Graphical Communicating Shared Resources: A Language for the Specification, Refinement, and Analysis of Real-Time Systems

Hanene Ben-Abdallah
University of Pennsylvania

Follow this and additional works at: http://repository.upenn.edu/ircs_reports

http://repository.upenn.edu/ircs_reports/98


This paper is posted at ScholarlyCommons. http://repository.upenn.edu/ircs_reports/98
For more information, please contact libraryrepository@pobox.upenn.edu.
Graphical Communicating Shared Resources: A Language for the Specification, Refinement, and Analysis of Real-Time Systems

Abstract
The Communicating Shared Resources (CSR) paradigm is an ongoing project at the University of Pennsylvania to build a framework for the development of real-time systems. This project has been motivated by a demand for a rigorous framework in which various design alternatives for a real-time system can be formally specified and rigorously analyzed and tested before implementation. This is an effort to reduce the potentially high cost associated with incorrect operation of real-time systems which are often embedded in safety-critical applications.

The work presented in this thesis is a first step towards incorporating software engineering practices into the CSR paradigm. This is achieved, on one hand, by developing a formal, graphical CSR formalism, the Graphical Communicating Shared Resources (GCSR); the GCSR language adopts the intuitive concepts of nodes and edges in state diagrams, an informal specification language that is popular within the software engineering community. In addition, defining a refinement theory for GCSR allows the development of real-time systems within this formalism in a top-down and modular fashion, also a popular design methodology within the software engineering community.

The GCSR language adopts a syntax that allows a modular and hierarchical, thus, scalable description of a real-time system. It supports notions of communication through events, interrupt, concurrency, and time to describe the functional and temporal requirements of a real-time system. In addition, GCSR allows the explicit representation of resources and priorities to resolve resource contention, in such a way that produces easy to understand and modify specifications. The semantics of GCSR is defined operationally either through a direct translation of a GCSR description to a labeled transition system, or indirectly through a sound translation to the Algebra of Communicating Shared Resources (ACSR) [LBGG94] a timed process algebra that also has an operational semantics. The GCSR-ACSR correspondence makes GCSR benefit from process algebraic analysis techniques such as equivalence checking, state space exploration, testing as well as simulation. In addition, the tight correspondence between GCSR and ACSR makes it possible to use the graphical and textual notations interchangeably and to have a sound theory for graphical transformation operations, e.g., to minimize the number of edges and nodes in a GCSR specification without affecting the behavioral description.

To support the top-down and modular development of a real-time specification in GCSR, we have augmented ACSR and thus GCSR with a refinement theory. The refinement theory allows relabeling of events, addition of implementation events, and substitution of a time and resource-consuming action with a process that may use fewer or more resources than the refined action. Consistency between an abstract specification and a refined specification is defined in terms of an ordering relation over traces that is extended to sets of traces according to the Hoare ordering or Egli-Milner ordering. The trace ordering relation relates traces that share timing properties such as equal duration and preservation of timed occurrences of communication events of the abstract specification. To facilitate the practical use of the refinement theory, we have characterized the extended trace ordering relations by a set of transformation rules that syntactically derive a refined process from an abstract one. The transformation rules define basic graphical operations that represent GCSR refinements.

This thesis or dissertation is available at ScholarlyCommons: http://repository.upenn.edu/ircs_reports/98
To experiment with the GCSR language and its refinement theory, we have developed a tool set that allows the specification, refinement, and analysis of real-time systems modeled in GCSR. We report our evaluation in the case of the Production Cell case study [LL95].

**Comments**
Graphical Communicating Shared Resources: 
A Language for the Specification, Refinement, 
and Analysis of Real-Time Systems 
Ph.D. Dissertation 

Hanene Ben Abdallah

University of Pennsylvania 
3401 Walnut Street, Suite 400A 
Philadelphia, PA 19104-6228 
October 1996 

Site of the NSF Science and Technology Center for 
Research in Cognitive Science
GRAPHICAL COMMUNICATING SHARED RESOURCES: A LANGUAGE FOR THE SPECIFICATION, REFINEMENT AND ANALYSIS OF REAL-TIME SYSTEMS

HANENE BEN ABDALLAH

A DISSERTATION

in

COMPUTER AND INFORMATION SCIENCE

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy.

August 1996

Insup Lee
Supervisor of Dissertation

Susan B. Davidson
Supervisor of Dissertation

Peter Buneman
Graduate Group Chairperson
Je dédie cet ouvrage à mes parents, Hédi et Walida, pour leurs amour, prières et encouragements.
ACKNOWLEDGEMENTS

I would not have made it through the PhD journey without the help of several special people, across cultural and geographical boundaries. I owe them much and can only try to express my gratitude.

I am deeply indebted to my advisors and friends Insup Lee and Susan B. Davidson. I would not have realized my dream without the guidance, unending patience, constant support and encouragement they have provided over the years.

I am thankful to my committee members: Norm Badler, Peter Buneman, Carl Gunter, and Lalita Jategaonkar Jagadeesan. Special thanks go to Peter Buneman for suggesting to investigate the efficiency of the translations between the graphical and textual languages. Special thanks are also due to Lalita Jategaonkar Jagadeesan for her feedback and help with the refinement theory.

I would like to acknowledge the pleasant and fruitful discussions with several colleagues in the Real-Time Systems Group at Penn: Jin-Young Choi, Duncan E. Clarke, Inhye Kang, and Hong-Liang Xie. I am particularly thankful to Duncan E. Clarke for his friendship, discussions of the socio-political American life, and for all the “sanity-keeping” lunch breaks at the bagel shop. I am also grateful to my officemate Inhye Kang for her friendship, encouragement, and all the hours of patient listening to my complaints when the wagon was not moving.

I also wish to thank Oleg Sokolsky for his help with the design of the GCSR tool set.

The research presented in this thesis was supported by NSF CCR-9415346 and AFOSR F49620-95-1-0508 grants, and a fellowship from the Institute for Research in Cognitive Science at the University of Pennsylvania.

Finally, my thanks go to my family and in particular to my mother for her love, patience and the beautiful sweaters she knitted to keep me warm, and to my father for the weekly telephonic rendez-vous he kept throughout the journey. I cannot put in words how fortunate and proud I feel to have you as parents and more importantly as friends.
ABSTRACT

Graphical Communicating Shared Resources:
A Language for Specifying, Refining and Analyzing Real-Time Systems

Hanéne Ben Abdallah
Advisors: Insup Lee and Susan B. Davidson

The Communicating Shared Resources (CSR) paradigm is an ongoing project at the University of Pennsylvania to build a framework for the development of real-time systems. This project has been motivated by a demand for a rigorous framework in which various design alternatives for a real-time system can be formally specified and rigorously analyzed and tested before implementation. This is an effort to reduce the potentially high cost associated with incorrect operation of real-time systems which are often embedded in safety-critical applications.

The work presented in this thesis is a first step towards incorporating software engineering practices into the CSR paradigm. This is achieved, on one hand, by developing a formal, graphical CSR formalism, the Graphical Communicating Shared Resources (GCSR); the GCSR language adopts the intuitive concepts of nodes and edges in state diagrams, an informal specification language that is popular within the software engineering community. In addition, defining a refinement theory for GCSR allows the development of real-time systems within this formalism in a top-down and modular fashion, also a popular design methodology within the software engineering community.

The GCSR language adopts a syntax that allows a modular and hierarchical, thus, scalable description of a real-time system. It supports notions of communication through events, interrupt, concurrency, and time to describe the functional and temporal requirements of a real-time system. In addition, GCSR allows the explicit representation of resources and priorities to resolve resource contention, in such a way that produces easy to understand and modify specifications. The semantics of GCSR is defined operationally either through a direct translation of a GCSR description to a labeled transition system, or indirectly through a sound translation to the Algebra of Communicating Shared Resources (ACSR) [LBGG94], a timed process algebra that also has an operational semantics. The GCSR-ACSR correspondence makes GCSR benefit from process algebraic analysis techniques such as equivalence checking, state space exploration, testing as well as simulation.
In addition, the tight correspondence between GCSR and ACSR makes it possible to use the graphical and textual notations interchangeably and to have a sound theory for graphical transformation operations, e.g., to minimize the number of edges and nodes in a GCSR specification without affecting the behavioral description.

