/1 constraint in the rule are defined by the built-in constraint $in$. It is clear that for all chunks $c$ in $\Delta$, it follows that $c$ in $\Delta'$ since $\Delta \equiv \Delta'$. Hence, the rule $r$ is applicable both in $[\rho]$ and $[\rho']$.

Let the successor state of $\rho'$ be $\tau'$, i.e. $[\rho'] \rightarrow_r [\tau']$. It follows from Definitions 72 and 74 that $[\tau] \approx^A [\tau']$ and hence $[\tau] \downarrow \approx^A [\rho']$.

10.3.3 Definition of the Confluence Criterion

Lemma 31 shows that the $\beta$-property is satisfied for all translated states and hence the $\beta$-property test for individual models is obsolete.
Hence, we can directly apply the $\alpha$-property test from Lemma 17. Due to Lemma 29, the $\alpha$-property test collapses such that the overlap states are not required to be extended by (non-trivial) minimal elements.

**Corollary 10 (A-Local Confluence Modulo $\approx^A$).** A CHR program is $A$-local confluent modulo $\approx^A$ if and only if for all critical pairs $(\rho_1, \rho_2)$ with overlap $\rho$ for which $A(\rho)$, it is $\rho_1 \downarrow^A \rho_2$.

**Proof.** This follows directly from Theorem 10 and Lemmas 29 and 31. \hfill \qed

Due to the structure of the ACT-R embedding that leads to invariant $A$ and equivalence relation $\approx^A$ the confluence test coincides with the regular confluence test of CHR as defined in Theorem 5 with the addition that joinability is relaxed such that the order of chunk stores does not play a role. The reason is that $A$ leads to only empty minimal extensions and the $\beta$-property is trivially satisfied in the embedding.

Note that for overlaps where $A(\rho)$ does not hold, Theorem 10 is trivially satisfied, since there are no minimal extensions that extend $\rho$ such that the invariant holds, i.e. $M^2(\rho) = \emptyset$. For overlaps where $A(\rho)$ holds, Lemma 9 guarantees that the unique minimal element is the empty state $[\sigma_0]$ which is the neutral element for state merging (c.f. Lemma 5). Therefore, if $A(\rho)$ holds, it suffices to test the critical pairs that stem from the overlap states $\rho$.

We now have a criterion to decide $A$-confluence of $A$-terminating CHR programs that have been translated from an ACT-R model. In the next theorem, we show that $A$-confluence of such CHR programs actually coincides with ACT-R confluence, i.e. that the method of proving $A$-confluence modulo $\approx^A$ in the embedding is equivalent to confluence of the original model. Therefore, the confluence criterion is applicable to decide confluence of ACT-R models.

**Theorem 15 (Confluence in ACT-R).** Let $M$ be an ACT-R model. Then $M$ is terminating and confluent if and only if $\text{chr}(M)$ is $A$-terminating and $A$-confluent modulo $\approx^A$.

**Proof.** $A$-termination is maintained due to soundness and completeness. We now show that confluence for terminating models and their CHR counterparts coincides. Confluence is defined as $(\sigma \rightarrow^* \sigma_1) \land (\sigma \rightarrow^* \sigma_2) \rightarrow (\sigma_1 \downarrow \sigma_2)$ for all states $\sigma, \sigma_1, \sigma_2$. It remains to show that joinability in ACT-R and CHR are equivalent modulo $\approx^A$, i.e. $(\sigma_1 \downarrow \sigma_2) \leftrightarrow ([\text{chr}(\sigma_1)] \downarrow^A [\text{chr}(\sigma_2)])$.

"$\Rightarrow$": If $(\sigma_1 \downarrow \sigma_2)$, there is a state $\sigma'$ such that $\sigma_1 \rightarrow^* \sigma'$ and $\sigma_2 \rightarrow^* \sigma'$. Due to soundness and completeness of the embedding modulo the first-order representation of the chunk stores, we have that $[\text{chr}(\sigma_1)] \rightarrow^* [\text{chr}(\sigma'_1)]$ and $[\text{chr}(\sigma_2)] \rightarrow^* [\text{chr}(\sigma'_2)]$ and $[\text{chr}(\sigma'_1)] \approx^A [\text{chr}(\sigma'_2)] \approx^A [\text{chr}(\sigma')]$. 