To support the top-down and modular development of a real-time specification in GCSR, we have augmented ACSR and thus GCSR with a refinement theory. The refinement theory allows relabeling of events, addition of implementation events, and substitution of a time and resource-consuming action with a process that may use fewer or more resources than the refined action. Consistency between an abstract specification and a refined specification is defined in terms of an ordering relation over traces that is extended to sets of traces according to the Hoare ordering or Egli-Milner ordering. The trace ordering relation relates traces that share timing properties such as equal duration and preservation of timed occurrences of communication events of the abstract specification. To facilitate the practical use of the refinement theory, we have characterized the extended trace ordering relations by a set of transformation rules that syntactically derive a refined process from an abstract one. The transformation rules define basic graphical operations that represent GCSR refinements.

To experiment with the GCSR language and its refinement theory, we have developed a tool set that allows the specification, refinement, and analysis of real-time systems modeled in GCSR. We report our evaluation in the case of the Production Cell case study [LL95].
# Contents

1 Introduction .................................................. 1  
   1.1 Contribution ........................................... 3  
   1.2 Thesis Organization ................................... 6  

2 Related Work .................................................. 8  
   2.1 Graphical Formalisms for Real-Time Systems .......... 10  
      2.1.1 Communicating Real-time State Machines .......... 10  
      2.1.2 Statecharts ....................................... 10  
      2.1.3 Modechart ......................................... 12  
      2.1.4 Hierarchical Multi-State Machines ............... 13  
      2.1.5 Discussion ........................................ 15  
   2.2 Refinement Theory ....................................... 15  
      2.2.1 Refinement at the Syntactic Level ............... 16  
      2.2.2 Refinement at the Semantic Level ............... 19  
      2.2.3 Discussion ........................................ 20  
   2.3 Summary ................................................ 21  

3 The GCSR Language ............................................. 22  
   3.1 An overview ............................................. 23  
      3.1.1 Example ........................................... 23  
      3.1.2 GCSR Nodes and Edges ............................ 24  
      3.1.3 Informal Semantics ............................ 26  
   3.2 The Syntax of GCSR ..................................... 28  
   3.3 The Operational Semantics of GCSR ..................... 31  
      3.3.1 Alphabet ......................................... 33  

3.3.2 States .................................................. 33
3.3.3 Transitions ............................................. 35
3.3.4 Prioritized Operational Semantics ...................... 43
3.4 The GCSR-ACSR Correspondence ............................ 44
  3.4.1 ACSR ..................................................... 44
  3.4.2 From GCSR to ACSR .................................... 46
  3.4.3 From ACSR to GCSR .................................... 49
  3.4.4 Efficiency of the Translations: Tying it Together .... 51
3.5 Busy Railroad Crossing at an Intersection ................. 55
  3.5.1 Safe Design ............................................. 59
  3.5.2 A Faulty Train .......................................... 61
  3.5.3 A Faulty Gate .......................................... 61
3.6 Summary .................................................... 63

4 Refinement in ACSR 64
  4.1 Motivation ............................................... 65
  4.2 Trace Refinement ......................................... 68
  4.3 ACSR Process Refinement in the Hoare Ordering ...... 73
    4.3.1 Compositionality ..................................... 74
    4.3.2 Refinement Transformation Rules ................... 79
    4.3.3 Process Refinement in the Egli-Milner Ordering ... 85
  4.4 Refinement in the Prioritized Semantics ................. 87
  4.5 Example: Task System .................................... 90
  4.6 Discussion ................................................ 94
    4.6.1 Advantages of Our Approach ......................... 94
    4.6.2 Limitations of Our Approach ....................... 96

5 The GCSR Tool Set 100
  5.1 The VERSA System ....................................... 101
  5.2 The GCSR Tool Set ....................................... 102
    5.2.1 Language ............................................. 102
    5.2.2 Views ................................................. 103
    5.2.3 Operations ........................................... 106
  5.3 Refinement in GCSR ...................................... 110
6 Production Cell: a Case Study 116
   6.1 Informal Description ........................................ 117
   6.2 Design Strategy .............................................. 119
   6.3 Design Solution ............................................... 121
      6.3.1 System Structure .................................... 123
      6.3.2 Deposit Belt ...................................... 123
      6.3.3 Traveling Crane .................................... 126
      6.3.4 Feed Belt ...................................... 131
   6.4 Design Solution Analysis .................................. 133
      6.4.1 Safety analysis: Part 1 ......................... 134
      6.4.2 Safety analysis: Part 2 ......................... 134
      6.4.3 Liveness Requirements ......................... 137
   6.5 Evaluation .................................................. 140
      6.5.1 Modeling ........................................ 140
      6.5.2 Refinement .................................... 142
      6.5.3 Analysis ........................................ 144
      6.5.4 GCSR/VERSA Tool Sets ......................... 144

7 Conclusion 145

A Selected Proofs for Chapter 3 150
   A.1 Proof of Theorem 3.4.1 .................................. 150
   A.2 Proof of Theorem 3.4.5 .................................. 152

B Selected Proofs for Chapter 4 154
   B.1 Proof of Theorem 4.3.1 .................................. 154
   B.2 Proof of Lemma 4.3.1 .................................. 159
   B.3 Proof of Theorem 4.3.2 .................................. 161
   B.4 Proof of Theorem 4.3.3 .................................. 162
   B.5 Compositionality in the Smyth Ordering .............. 167

C Production Cell Details 169
   C.1 Design Assumptions ...................................... 169
List of Figures

3.1 Gate example in GCSR .................................................... 24
3.2 GCSR nodes ................................................................. 25
3.3 GCSR edges ................................................................. 26
3.4 GCSR samples of partial specifications ............................... 27
3.5 Short-hand notation in GCSR ......................................... 31
3.6 Two equivalent GCSR processes ...................................... 33
3.7 GCSR to ACSR translation, $T_{GA}$ .................................. 47
3.8 Special cases of Step 5 of the GCSR to ACSR translation ....... 48
3.9 ACSR to GCSR translation, $T_{AG}$ .................................. 50
3.10 GCSR transformations, $T_{GG}$ (part 1 of 2) ....................... 52
3.11 Illegal GCSR transformation ......................................... 52
3.12 GCSR transformations, $T_{GG}$ (part 2) ............................ 53
3.13 Top view of the Railroad Crossing example ...................... 55
3.14 High level GCSR specification of the railroad crossing .......... 57
3.15 GCSR specification of a car .......................................... 58
3.16 GCSR specification of a bus ......................................... 58
3.17 The traffic light in GCSR ............................................. 59
3.18 The gate in GCSR ...................................................... 59
3.19 The Train in GCSR .................................................... 60
3.20 A faulty train in GCSR ................................................ 61
3.21 A faulty gate in GCSR ................................................. 62
5.1 The GCSR/VERSA tool sets .......................................... 100
5.2 Example in the GCSR storage language ............................ 104
5.3 A session of the GCSR tool set ...................................... 105
5.4 Drawing a time-consuming node ........................................ 107

6.1 Top view of the production cell system ............................... 118
6.2 Component dependencies in the production cell .................... 122
6.3 Deposit belt at levels 1 (a), 2 (b) and 3 (c) ......................... 124
6.4 Traveling crane at levels 1 (a), 2 (b) and 3 (c) .................... 126
6.5 Controller for the traveling crane at level 1 ......................... 127
6.6 Controller for the traveling crane at level 3 ........................ 128
6.7 Model for the traveling crane at levels 3 (a) and 4 (b) ............. 130
6.8 First attempt to the design of the crane ............................. 131
6.9 Feed belt at levels 1 (a), 2 (b) and 3 (c) .......................... 132
6.10 Controller and model for the feed belt at level 3 ................. 133