"$\Leftarrow$": If $(\sigma'_1 \downarrow \sigma'_2)$, there is a state $\sigma'$ such that $\sigma'_1 \rightarrow^* \sigma'$ and $\sigma'_2 \rightarrow^* \sigma'$. Due to soundness and completeness of the embedding modulo the first-order representation of the chunk stores, we have that $[\text{chr}(\sigma'_1)] \rightarrow^* [\text{chr}(\sigma')]$ and $[\text{chr}(\sigma'_2)] \rightarrow^* [\text{chr}(\sigma')]$ and $[\text{chr}(\sigma'_1)] \approx^A [\text{chr}(\sigma'_2)] \approx^A [\text{chr}(\sigma')]$. 


This is analogous. We just have to construct the ACT-R state from the joined CHR state $[\rho']$. Since $A([\rho'])$ holds by Lemma 28, this state exists.

In the following, the confluence test is applied to a classical example of a cognitive model.

**Example 50 (Confluence of Counting Model).** Reconsider the classical counting model from Example 18. It consists of the following production rule:

$$
\begin{aligned}
&=(\text{goal}, g, \{(\text{current}, X)\}), \\
&=(\text{retrieval}, \text{succ}, \{(\text{number}, X), (\text{successor}, Y)\}) \\
\Rightarrow
&=(\text{goal}, g, \{(\text{current}, Y)\}), \\
&+(\text{retrieval}, \text{succ}, \{(\text{number}, Y)\}).
\end{aligned}
$$

Recall the semantics of the rule: The left-hand side tests if there is a chunk of type $g$ in the goal buffer. The value of its current slot is bound to variable $X$ by the matching. The second buffer test checks the retrieval buffer for a chunk of type order that has $X$ in its number slot. The value of the second slot is bound to variable $Y$. The right-hand side modifies the chunk in the goal buffer such that $Y$ is written to the current slot. The second action requests the retrieval buffer for a succ chunk that has $Y$ in its number slot.

The CHR translation of the rule is $H \Leftrightarrow G | B$, where

$$
\begin{aligned}
H &:= \{ \delta(D), \gamma(g, C_g, 0), \gamma(\text{retrieval}, C_r, 0) \}, \\
G &:= \text{chunk}(C_g, g, \{(\text{current}, X)\}) \text{ in } D \land \text{chunk}(C_r, \text{order}, \{(\text{number}, X), (\text{successor}, Y)\}) \text{ in } D.
\end{aligned}
$$

We assume that in the declarative memory each number chunk only appears in at most one order chunk at number or successor position. This means that the model has learned a stable order on the numbers and hence requests to the declarative module are deterministic. This is an assumption on the constraint theory of the declarative model that we assume for the sake of this model.

It is clear that this example model terminates for finite declarative memories. Therefore, we can apply our confluence criterion. We start by proving the $\alpha$-property.

The rule overlaps with itself, e.g. $(\delta(D), \delta(D'), \ldots; \ldots)$. This state invalidates invariant $A_1$ and hence is not part of the confluence test. Another overlap is $(\delta(D), \gamma(g, C_g, 0), \gamma(g, C_g', 0), \ldots; \ldots)$. It violates invariant $A_2$, because it has two gamma constraints for the same buffer.