B.1 Possible trace transitions of $P$ ....................................... 164

C.1 Controller for the elevating rotary table ......................... 173
C.2 Components of a model for the elevating rotary table .......... 174
C.3 Robot controller (part 1 of 2) ....................................... 175
C.4 Robot controller (part 2 of 2) ....................................... 176
C.5 Robot model (part 1 of 2) ........................................... 177
C.6 Robot model (part 2 of 2) ........................................... 178
C.7 Press controller ...................................................... 179
C.8 Press model ......................................................... 180
Chapter 1

Introduction

The Communicating Shared Resources (CSR) paradigm is an ongoing project at the University of Pennsylvania to build a framework for the development of real-time systems. This project has been motivated by a demand for a rigorous framework in which various design alternatives for a real-time system can be formally specified and rigorously analyzed and tested before implementation. This is an effort to reduce the potentially high cost associated with incorrect operation of real-time systems which are often embedded in safety-critical applications.

The CSR paradigm is based on the premise that the timed behavior of a real-time system is affected not only by the time its components take to execute and synchronize, but also by delays introduced due to the scheduling of tasks that compete for shared resources. One of the objectives of the CSR paradigm is therefore to provide a formalism where the run-time resource requirements of a real-time system can be specified together with its functional requirements. The integration of the two types of requirements allows designers to consider resource-induced constraints during the design stage of the development cycle and to eliminate unimplementable design alternatives without expensive prototyping.

The work presented in this thesis is a first step towards supporting software engineering practices within the CSR paradigm. More specifically, it tries to incorporate within a CSR formalism both popular notations in languages used to model complex systems and incremental development methodologies.

**Graphical representation.** Control flow and data flow diagrams have been popular specification languages widely used in software engineering through methodologies such as
Structured Analysis [DeM87, WM85, HP87]. These languages however lack formal semantics and thus are not reliable for safety critical systems. We have developed a graphical language for real-time systems called Graphical Communicating Shared Resources, GCSR. The GCSR language adopts the intuitive graphical concepts of nodes and edges in control flow diagrams. In addition, the syntax of GCSR was carefully designed so that it produces modular, hierarchical, thus, scalable specifications that are also easy to modify—for example, to reflect different run-time resource requirements. Being a CSR formalism, GCSR supports the explicit representation of resources and priorities within a formal semantics. It also offers analysis techniques to allow a designer to have more confidence in a design solution.

**Theory to support incremental development.** Top-down development has been a prominent design strategy for complex systems. It facilitates the specification and understanding of large scale systems. We have developed a refinement theory to make GCSR suitable for the top-down and modular specification of real-time systems.

Our work benefits from two areas of research. One is the work done within the CSR paradigm that produced the process algebra ACSR [LBGG94], together with the pioneering work on graphical formalisms for real-time systems which is done in Communicating Real-time State Machines [Sha92], Statecharts [Har87, HPSS87], Modechart [Jah88, JM89], and Hierarchical Multi-State Machines [Gab91, GF91, GF88]. ACSR laid out the formal grounds for GCSR by its formal treatment of priorities and resources. In addition, ACSR complements the GCSR language with process algebraic techniques for verification through equivalence checking, state space exploration, and testing [LBAC96]. Furthermore, the ACSR tool set VERSA (Verification, Execution, and Rewrite System for ACSR) [CLX95b] facilitated an experimental evaluation of the GCSR language.

The above mentioned graphical languages, on the other hand, helped us to understand necessary features in a graphical language, such as nesting, to produce scalable specifications. In addition to supporting notions of resources and priorities, GCSR is distinguished from these languages by its notions of structured modular and hierarchical specification: a specification where dependencies among system components, which are expressed as communication events, can be limited, and one where edges do not cross containing node boundaries.

The second area of research from which our work has benefited is refinement theories
based on relabeling [Hoa85, Mil89] and action refinement [Ace92, AM93, CvGG93, Jat93, NEL88, vGG89], which have been developed mainly for non-real-time formalisms—such as CCS [Mil89], Petri nets [Jat93, vGG89], and labeled transition systems [GMM88, Jif89].

Our notion of refinement adopts a dual approach where both types of refinement are combined. This dual approach to refinement was motivated by two observations. First, while relabeling preserves the "granularity" of refined actions, action refinement suggests that a specification action has a coarser granularity than an action in the refined specification. Secondly, unlike other models where actions uniformly denote one type of activity (mainly communication), GCSR (and ACSR) divides actions into instantaneous actions, called events, and time consuming actions, called actions. Since events are instantaneous at all levels of abstraction, it seems natural to use relabeling to refine them. On the other hand, action refinement seems a natural refinement concept for actions, since they are time-consuming activities and hence can be split to describe a finer granularity.

### 1.1 Contribution

The main contributions of our work are three-fold: 1) design a graphical, formal language within the CSR paradigm; 2) develop a refinement theory that allows incremental design within the language; and 3) experimentally evaluate the advantages and limitations of the language and its refinement theory.

#### The GCSR Language.

As mentioned earlier, the GCSR language adopts two essential features that have been developed within the CSR paradigm: the explicit representation of resources and priorities and the formal treatment of these concepts. In addition, we have carefully designed the syntax of GCSR to provide for a structured modular, hierarchical, and scalable description of real-time systems. The GCSR syntax is relatively simple and reminiscent of control flow diagrams, which would facilitate its transfer to the software engineering community.

Furthermore, we define the formal semantics of GCSR operationally as a labeled transition system. This allows the execution of a GCSR specification to inspect sample behaviors for a better understanding of the specification. We provide two ways to construct the labeled transition system of a GCSR specification. One way is directly from the GCSR specification; this way lays the grounds for the development of a visual simulator for the
language, an essential component of a design environment. The second way to derive the labeled transition system of a GCSR specification is indirectly through a well-defined mapping between GCSR and ACSR which also has an operational semantics. The labeled transition systems of a GCSR specification obtained directly or via ACSR are tightly equivalent in a sense described precisely in Chapter 3.

The GCSR-ACSR correspondence has several advantages. One is that it makes a well-founded algebraic formalism, namely ACSR, more accessible to software engineers. Another advantage is that it permits the interchangeable use of the graphical and textual notions; for example, to describe the structure of a system graphically and fill in the details textually. In addition, the GCSR-ACSR correspondence makes it possible for GCSR to benefit from the analysis techniques and system offered by ACSR. One particularly useful analysis technique is equivalence checking through the various equivalence relations in ACSR. These relations can be used to restructure a GCSR specification to a graphically more succinct, e.g., fewer edges and nodes, yet, equivalent GCSR specification. This distinguishes GCSR from other graphical specification languages, e.g., [Har87, JM89, Sha92, Gab91], which have this facility.

Refinement Theory. To make GCSR suitable for the top-down and modular development of real-time systems, we expanded ACSR and thus GCSR with a refinement theory. With this theory, we envision the design of a complex system to begin with a GCSR specification that describes the behavior of the object system at a certain level of abstraction. At this stage, the system’s resource requirements may not be exactly known and are therefore estimated; for example, at this stage, a computation action is specified as requiring a cpu and memory simultaneously. Various, subsequent stages of the design process gradually describe the system at more detailed levels towards a specification that is appropriate to permit the system implementation. These stages may use different notation, e.g., different event names, describe subactions of an abstract action, e.g., the computation action consists of a calculation action followed by a result storing action, and they may tighten the resource requirements, e.g., the cpu is used at the beginning of the computation action after which memory is accessed to store results.