There is a set of overlaps of the following form:

$$(\delta(D) \cup G; \text{chunk}(C_g, g, \{(\text{current}, X)\}) \text{ in } D \\
\land \text{chunk}(C_g, g, \{(\text{current}, X')\}) \text{ in } D \land B; \{D, X, X'\} \cup V).$$

Since the invariant $A$ holds in the state, it follows from $A_3$ that $X \equiv X'$, because otherwise there are two different chunk terms in the same chunk.
store with the same chunk identifier. Note that this is not an extension of the state, since the invariant holds already in the state for the corresponding chunk stores D. For the D, where it does not hold, there is no extension to reestablish the invariant. Hence, we do not have to extend the state, but just derive from the constraint theory that in all states where the invariant holds, it follows that

\[ CT \models \text{chunk}(C_g, g, \{\text{current}, X\}) \text{ in } D \]
\[ \land \text{chunk}(C_g, g, \{\text{current}, X'\}) \text{ in } D \rightarrow X \equiv X'. \]

The overlap then simplifies to:

\[ \langle \delta(D) \cup G; \text{chunk}(C_g, g, \{\text{current}, X\}) \text{ in } D \]
\[ \land X \equiv X' \land B; \{D, X, X'\} \cup V \rangle. \]

The remaining overlap state to consider is \( \langle H; G; V \rangle \) that only consists of the head and guard of the rule where \( V \) contains all variables of \( H \) and \( G \). It is joinable, because we assumed determinism of requests, i.e. there is only one possible result chunk for each request. It can be seen that all possible overlaps in this small example invalidate the ACT-R invariant \( A \) or are joinable. Therefore, the model consisting only of this one counting rule is confluent. If we would assume an agent that has not learned a stable order of numbers, yet, i.e. there are numbers with different successors, the model would not be confluent. The confluence test constructs minimal representations of the states that are not joinable, i.e. giving an insight to the reason why a model is not confluent. This allows to decide whether the model has the desired behavior when it comes to different available strategies.

For the \( \beta \)-property, Lemma 31 ensures that it is satisfied for all embedded states. Hence, the ACT-R model is confluent under the assumption that the declarative memory is unique. Note that this restriction would be revealed by our confluence test. I.e. that even if we decline this assumption, the reason for non-confluence is revealed by our confluence criterion.

10.4 Discussion

We have presented a decidable confluence criterion for ACT-R models using invariant-based confluence modulo equivalence for CHR. In this section, limitations and possible extensions of our approach are discussed (c.f. Sections 10.4.1 and 10.4.2). Furthermore, the relation of this approach to our published paper [GF17] is discussed in Section 10.4.4.

10.4.1 Isomorphic Chunk Networks

In ACT-R states, chunk stores are defined over a chunk identifier function. This is important, since in an ACT-R rule one can refer to those identifiers that play a role for matching. However, the actual identifiers are irrelevant for the semantics of most models, as long
as they are consistent and the chunk stores are isomorphic. During
the execution of an ACT-R model, new chunks can be created and
new identifiers are produced for them. The identifiers can vary from
implementation to implementation and even from run to run of the
model.

Depending on the definition of the identifier function \( id \), this can
be a problem for confluence: Consider a state where a rule \( r_1 \) can
be applied that produces a new chunk \( c_1 \). In the same state, a rule \( r_2 \) is
applicable that produces a chunk \( c_2 \).

There could be a constellation for a state \( \sigma \), that allows the following
transitions: \( \sigma \xrightarrow{r_1} \sigma_1 \xrightarrow{r_2} \sigma_1' \) and \( \sigma \xrightarrow{r_2} \sigma_2 \xrightarrow{r_1} \sigma_2' \). We assume that
there is a clean-up rule \( r \), that empties all buffers in the end, so the
cognitive state \( \gamma \) is equivalent: \( \sigma_1' \xrightarrow{r} \sigma_1'' \) and \( \sigma_2' \xrightarrow{r} \sigma_2'' \).

The both states have equivalent cognitive states and additional
information. Their chunk stores \( \Delta''_1 \) and \( \Delta''_2 \) both contain the chunks \( c_1 \) and \( c_2 \), because \( r_1 \) and \( r_2 \) have both been applied. However, \( \Delta''_1 \) and \( \Delta''_2 \)
might have different identifier functions that map different identifiers
to \( c_1 \) and \( c_2 \) such that \( id_{\Delta''_1}(c_1) \neq id_{\Delta''_2}(c_1) \) and \( id_{\Delta''_1}(c_2) \neq id_{\Delta''_2}(c_2) \).

The confluence test hence fails, although semantically, the states are
equivalent and should therefore be joinable.