Refinement in ACSR therefore allows an implementation to relabel the specification events, introduce new events, show the details of an abstract time consuming action, and reduce or increase the resource requirements in the specification. It is supported through
two ordering relations over process terms whose semantics are defined by extending a preorder relation over traces. The extensions reflect two common notions of “implementation” and are based on ordering relations in the powerdomain theory [Smy78, Gun92]: the Hoare ordering and Egli-Milner ordering.

One important property of the trace preorder relation is that it preserves the timed occurrences of the specification events. This in turn makes it possible for an implementation to inherit the specification’s properties such as safety properties. In addition, two desirable properties of the extended trace relations are that they be *compositional* with the ACSR operators which allows the modular application of refinement, and that they be represented syntactically through a set of operators. The second property has practical motivations and allows the refinement theory to provide both semantic support for a hierarchical design methodology as well as tools to derive implementations from specifications incrementally within a design environment.

In the unprioritized semantics, the extended trace relations are, under reasonable assumptions, compositional with all of the ACSR operators except for the Parallel operator; compositionality with the Parallel operator is limited due to resource sharing. In addition, each extension can be characterized by a set of syntactic transformation rules that incrementally rewrite a specification into an implementation. The transformation rules use the basic operators of event relabeling, event addition, and action refinement. They use a notion of action refinement that is relational as they allow two occurrences of an action to be refined differently. This gives the designer more flexibility especially that an action in the CSR paradigm is an abstract representation of resource usage. In the Hoare ordering, the transformation rules augmented with the axioms of strong bisimulation [BGCL93] are complete for finite processes. In the Egli-Milner ordering, the transformation rules are complete modulo trace equivalence. In the prioritized semantics, compositionality is very limited. This is due to the fact that the trace refinement does not incorporate any notion of preemption.

**Experimental Evaluation.** To examine the usefulness and limitations of the GCSR formalism, we have developed a prototype tool set that allows the modeling, refinement and analysis of real-time systems in GCSR. The GCSR tool set is based on a graphical user interface that allows the drawing and manipulation of GCSR specifications. It implements the refinement transformation rules through graphical operations. In addition, it supports
an automated translation of GCSR specifications to ACSR processes, which allows the user to conduct analysis within the VERSA system.

We have used the GCSR/VERSA tool sets to specify, refine and analyze the production cell case study [LL95]. This case study confirmed that the graphical syntax of GCSR helps to visualize the overall structure of the system, its component dependencies, and the flow of control within each component. The notions of resources and priorities along with the temporal constructs in GCSR (and ACSR) are very expressive and natural to use. In particular, the explicit representation of resources made the formulation and verification of several safety requirements straightforward. Overall, the formal semantics and the process algebraic analysis techniques are suitable for real-time systems in the class of the production cell application.

During the design of the production cell, we relied on the transformation rules for refinement in the Hoare ordering. This ordering was sufficient as we were interested in preserving safety requirements that are expressed in terms of the timed occurrences of events. In this case study, we also noticed that a judicious application of refinement steps can overcome the limited compositionality of refinement in the prioritized semantics. Furthermore, a lesson we learned through the case study is that providing tools for deriving design solutions incrementally is not sufficient to speed the design phase. To avoid an ad hoc design process, it is necessary to develop a structured methodology that guides the application of these tools according to what is considered as essential behavior at each level of abstraction.

1.2 Thesis Organization

The remainder of this thesis is organized as follows. Chapter 2 is composed of two parts: the first reviews related work in the area of graphical formalisms for real-time systems; the second reviews relevant theories to support system specifications at different levels of abstraction.

Chapter 3 presents the GCSR formalism, its syntax, formal semantics as well as its correspondence with ACSR. This chapter illustrates the GCSR formalism through an extended version of the railroad crossing example benchmark [HJJ93].

Chapter 4 augments ACSR with a refinement theory that allows top-down, modular specification in GCSR. This is done in three parts. First, a notion of trace refinement is
defined. Second, the trace refinement is extended to relate processes in the unprioritized semantics; compositionality and characterization of the extended relations are examined. Third, the effects of priorities on compositionality are described. This chapter illustrates the refinement theory and its properties through a router and a task system example.

Chapter 5 gives a brief tutorial of the GCSR tool set for modeling real-time systems in GCSR. It also connects the ACSR refinement to graphical operations for the refinement of GCSR specifications.

Chapter 6 evaluates and reports our experiments using GCSR for the specification, refinement and analysis of the production cell case study [LL95].

Chapter 7 summarizes our contributions and outlines future work.

Appendix A contains selected proofs of theorems and lemmas for Chapter 3. Appendix B contains selected proofs of results for Chapter 4. Appendix C includes more details about the production cell case study. Finally, to make this thesis self contained, Appendix D lists the operational semantics of the ACSR process algebra.
Chapter 2

Related Work

There are two important features in a design environment for complex systems: one is the specification language used to model the system behavior; another is the theory which allows incremental development of a system model at different levels of detail.

**Specification language.** Specification languages for real-time systems can be classified into informal or formal languages. Informal specification languages describe a system by using a combination of graphics and semiformal textual grammars. The wide use of informal specification languages is attributed to their ease of learning and their intuitive constructs and flexibility. Informal languages however have several disadvantages. One is that they may produce imprecise and ambiguous specifications. Another is human reasoning, which is error-prone, being the main means of analysis in these languages [Zav91]. Several informal specification languages extended popular techniques for the design of non-real-time systems—e.g., real-time extensions of Yourdon-DeMarco Structured Analysis [DeM87] such as those developed by Ward and Mellor [WM85] and Hatley and Pirbhai [HP87], and the object-oriented approach embodied in the Information Model described by Shlaer and Mellor [SM88].

Formal specification languages, on the other hand, describe a system as mathematical objects and use formal notation that helps to produce succinct and precise specifications. A main advantage of formal specification languages is their support of, possibly automated, analysis techniques that are based on mathematical operators and proof procedures to check the internal consistency of a specification. There has been a significant progress in the development of formal languages for real-time systems. Much of this work falls into
the traditional categories of temporal logics (e.g. [HLM88, RMSM+93]), assertional methods (e.g. the Vienna Development Method), net-based models (e.g. Petri Nets [Rei85]), automata theory (e.g. [MMT91]) and process algebras (e.g. [BG94, DS89, Yi91]).

The often complex semantics and cumbersome notations of formal techniques have, however, impeded their access by non-technical people [FV91, Zav91]. Recently, in an effort to make formal languages more user friendly, several formal specification languages are based on graphical notations that are adapted from graphical languages such as finite state diagrams and control flow diagrams, which have been popular in the software engineering community. Statecharts [Har87, HPSS87], Modechart [Jah88, JM89], Communicating Real-time State Machines [Sha92], and Hierarchical Multi-State machines [Gab91, GF91, GF88] are among the pioneering graphical, formal languages for real-time systems.

**Refinement Theory.** There is a significant amount of research done about theories to support top-down and bottom-up development methodologies within formal specification languages. Most of these theories have been developed for concurrent, untimed formalisms. This work can be divided into two approaches. One approach is based on extending a given formalism with an operator, *relabeling* or *action refinement*; this approach, which we call *syntactic* approach, uses the notion of equivalence or preorder in the original language as the semantics of the new operator. Another approach, which we call *semantic* approach, is based on developing a relation that maps behaviors of specifications at different levels of detail.

**Chapter organization.** The remainder of this chapter is divided into three parts. Section 2.1 reviews graphical, formal specification languages that have been developed for real-time systems. Section 2.2 reviews relevant theories that support specifications at different levels of detail. The last part, Section 2.3, motivates the need for a real-time formalism that combines several features, some of which are addressed in different research efforts.
2.1 Graphical Formalisms for Real-Time Systems

In this section, we review relevant, graphical, formal specification languages for real-time systems in terms of their expressiveness, representation, and analysis techniques they support.