This problem can be fixed by defining an equivalence relation \( \simeq \)
over ACT-R states that ignores chunk identifiers. With this equivalence
relation, the equivalence relation \( \cong^A \) over CHR states can be extended
to a combined equivalence relation \( \cong^A \) as follows: for two ACT-R
states \( \sigma, \sigma' \), it holds that \( \sigma \simeq^A \sigma' \) if and only if \( \text{chr}(\sigma) \simeq^A \text{chr}(\sigma') \). Then,
the confluence modulo \( \cong^A \) criterion over the translated CHR program
yields the desired result.

### 10.4.2 Module Requests

By its modular structure, the ACT-R theory offers the possibility to
extend the architecture by arbitrary modules for special cognitive tasks.
The procedural module can state requests to those modules that can
return results of arbitrarily complex computations. The computation
of those requests typically uses the additional information of the states.
Obviously, confluence heavily depends on the results of the requests
as soon as they are part of a rule, because they can produce arbitrary
results.

In CHR, requests are realized as built-in constraints. As those, they
have an underlying constraint theory. For each module, a constraint
theory has to be defined to improve reasoning about the results of
requests to those modules. For specific cognitive models, reasoning can
be improved by defining specific constraint theories that suit the data
structures and computations of the ACT-R models. For instance, in
Example 50 the constraint theory of the declarative module integrated

Confluence modulo an isomorphism equivalence relation can be used to
reestablish confluence in such models.

Module requests can return arbitrary results.

The constraint theories of the modules can integrate
domain-specific knowledge of the model.
Due to the properties of $A$ and $\approx^A$, the methods used in this chapter can be transferred to other analyses.

The work on invariant-based operational equivalence modulo equivalence is directly available for the ACT-R embedding.

This thesis lifts concepts that have been used informally in prior work on a formal basis.

10.4.3 Generality of Approach

Soundness and completeness of the embedding in Chapter 9 allow to use CHR analysis methods directly. However, as could be seen for the confluence criterion in this chapter, sometimes the analysis tools have to be adapted to match the application domain of cognitive modeling with ACT-R.

The ACT-R invariant $A$ and equivalence relation $\approx^A$ defined in this chapter can be reused directly for all invariant-based analyses modulo equivalence. The properties of the invariant and the equivalence relation allow for a broad use: The set of minimal extensions of $A$ has been shown to be finite or even empty for all states and $\approx^A$ is fully $\diamondsuit_V$-compatible. Furthermore, $A$ is maintained by $\approx^A$, i.e. the invariant and the equivalence relation are compatible. This should make them applicable in many different contexts.

For this reason and due to the close relation of invariant-based confluence modulo equivalence and invariant-based operational equivalence modulo equivalence, the results of this chapter directly imply an operational equivalence criterion for ACT-R.

10.4.4 Relation to Prior Work

In [GF17], only invariant-based confluence without user-defined equivalence relations has been used. The reason is that in this publication a CHR variant has been defined that allows states to contain sets as arguments which is typically excluded in regular state definitions and implementations. States containing such sets are considered equivalent in that paper, if they are state equivalent modulo the sets. However, in the published paper, this argumentation is only justified informally. The motivation was to simplify reasoning about the states: In the translation in Chapter 9, some (arbitrary) order over the chunks takes care of making the CHR representation of a chunk store unique. Since this order complicates reasoning about equivalence of states, the authors introduced the notion of sets as arguments of constraints, but only informally.

In this chapter of the thesis, the informal statements from [GF17] have been put on a formal basis. Although the work in this thesis uses confluence modulo equivalence which regularly requires to show the $\beta$-property in addition to the criterion presented in [GF17], the two tests eventually coincide. This is possible because of the structure of the translated ACT-R rules and the equivalence relation $\approx^A$: The translated
rules only use the \( \text{in}/2 \) constraints to check their applicability. The \( \text{in}/2 \) constraint coincides for sets and lists and therefore it can be shown that the \( \beta \)-property is always satisfied (c.f. Lemma 31). This formally supports the argumentation used in [GF17]. Furthermore the irrelevant order of chunks in the translation in Chapter 9 that was necessary to show exact soundness and completeness with respect to state equivalence is made obsolete for confluence (modulo the order of chunks). This allows to think about the translated chunk stores as actual sets on a formal basis.