2.1.1 Communicating Real-time State Machines

The Communicating Real-time State Machines (CRSMs) [Sha91, Sha92] extended the finite state diagrams with CSP [Hoa85] style of synchronous communication. The CRSM paradigm was designed to specify embedded real-time systems that are composed of concurrent, communicating components. Thus, a CRSM specification consists of at least two submachines: one to describe the required system behavior and a second to describe the behavior of the environment with which the system interacts. The submachines execute concurrently and communicate synchronously over unidirectional channels.

States in a CRSM are “places” where the system control remains until a transition out is enabled and taken. Transitions in a CRSM are labeled with guarded timed commands that represent synchronous send or receive, or setting of local variable. The time associated with an IO command represents when synchronization must instantaneously occur. The time associated with a computation command represents when the computation is instantaneously executed. A transition guard is a predicate testing values of local variables. When the guard of a transition becomes true, it is enabled and its command can be executed at the time associated with it after which control moves to the target state of the transition.

The formal semantics of CRSM is based on history of timed traces that are composed of pairs of a list of IO commands and the time when the commands in the list were completed. Because of their precise operational semantics, CRSMs are executable. An algorithm for simulating the execution of a CRSM specification is presented in [Sha91]. Besides simulation, the CRSM paradigm does not offer other analysis techniques.

2.1.2 Statecharts

Statecharts [Har87, HPSS87] is a specification language for reactive systems, that has gained popularity within the academic and industrial communities. Statecharts combine
a state-based formalism reminiscent of finite state machines with their graphical counterpart state transition diagrams. Statecharts added three main constructs to state transition diagrams: 1) instantaneous broadcasting communication through global variables; 2) modularity and hierarchy through state decomposition into substates; and 3) history remembrance through history states.

In Statecharts, a system is described by a control flow through states that can be nested in other states and connected by directed labeled edges. A state with nested substates describe concurrent execution or orthogonal execution where control can be in one substate at a time. Connected states describe sequential execution. Each edge is labeled with an enabling condition-action pair, \( e/a \). The enabling condition \( e \) is a predicate over state entering and exiting variables and values of other system variables. The action \( a \) can be either null, or a sequence of one or more primitive actions. A primitive action consists of setting a state or other system variable. When the enabling condition of a transition becomes true, the transition is taken by instantaneously executing its action and moving control to the target state of the transition.

The semantics of Statecharts is defined as timed sequences of changes that occur in the system. The time instants when the changes are collected correspond to the sampling rate of the system under development. Changes are represented in terms of maximal lists of taken transitions and the events generated when the transitions are taken. Since one taken transition may enable other transitions through its action, Statecharts semantics may lead to semantic anomalies due to conflicting chain reactions. Conflicting chain reactions can occur because shared variables can be set instantaneously on transitions and results are broadcast throughout the system, which therefore may lead to possible conflicting values; the reader is referred to [HPSS87] and [HiR91] for examples. More recently, the semantics of a subset of Statecharts have been defined using labeled transition systems [US94].

STATEMATE and ROOM are two design environments that have been developed based on Statecharts. STATEMATE [HLNP90] uses a subset of Statecharts to define the dynamic behavior of the system and uses step-by-step executions to analyze the control flow of system activities. ROOM [SGME92, SGW94] is another attempt to build a methodology for real-time system development which is based on a modified version of Statecharts. ROOM allows designers to describe the behavioral view of a real-time system in an object-oriented paradigm with a graphical interface.
2.1.3 Modechart

Modechart [JM89, Jah88] is a specification language for real-time systems that extended Statecharts with constructs to express timing constraints such as deadlines and alarms. It also limited the liberal transition labels of Statecharts to avoid semantic anomalies.

In Modechart, a real-time system is described through a set of modes (states in Statecharts). Modes describe actions or control information and partition the system's state space. They can be connected by labeled edges to represent sequential execution, and they may contain other modes to represent system components and concurrent execution. An edge in Modechart is labeled with an enabling condition that describes the condition under which control may move from the source mode to the target mode instantaneously. An enabling condition is a disjunction of conjunctions each of which describes a lower bound and upper bound restriction on when the transition may be taken after being enabled. The lower and upper bounds in a conjunction are expressed in terms of the timed occurrence of events; for example, the label $e + I_1 \leq t \leq e + I_2$ states that the transition is enabled if the event $e$ is produced at time $t$ and that the transition may occur at least $I_1$ time units and at most $I_2$ time units after $t$ [JLM88].

Similar to Statecharts, Modechart uses events as the means of broadcasting communication. Modechart classifies events into six categories: external event which is produced by the system environment, e.g., operator presses a button; start event which marks the beginning of an action; stop event which marks the end of an action; state variable transition event which marks state change; mode entry event which marks the entry to a mode; and mode exit event which marks exit from a mode.

The semantics of Modechart is defined either in terms of sets of traces of event occurrences, the Real-Time Logic (a first-order predicate logic) [JLM88], or more recently in terms of a labeled transition system [PMS95]. The Real-Time Logic uses discrete time and can express explicitly various timing constraints such as sporadic and periodic actions, ordering of events, distances among events, delays, and deadlines. The trace-based semantics of Modechart allows the execution of systems modeled in Modechart. A system execution represents a sequence of sets of events that occur at specific times. For a given sequence and a specific time instant $t$, the set represents the events that happened concurrently in the system at time $t$. 

12
Modechart does not have built-in constructs to explicitly express run-time resource requirements. However, some run-time resource requirements may be encoded in Modechart through the enabling conditions. For an example, the reader is referred to [CHLR93] where parts of a missile system along with the operating system are modeled in Modechart. In this example, the scheduler part of the operating system is encoded as a mode that executes in parallel along with the missile system application. The scheduler mode is itself a parallel mode where submodes represent schedulers of the processors in the system. The scheduler of a particular processor is described as a sequential mode where submodes represent idling actions or actions that describe application tasks acquiring and losing the processor. As a processor scheduler becomes more complex due to the large number of tasks allocated to it, this method of integrating run-time resource requirements may, however, produce specifications that are difficult to understand. This method also lacks modularity. For example, the scheduler specification may have to be totally restructured when more system software components, i.e., tasks are added with different priorities. It is more natural to think about priorities, which are associated with tasks, as attributes associated with the task’s modes. This however would require changing the semantics of Modechart to add the notion of prioritized modes.

STARTOR and MT are two toolsets that have been built based on Modechart. STARTOR [HL91] is an experimental toolset to build and prove properties about hard real-time software and is based on Modechart to describe the functional view of the software. MT [CHLR93, RPC94] is another toolset that has been developed for the specification, verification, and simulation of real-time systems using a subset of Modechart. In MT, one can write a Modechart specification and an RTL formula that describes system properties, such as safety properties, and automatically verify whether the specification satisfies the RTL formula.

2.1.4 Hierarchical Multi-State Machines

Hierarchical Multi-State machines (HMS) [GF88, Gab91, GF91] are yet another remodeling of finite state machines by adding some concepts from timed Petri nets [GMea89]. HMS machines are parallel, hierarchical automata where transitions are controlled by Temporal Interval Logic predicates. In an HMS machine, concurrency is described by allowing multiple states to be active and multiple transitions to occur simultaneously. Hierarchy in HMS is allowed by letting states be themselves HMS machines. Synchronization and
causal interactions, i.e., communication among events are expressed with conditions on deterministic and nondeterministic transitions. Timing constraints are explicitly described as interval delays on enabling conditions of transitions. Enabling conditions are Temporal Interval Logic predicates.

The HMS operational semantics borrows the Petri net concept of associating tokens with states. Informally, an HMS machine executes by firing all of its enabled deterministic transitions and a subset of its enabled nondeterministic transitions. A transition is enabled when its source state is marked with a token and its Temporal Interval Logic predicate is satisfied. When a transition is enabled, it may be fired by placing a token in its target state.

Another goal behind the design of the HMS model is to be a formalism where a real-time system can be specified at multiple levels of details. The notion of hierarchical (or multilevel) specification in HMS machines differs from the notion of hierarchy in top-down and bottom-up specification methodologies. In the HMS model, a higher-level specification imposes constraints on any lower-level specification. Furthermore, all levels remain part of the final specification. This is to be contrasted to the notion of hierarchy in for instance a top-down specification methodology where starting at one level of abstraction, other more refined specifications can be constructed and must be proven to be consistent with the first specification. The final result is one level of specification.

A multilevel specification is derived as follows. For a given basic, i.e., lowest level HMS machine, higher-level machines are HMS machines with time consuming transitions called “policy transitions”. Informally, a policy transition describes the desirable behavior of the lower-level machine [GF91]. It is labeled with three control parts: 1) beginning control which describes the enabling condition of the transition; 2) middle control which describes the condition for the transition to remain enabled; and 3) end control which describes the condition for the transition to end, and after which the target states become marked with tokens. The execution of an HMS machine at every level is determined by the execution of the HMS machine which is at the next lower level and which meets the requirements imposed by all higher-levels machines. This notion of consistency is satisfied by having each HMS machine direct the selection among enabled, nondeterministic transitions in the lower-level HMS machine.
2.1.5 Discussion

The pioneering work done in CRSM, Statecharts, Modechart, and HMS outlined necessary graphical constructs in a language for real-time systems. These languages support the specification of currency, communication and timing constraints in a modular and hierarchical way except for CRSM which only supports modularity. Similar to other textual real-time formalisms, the underlying semantics of these languages, however, lack constructs to specify explicitly the run-time resource requirements of the system. We next briefly review Communicating Shared Resources based formalisms which do support such constructs. These formalisms are however not based on graphical languages. Augmenting the CSR paradigm with a graphical language is one of the contributions of this work.

Within the CSR paradigm, two formal specification languages have been developed: the Calculus for Communicating Shared Resources, CCSR, [Ger91, GL92] and the Algebra of Communicating Shared Resources, ACSR, [BG94]. CCSR is based on a discrete time domain where each action takes one time unit. It has a priority-sensitive, operational semantics that is characterized by a set of laws [Ger91]. The set of laws can be used to perform algebraic verification to check the behavioral equivalence of given specifications. ACSR is another algebraic approach to the CSR paradigm. ACSR distinguishes between instantaneous communication and time and resource consuming actions. It can support both discrete [LBGG94] and dense time domains [BG94]. Furthermore, ACSR has several notions of equivalence based on strong and weak bisimulation. In the discrete version of ACSR, the notions of equivalence can be used either in a syntax-based analysis technique through a set of algebraic laws, or in a semantics-based analysis technique [LBAC96].

2.2 Refinement Theory

Just as the choice of a specification language to model a system is important, so is a sound theory that allows incremental development of a system specification. The two popular specification development methodologies are top-down and bottom-up. In a top-down methodology, one system specification is transformed into a more detailed specification. The transformation in such a methodology is commonly known as “elaboration”, “implementation”, or “refinement” as we will refer to it. In a bottom-up methodology, one system specification is transformed into a less detailed specification that is easier to understand. The transformation in such a methodology is known as “abstraction”. Since one type of
transformation can be seen as the reverse of the other, in the remainder of this section, we review refinement transformations.

As far as graphical specification languages are concerned, we are unaware of any formal refinement theory developed for the reviewed graphical languages. Despite their graphical syntaxes which make them look hierarchical, Statecharts as well as Modechart have “flat” semantics; that is, in order to reason about the system’s behavior, all details in the specification hierarchy have to be known. This is in part due to the syntax of both languages: 1) the labels on edges can refer to different state and mode variables that may be nested in a deeper, i.e., more detailed level of specification; and 2) edges can cross a state and mode boundaries to connect states and modes at different levels of nesting. Also, as mentioned earlier the notion of hierarchy in HMS differs from that in a top-down design. In HMS, all levels of specification remain part of the specification; however, in a top-down design each level of specification, once verified, stands independently of the higher levels.

On the other hand, there has been a significant amount of work done to support hierarchical specifications within textual languages. The general approach to refinement in the literature is based on defining a “relation” over system specifications. Such a relation can be defined either at the syntactic or semantic level of the specification language.

2.2.1 Refinement at the Syntactic Level

To define refinement at the syntactic level of a specification language, several formalisms augment a basic language with an operator. Refinement operators defined in the literature are based on the idea of syntactic substitution and are divided into two types:

*Relabeling.* The operator syntactically replaces an action in the specification by another action. Relabeling can in fact be considered as an abstraction operator; that is, it transforms a detailed specification to an abstract specification.

*Action refinement.* The operator syntactically replaces an action by a specification, i.e., a complex structure such as a process in process algebras, and a Petri net in Petri nets. In the context of process algebras, augmenting a basic process algebra with an action refinement operator requires two necessary notions in the basic language: 1) a notion of process prefixing which is often denoted as $P;Q$, as opposed to action prefix which is usually denoted as $a.P$ where $a$ is an action; and, as a result, 2) a notion of successful process termination, as opposed to a deadlocked or divergent
The semantic relations used for most refinements defined at this level are based on a notion of behavioral equality in the basic language. Thus, a main property of these refinements is that the refinement operator preserves semantic equality such as strong or weak bisimulation [Ace92, AH89, CvGG93, Jat93, Mil89] and rooted branching bisimulation [AM93] for process algebras, and event structure isomorphisms [vGG89] for event structures. This property however requires certain restrictions either on the application of the refinement operator or the semantic equality of the basic language.

In the case of relabeling, it is restricted to one-to-one, complement preserving functions. Being a function is tied to the fact that relabeling is an operator of the language and that the language may contain recursion. The other two restrictions are required to preserve potential synchronization and avoid unexpected synchronization in the presence of the parallel and synchronization operators. In the case of action refinement, the restrictions are more complex for languages that are based on interleaving semantic models than those with concurrent semantic models. At the heart of the problem is the fact that interleaving models reduce concurrent execution to interleaved execution. The famous problematic example is the process $a|b$ which will be equated by most behavioral equalities in the interleaving models to the process $a.b + b.a$. In the presence of action refinement, however, this equality is violated and thus semantic equality is not preserved by action refinement. As an example, consider the action refinement that syntactically replaces $a$ with the process $a_1.a_2$. In this case, any behavioral equality will not equate the refined processes, since $(a_1.a_2)|b$ can execute $a_1.ba_2$ while $a_1.a_2.b + b.a_1.a_2$ can not.

The problem stems from a disagreement in the “granularity” of an action between the basic and augmented languages: An action in the augmented language can be split and thus has a finer granularity than an action in the basic language. In the untimed interleaving models, the problem is resolved in two ways: 1) define a semantic equality that is tighter than in the basic language and thus it distinguishes true parallelism from interleaving [Ace92, AH89]; or 2) restrict refinable actions so that actions in the scope of a parallel operator cannot be refined [CvGG93].

In the timed process algebra CIPA [AM93], where action refinement preserves rooted branching bisimulation, the above problem was avoided by associating timing information with the labeled transition system. The occurrence time and duration of actions allow the
discovery of independent actions. In the earlier example, if \( a \) takes two time units and \( a_1 \) and \( a_2 \) take each one time unit, then the process \((a_1, a_2)\) will be equal to the process \(a_1, a_2, b + b, a_1, a_2\). The additional trace of the first process \((a_1, 0, b, 2, a_2, 1)\) is ill-timed but well-caused since it may only come from the refinement of the independent actions \( a \) and \( b \) executing concurrently. An additional property of action refinement in CIPA is that it preserves the timed occurrence and duration of an action; that is, an action is refined to a process that executes at the same time and for the same duration as the refined action.