10.5 RELATED WORK

The results on invariant-based confluence analysis have been used successfully to the embedding of graph transformation systems in CHR [Rai07; RF11].

In the context of ACT-R, there are – to the best of our knowledge – no other approaches that deal with confluence so far. There have been other approaches to formalize the architecture with the aim to reason about cognitive models. For instance, F-ACT-R [AW14b; AW14a] formalizes the architecture of ACT-R to simplify comparison of different models or to use model checking techniques. In [Sai+16] mathematical reformulations of ACT-R models are used for parameter optimization by mathematical optimization techniques.
Part IV

CONCLUSION AND FUTURE WORK
CONCLUSION

In this chapter, we summarize the results of the thesis and present ideas for future work. Pursuing the goal of introducing formal program analysis methods for the cognitive architecture ACT-R, we followed the idea of using CHR as a lingua franca of rule-based programming languages and formalisms.

Therefore, the first step was to extend the analysis techniques of CHR to make them suitable for more complex analyses. Then, the cognitive architecture has been formalized to allow for formal program analysis of ACT-R models. The semantics has been embedded to CHR by a source-to-source transformation. This allowed us to use our formal CHR analysis techniques to define a decidable confluence criterion for ACT-R. Our results can be directly applied to operational equivalence of ACT-R models.

11.1 PROGRAM ANALYSIS OF CONSTRAINT HANDLING RULES WITH USER-DEFINED EQUIVALENCE RELATIONS

First of all, we summarize our work on program analysis of CHR modulo user-defined equivalence relations. The results contain some general considerations about the implications of introducing user-defined equivalence relations to program analysis, an invariant-based confluence modulo equivalence result, and an invariant-based operational equivalence modulo equivalence result.

REASONING ABOUT USER-DEFINED EQUIVALENCE RELATIONS

We have shown that in CHR, reasoning with a meta-language that allows to constitute subsets of states by filtering with a first-order predicate, can be simulated by corresponding object language level states directly. This can be achieved by adding the first-order predicate to the built-in constraint store of the object language level state. This allows to reuse all analysis techniques available for CHR directly, by replacing abstract state descriptions on the meta-level with their object language level representations. Thereby, equivalence relation definitions over states as they are formulated in [CK17], can be expressed on the object language level.

Additionally, we have shown that reasoning over the ground, range-restricted fragment of CHR is possible by using the established methods of invariant-based confluence with the groundness invariant. We have also shown that the method of case splitting as proposed formally in [CK17] for the meta-level, can be transferred to object language rep-
The properties of equivalence relations that allow to exploit monotonicity have been defined.

A class of compatible equivalence relations has been defined.

Sufficient and Necessary Criterion for Invariant-Based Confluence Modulo Equivalence

Decidability of the Criterion

The results can be extended to invariant-based operational equivalence modulo equivalence.

Conclusion

The properties of equivalence relations that allow to exploit monotonicity have been defined. A class of compatible equivalence relations has been defined.

Sufficient and Necessary Criterion for Invariant-Based Confluence Modulo Equivalence

By applying those preliminary considerations, we have presented a sufficient and necessary criterion for invariant-based confluence modulo equivalence of terminating CHR programs. It is the first decision criterion for CHR in the domain of confluence modulo equivalence that is entirely based on the object language level and therefore complements a manifold ecosystem of existing analysis techniques for CHR. It is also the first decision criterion that takes the necessary restrictions on equivalence relations into account that have been informally assumed in other work on the topic [CK17].

For p-state equivalence relations, the criterion is decidable depending on the invariant. The restriction on the invariant originates from the invariant-based confluence criterion [Rai10, pp. 75–87; DSS07; DSS06].