We note that in the presence of parallel and synchronization operators, action refinement is restricted in a similar way to relabeling in order to preserve potential synchronization and avoid unexpected synchronization. In this case, an action and its complement are refined to complement processes, and \( \alpha \)-conversion is used along with syntactic substitution to ensure that actions in a refining process are “new” and hence are not caught in the scope of a synchronization unintentionally. (We will give examples of these restrictions in Chapter 4 where we present refinement for ACSR.)

In [BLB93], the authors present a special case of action refinement for a subclass of the process algebra LOTOS. They call their notion of refinement “functional decomposition”. In this syntactic approach, the authors present an algorithm that given a process and a bipartition of its actions, produces two processes that can synchronize on a new event \( \text{sync} \) and when running in parallel with \( \text{sync} \) hidden, the two processes are observation equivalent to the original process. The main idea of the algorithm is to coordinate progress between the two refining processes by exchanging messages on \( \text{sync} \) to inform each other that an action (including an internal action) is produced. In other words, the synchronizations over \( \text{sync} \) capture the dependencies (i.e., parallel versus sequential) between the actions of the abstract process. The bipartition of actions is essential and includes internal actions as well as information about an action occurrence. The algorithm is compositional with the operators in the LOTOS subclass under certain restrictions about the action bipartition and the absence of global choice and initial exit in the abstract process. Another essential restriction in this algorithm is that each refining process is restricted to have actions from one partition class only. In terms of ACSR, the latter restriction can be compared to the restriction that a parallel process is refined by refining its subprocesses in case they do not share resources.
In addition to process algebras, action refinement has also been defined for event structures [vGG89], Petri nets [Jat93, vGG89] and pomsets [NEL88]. Since concurrency is explicitly represented in these models, fewer restrictions were required on action refinement in order to preserve the semantic equality or pre-order in the case of pomsets. Two common restrictions in these formalisms are: 1) an action is refined the same way everywhere it occurs, which makes refinement an operator of the language, and 2) the refinement is unforgettable, i.e., does not eliminate actions or constraints (e.g., edges in Petri nets), which makes the refinement operator a syntactic substitution. In Petri nets, additional restrictions on the refining nets were necessary—e.g., the refining nets must have initial places with no incoming edges and final places with no outgoing edges [vGG89].

2.2.2 Refinement at the Semantic Level

Formalisms that define refinement at the semantic level of the specification language regard semantic equality as a notion that is too strong for transformations intended as “implementation” relations. Instead, their semantic relations allow one specification to describe more or fewer behaviors than the one related to it. The main goal of work done on refinement at the semantic level is to define transformation relations that preserve certain behavioral properties between related specifications—e.g., safety, liveness, fairness, divergence and deadlock freeness. When there is a higher level specification language (e.g., CSP with behaviors as labeled transition systems [Jif89]), semantic refinements are used as follows. First, the specifications are translated to their behaviors; e.g., the CSP terms are translated to labeled transition systems. Then, the semantic relation between the given behaviors is verified; e.g., verify whether the labeled transition systems are related by a given relation.

Refinement at the semantic level has been defined for labeled transition systems [GMM88, Jif89], I/O automata [Bes91, IV91, IV93, LT87], and traces, failures and divergences [BJO91, For93]. A common property of refinements in these models is their soundness with respect to trace set inclusion; that is, if there is a refinement from a detailed specification $Imp$ to an abstract specification $Spec$, then the traces of $Imp$ are a subset of the traces of $Spec$. A common assumption of refinements in these models is that the abstract and refined specifications share a common set of actions.

For timed models, refinement at the semantic level has been defined for variations of timed I/O automata [Bes91, IV91, IV93] and for timed CSP [Sch90]. In these timed
models, actions are instantaneous and time represents delays. In the case of models based on timed I/O automata, the refinement relations relate specifications such that the timed occurrence of mapped actions is the same. In the case of timed CSP, two refinement relations, called *timewise refinements*, have been defined to relate specifications in the untimed model to specifications in the timed model. These refinement relations build upon a hierarchical classification of the various models for timed CSP, e.g., traces, timed traces, and timed failures stability models. One motivation of the timewise refinements is an attempt to simplify the proof of timed properties by carrying them in an untimed setting, for which proofs systems are available. Therefore, an essential property about the two timewise refinements is that all properties satisfied by an untimed specification are satisfied by the related timed specification.

A more closely related work to ours is the trace-set ordering relations explored in [BG94, BGBAL96]. This work explored sample notions of trace preorders, e.g., according to resource consumption and speed. It is more general than the work presented in this dissertation, as it defined a set of conditions that any trace relation must satisfy in order for its trace-set extension to be compositional with the ACSR operators. This work however gave no syntactic characterization to any particular trace-set ordering relation. The refinement theory we present in this dissertation is more specific as it fixes one notion of trace preorder and characterizes it with a set of syntactic transformation rules that can be implemented within a design environment.

### 2.2.3 Discussion

Relabeling and action refinement as refinement operators in the modeling language are very practical and powerful concepts. Being functions whose semantics is defined through a notion of behavioral equality of the language, these refinement operators can be too restrictive in practice. Refinement at the semantic level of a language, on the other hand, may be impractical in the case of large or recursive systems. A hybrid approach to defining refinement seems more promising. By combining a semantic transformation with a syntactic one, a hybrid approach offers more freedom in defining the notion of consistency between a specification and its refinement. This is the approach we pursued in defining refinement for ACSR.

Another note about relevant refinement theories is that most of them have been defined for untimed models or models with a primitive notion of action, mainly communication,
and with time represented as delays. Most real-time systems in practice however have two
types of actions: communication and computation. A practical refinement theory may have
to combine relabeling and action refinement to deal with the two types of actions. Since
communication is usually instantaneous, relabeling seems a more appropriate refinement
operator for it. On the other hand, since computations are usually time consuming, action
refinement is a more appropriate refinement operator for them. This is the approach we
adopted in defining refinement at the syntactic level for ACSR.

2.3 Summary

The development of the CSR paradigm is based on the premise that just as notions of con-
currency, communication, and time are essential to model real-time systems, so are notions
of resources and priorities. The ACSR specification language, within the CSR paradigm,
combines these notions in a process algebraic setting. ACSR however is based on a textual
notation that often produces difficult to understand specifications. We propose a graphical
language, the Graphical Communicating Share Resources (GCSR), whose semantics is tied
to the semantics of ACSR and thus will facilitate the use of ACSR, on one hand, and on
the other profits from ACSR's analysis techniques and tools. The development of GCSR
benefits from the graphical notations reminiscent of popular graphical languages such as
control flow diagrams as well as the graphical languages for real-time systems which we
reviewed in this Chapter. In addition, GCSR allows the specification of a system in a
modular, hierarchical and thus scalable fashion. GCSR is presented in Chapter 3.

In order to make ACSR (and thus GCSR) suitable for a design environment, ACSR
must be augmented with a refinement theory that allows the specification of a real-time
system in an incremental way. Since ACSR encapsulates several notions (communication,
resource consumption, time, and priorities), the refinement theory for ACSR benefits from
refinement theories based on relabeling and action refinement, in addition it introduces a
notion of resource refinement. Our refinement theory for ACSR is presented in Chapter 4.
Chapter 3

The GCSR Language

This chapter presents the Graphical Communicating Shared Resources, GCSR, a graphical, formal specification language for real-time systems.

GCSR adopts intuitive graphical concepts of nodes and edges from control flow diagrams which have been widely used in software engineering through methodologies such as Structured Analysis [DeM87, WM85, HP87]. GCSR has several advantages: it allows scalable specification of complex systems in a modular and hierarchical fashion; it allows the integrated specification of the functional requirements of a system with its resource requirements in a natural way that produces easy to understand and modify specifications; and it has a precise operational, i.e., executable, semantics that can be tied to the semantics of the Algebra of Communicating Shared Resources, ACSR [BG94, LBGG94]. The GCSR-ACSR correspondence makes it possible to combine both types of specifications and to use the analysis techniques provided by the Verification, Execution, and Rewrite System for ACSR (VERSA) [CLX95b].

Chapter organization. Section 3.1 overviews the GCSR paradigm. Section 3.2 defines the GCSR syntax. Section 3.3 presents the GCSR operational semantics. Section 3.4 briefly reviews ACSR and describes the correspondence between GCSR and ACSR. Section 3.5 models and analyzes in the GCSR formalism an extended version of the standard railroad crossing benchmark example [HJL93]. Finally, Section 3.6 summarizes the main features of GCSR.
3.1 An overview

The GCSR paradigm is based on the view that a real-time system consists of a set of communicating components, called *processes*, that execute on a finite set of serially shared resources and synchronize with one another through communication channels. The use of shared resources is represented by timed *actions*, and synchronization is supported by instantaneous *events*. The execution of an action is assumed to take nonzero time units with respect to a global clock, and to consume a set of resources during that time. The execution of an action is subject to the availability of the resources it uses. Contention for resources is arbitrated according to the priorities of competing actions; priorities are static, i.e., fixed and are drawn from the set of natural numbers. To ensure uniform progress of time, processes execute actions synchronously. Time can be either dense or discrete; however, we consider discrete time only to simplify the description of the GCSR semantics and for implementation reasons.

Unlike an action, the execution of an event is instantaneous and never consumes any resource. Processes execute events asynchronously except when two processes synchronize through matching event names, i.e., channels.

Graphically, a GCSR process is represented by a finite set of nodes that are connected with directed edges. Figures 3.2 and 3.3 show the graphical GCSR objects. Before describing the details of these symbols, we first introduce a simple example to illustrate the intuition behind the GCSR objects.

3.1.1 Example

Figure 3.1 shows the GCSR specification of a variant of the gate component of a railroad crossing system [HJL93]. The GCSR specification of the gate is divided into two modules: an initial process, *Gate*, and a process responsible of lowering and raising the gate, process *GD*.

The behavior of the *Gate* system is as follows. When control enters the initial node of *Gate*, an instantaneous choice must be made between two possible next behaviors that correspond to the two outgoing edges. One possibility is to idle by transferring control through the unlabeled edge; the other possibility is to receive the event (*lower?*, 1), in which case control transfers to the Reference node named *GD* and the system starts behaving like the *GD* process. In the complete specification of the railroad crossing system where
the *Gate* is running in parallel with other components, e.g., train, the semantics of GCSR ensures that the transition on receiving the *lower* event has a higher priority than the idle transition.

Once control enters the initial node of the process *GD*, it stays in this node for 20 time units while continuously consuming the *cpu* resource at priority 1 and the *gate* resource at priority 1. This models the gate lowering activity. Afterwards, control transfers to the target node of the time labeled edge, sends the event (*Down!, 1*), and so on so forth. The nested process *GO* represents the gate raising activity. This process can be interrupted by the reception of the event *lower*; in this case, control transfers back to the process responsible of lowering the gate, *GD*. If there is no request to lower the gate, the gate is raised in 20 time units; it then sends the event *Up* the gate raised status followed by the event *done*, after which control transfers to the initial process *Gate*. In the latter case, the process *GO* relinquishes control by signaling the exception *done*.

### 3.1.2 GCSR Nodes and Edges

Figure 3.2 shows the graphical symbols for the GCSR nodes. The *Resource* attribute of a time-consuming node is a set of (resource, priority) pairs, with the restriction that each resource is listed at most once; this enforces the notion of serializable resource usage. The *Name* attribute of a reference node refers to the name of a GCSR process. The *Restrict* and *Close* attributes of a compound node are sets of event names and resource names, respectively.

The motivation for various node symbols in GCSR is a succinct and scalable representation of the different system activities and components. The instantaneous node
Figure 3.2: GCSR nodes

requires that no delay be allowed before the next activity. In contrast, the time-consuming node describes a time-consuming activity. Furthermore, the Resource attribute of a time-consuming node explicitly describes the required resources for the system activity, which makes it easy to modify any resource requirement to reflect different resource allocation and scheduling disciplines.

The nil node describes a halting process, i.e., end of system execution. The reference node allows the decomposition of a large specification into subspecifications which eases the visual structuring of such a specification. On the other hand, the compound node visually distinguishes a system action from a system component. It is essential in supporting scalable and modular specifications since it allows a designer to

1. group GCSR processes into a higher level entity,
2. connect several GCSR processes that are executed sequentially, and
3. reflect the fact that system components execute in parallel.

In addition to a structural modularity, compound nodes also provide for semantic modularity by encapsulating dependencies through their Restrict and Close attributes. The Restrict attribute identifies a set of communication events that are visible only among the GCSR processes inside the node; the Close attribute identifies a set of resources that are reserved for the nested GCSR processes, even if their time-consuming actions do not explicitly request them.

GCSR nodes can be connected with edges to describe sequential execution. GCSR offers four types of edges shown in Figure 3.3. We call unlabeled, event-labeled, and time-labeled edges as normal edges. The distinct symbols for a normal edge and an exception edge are
motivated by the desire to support a structured, hierarchical specification in which edges do not cross node boundaries and to distinguish graphically two types of control flow: one that is externally controlled by an interacting process and one that is triggered internally through voluntary release of control by raising an exception. The first type of control flow is described by a normal edge and the second by an exception edge. Control moves to the destination node of an exception edge when the process of the source node executes an exception event that labels the exception edge. The transfer of control through an exception edge allows synchronization between a process inside a compound node with an outside node and thus emulates a transition between nodes at different levels of nesting.

3.1.3 Informal Semantics

Intuitively, the behavior of a GCSR process consists of a sequence of execution steps each of which represents either a communication event or a time and resource consuming action. A communication event can be either a receive or send operation, respectively designated by the symbols “?" and “!" in front of the event name.

For example, in Figure 3.4 (a), once execution reaches the instantaneous node, the system sends the event named e at priority 1 and then execution moves instantaneously to the target node of the event-labeled edge from where execution continues. On the other hand, in Figure 3.4 (b) execution remains in the time-consuming node for 3 time units while using simultaneously the cup1 resource at priority level 1 and the cup2 resource at
priority level 2; after three time units in the source node, execution moves to the target node of the time-labeled edge.

In addition to this basic notion of sequential execution in GCSR, several processes can be combined through compound nodes to describe a large system where processes execute in sequence or in parallel. For example, in Figure 3.4 (c), once execution reaches the compound node, the GCSR process \( P \) is executed for at most \( t \) time units, after which execution moves to the target node of the time-labeled edge. The execution of \( P \) can also be terminated in two other ways. One way is through an interrupt; this is represented by the unlabeled edge, which can be taken any moment during the execution of \( P \). When this happens, the execution of \( P \) is aborted and execution moves to the target node. Another way of terminating the execution of \( P \) is through an exception raised by \( P \) sending the event \( a \). At this time, the execution moves to the target node of the exception edge which is labeled with the receive event \( a \) at priority \( p \). The difference is that in the interrupt case the enabling of the edge is caused by a process other than \( P \), whereas in the exception case the enabling of the edge is caused by the process \( P \) itself.

GCSR processes can also be combined in parallel by nesting them inside a compound node. When control reaches a compound node, it simultaneously enters all the initial nodes of its nested GCSR processes. Control can move through event-labeled edges in different nested processes in an interleaved fashion for unrestricted events, but synchronously in any two nested processes for restricted events. In addition, if control spends time in one nested process, then control in all the remaining nested processes must be in time-consuming nodes where it can spend time. This forces synchronous time passage between parallel processes. Furthermore, since resources are assumed to be serial, the set of resources used in all the time-consuming nodes that simultaneously have control must be disjoint. For example,
in Figure 3.4 (d) the processes $P_1$ and $P_2$ execute in parallel. Furthermore, $P_1$ and $P_2$ communicate privately through the event named $s$