Invariant-based operational equivalence modulo equivalence

The results are then transferred to a sufficient and necessary operational equivalence modulo equivalence result for programs that are terminating and confluent (modulo the equivalence relation) with respect to the invariant. To the best of our knowledge, it is the first criterion for operational equivalence modulo equivalence for CHR. In general, the criterion requires the equivalence relation to be fully \( \sigma_V \)-compatible. It is decidable for p-state equivalence relations.

11.2 Analysis of ACT-R Models

This section is a summary of our results for the analysis of ACT-R models. They contain an operational semantics of ACT-R, a formal embedding of ACT-R in CHR that is sound and complete with respect
to the semantics, and a decidable confluence result for ACT-R based on the embedding and our CHR analysis methods.

**Formal Semantics and Embedding** We have defined an operational semantics of ACT-R on two different levels of abstraction. The very abstract operational semantics serves as a common base for analysis of different operational semantics, since it leaves enough room for various ACT-R variants. We then have defined an abstract semantics as an instance of the very abstract semantics.

Similar to the very abstract semantics, the abstract semantics abstracts from details like timings, latencies, forgetting, learning and specific conflict resolution. However, it defines the matching of rules and the processing of actions as they are typically found in ACT-R implementations. Hence, the abstract semantics concentrates on an abstract version of the typical implementations of ACT-R’s procedural system. It has been proven that the refined semantics from [GF15b] that represents typical implementations is a sound instance of the abstract semantics. This makes it possible to reason about the general transitions that are possible in many ACT-R implementations. It makes the abstract semantics suitable for analysis of ACT-R models.

The proposed operational semantics and its instances are extensible by exchanging or extending the individual abstract components of the semantics as argued in Section 8.3. For instance, new modules can be represented by new allowed additional information with corresponding interpretation and conflict resolution can be introduced and adapted by replacing or extending the definition of the selection function.

We have proposed an embedding of ACT-R in CHR by a source-to-source transformation. The translation is sound and complete with respect to our abstract semantics of ACT-R. To the best of our knowledge, the abstract semantics together with the sound and complete translation to CHR is the first formal formulation of ACT-R that is also suitable for implementation.

**Confluence Test for ACT-R** We have presented a decidable confluence test for the abstract operational semantics of ACT-R. A confluence test can help to improve ACT-R models by identifying the rules that inhibit confluence. This enables the modeler to decide about the correct behavior of the model regarding competing strategies. In our approach, we use the sound and complete embedding of ACT-R in CHR to apply the invariant-based confluence modulo equivalence criterion for CHR to reason about ACT-R confluence, since standard CHR confluence is too strict.

For this purpose, we have defined the ACT-R invariant $\mathcal{A}$ on CHR states such that it is satisfied for all states that stem from a valid ACT-R state. We have proposed a decidable criterion for the ACT-R
invariant. Additionally, an equivalence relation over translated ACT-R states has been defined that allows to disregard the order of chunks in the chunk store imposed by the source-to-source transformation.

Then, the theoretical foundations for applicability of CHR invariant-based confluence modulo equivalence for the ACT-R invariant $\mathcal{A}$ and the equivalence relation $\approx^\mathcal{A}$ have been established. This leads to an invariant-based CHR $\mathcal{A}$-confluence test modulo $\approx^\mathcal{A}$.

Eventually, it is shown that $\mathcal{A}$-confluence modulo $\approx^\mathcal{A}$ of the embedded program coincides with ACT-R confluence. This makes our CHR approach applicable to decide ACT-R confluence. The criterion is decidable as long as the constraint theories behind the actions are decidable, because the invariant only leads to trivial minimal extensions and the equivalence relation is a $p$-state equivalence relation.

11.3 SUMMARY OF CONTRIBUTIONS

In summary, this thesis contains the following significant scientific contributions:

- A formalization of program analysis with user-defined equivalence relations for CHR using only object language level concepts. The latter simplifies reasoning as it does not introduce a new formulation of the semantics introducing a whole new meta-level. It furthermore allows to apply the well-established program analysis methods. The restriction to the object language level is formally justified.

- The identification of a subset of so-called $\diamondsuit_V$-compatible equivalence relations that do not violate monotonicity of CHR. This allows to base the corresponding program analysis methods on the well-established methods of CHR. The restriction to compatible equivalence relations is justified by the corresponding class of $p$-state equivalence relations that is non-trivial and contains many interesting equivalence relations.

- A sufficient and necessary confluence modulo equivalence criterion for terminating programs and compatible equivalence relations.

- A sufficient and necessary criterion for operational equivalence modulo $\approx$ of terminating and confluent modulo $\approx$ programs, where $\approx$ is a compatible equivalence relation.

- A formalization of the operational semantics of ACT-R in two levels of abstraction.

- A sound and complete embedding of ACT-R’s abstract semantics in CHR. This allows for the application of CHR program analysis to cognitive models.
11.4 Evaluation of the Approach

The summary of contributions justifies that the goal of developing a formal analysis framework for the cognitive architecture ACT-R has been achieved. Hence, the choice of employing CHR as an analysis framework has been proven successful.

The chosen approach demanded for defining a suitable invariant and equivalence relation over the CHR states, because the general notion of a CHR state does not fit the one of the embedded language. However, although this seems to complicate reasoning at first, the approach still has advantages over developing the techniques directly for ACT-R.

First of all, there are many analysis results for ACT-R and also successful examples of the approach, like e.g. for graph transformation systems [RF11; Rai10]. The results are applicable to other rule-based and declarative programming languages and formalisms that have a CHR embedding – including CHR itself.

But most importantly, the concepts behind the invariant and the equivalence relation are inherent to analysis of ACT-R models. For instance, the problems that are revealed by invalid overlaps in the CHR embedding and non-joinable pairs due to different orders of chunks in the chunk store, are actually the problems that will be observed in any confluence test for ACT-R.

In particular, the restrictions defined in the ACT-R invariant originate from the operational semantics. Overlapping ACT-R production rules directly instead of their translated versions, would just as well lead to invalid ACT-R states and one would have to consider, if those states can simply be ignored or if they have to be adapted in some way. The CHR embedding and the existing work on invariant-based confluence lifts those considerations on a formal basis and makes the problems explicit. I.e., the resulting confluence test would be similar to the test in the embedding.

However, instead of reproducing all necessary analysis tools for ACT-R, the embedding allows to access existing results from the CHR ecosystem. For instance, it is possible to apply important properties like monotonicity, that are not clear offhandedly for ACT-R.

Hence, by using the embedding it is possible to reuse many formal reasoning techniques and to make the new results applicable to many other formalisms and languages. Furthermore, the considerations necessary to develop such program analyses are made explicit due to the universality of the CHR analyses.
Our work leaves some open perspectives for future investigations. In this section, we summarize the ideas for future research for our two main parts of the thesis.

12.1 PROGRAM ANALYSIS OF CONSTRAINT HANDLING RULES WITH USER-DEFINED EQUIVALENCE RELATIONS

For our analysis methods proposed in Part ii, we suggest the following ideas for future work.

REASONING ABOUT USER-DEFINED EQUIVALENCE RELATIONS

As a continuation of our research on program analysis modulo user-defined equivalence relations, it can be investigated, how our work can be combined with the meta-level approach of Christiansen and Kirkeby to overcome the shortcomings of both approaches.

Furthermore, it would be interesting to investigate other classes of equivalence relations apart from $p$-state equivalence relations, that are fully or partially $\diamondsuit V$-compatible. Some promising candidates are equivalence relations that allow for different orders of arguments of constraints or summarize constraints that have the same value in one argument. In general, all equivalence relations that maintain the same number of user-defined constraints, i.e. that are non-zipping, seem to be promising candidates for interesting $\diamondsuit V$-compatible relations. Nevertheless, the class of zipping equivalence relations may also contain interesting representatives and could therefore be further investigated for its applicability in practical programs.

Finally, our work could be applied for other embeddings like term rewrite systems or GAMMA. Thereby, it is of interest, if the methods can be applied directly or have to be adapted to make them meaningful in the corresponding rule-based language or formalism.

INvariant-based confluence modulo equivalence

Confluence modulo equivalence can also be used to prove certain properties of a program. For example, in optimization algorithms an equivalence relation that considers different solutions as equivalent, if they have the same weight, can be used to show that every solution found by the optimization algorithm has the same weight and is therefore equally good. Future investigations could engage in suitable equivalence relations to prove other program properties with the help of confluence modulo equivalence.
An important result for CHR is a completion algorithm [Frü09, pp. 112–117], that generates rules for a non-confluent program such that it becomes confluent. Although completion modifies the semantics of a program, there are interesting examples where a program is extended by meaningful rules. However, the algorithm is not terminating in general. The completion algorithm could be adapted for confluence modulo equivalence.

**INVARIANT-BASED OPERATIONAL EQUIVALENCE MODULO EQUIVALENCE** In [Rai10, pp. 87–98], the invariant-based operational equivalence criterion is generalized to also accept non-confluent and non-terminating programs at the cost of decidability in general. We are confident that our criterion can be generalized similarly.

Furthermore, the approach could be generalized analogously to our confluence modulo equivalence criterion such that it can also be applied for equivalence relations that do not maintain the invariant. The idea is to again consider the set $M_{T,\approx}(\sigma, \sigma')$ for the rule states $\sigma$ and their equivalent states $\sigma' \approx \sigma$ instead of only $M \in (\sigma)$.

### 12.2 ANALYSIS OF ACT-R MODELS

Regarding the analysis of ACT-R models from Part iii, we suggest the following ideas for future work.

**FORMAL SEMANTICS AND EMBEDDING** The operational semantics of ACT-R could be extended by integrating theories about the sub-symbolic components of ACT-R to the constraint theories of the modules. For instance, there are equations for recall probabilities for the declarative module embracing noisy activation values [TLA06]. A suitable constraint theory of declarative knowledge could try to reason about the behavior of the declarative module including those findings on an abstract level. The power of CHR that allows for execution with incomplete states can then be used to integrate certain assumptions about ranges of parameters and reason about their implications.

To make predictions on the probability that a cognitive model has a certain result, the CHR extension CHRiSM [Sne+10a; SMV09] could be used. It enriches CHR rules with probabilities. Thereby, it supports probability computation and even an expectation-maximization learning algorithm that could be used for parameter learning of cognitive models.

**ACT-R CONFLUENCE** Reasoning about requests to modules that appear in a confluence proof can be extended by specific constraint theories on the modules that integrate domain-specific knowledge about the model. This idea can be extended by allowing for model-specific constraint theories. For instance, the integration of domain-
specific knowledge on chunk types in the context of a particular cognitive model could improve reasoning about module requests in such models. It could also be investigated, how the confluence test detects possibly unknown background knowledge or assumptions of a domain-specific model by unveiling the reasons for non-confluence.

The confluence test could be further relaxed with confluence modulo other equivalence relations. Thereby, the idea of proving program properties with confluence modulo equivalence can be applied to ACT-R models. For instance, an ACT-R model can be analyzed for the property that it always yields a certain class of chunks for the same input. More precisely, it could be of interest if a certain buffer always contains a chunk of a certain chunk type or with a certain value in some slot at the end of a computation. By that method, models could guarantee certain properties on their final states improving explanatory power and quality of cognitive models.

Another relaxation could be introduced by an equivalence relation that ignores chunk identifiers and therefore considers chunk stores equivalent, if they have isomorphic chunk networks. This can be important since different execution paths of a model might yield isomorphic chunk stores with differing names because ACT-R may introduce new chunk identifiers during execution. However, in many models the semantics of such isomorphic chunk stores is equivalent and therefore confluence and operational equivalence should involve chunk store isomorphism. This can be accomplished by extending the ACT-R equivalence relation accordingly.


Hariolf Betz, Frank Raiser, and Thom Frühwirth. “A complete and terminating execution model for Constraint Handling Rules.” In: *Theory and Practice of Logic Programming* 10 (Special Issue 4-6 July 2010), pp. 597–610. ISSN: 1475-3081. DOI: 10.1017/S147106841000030X.


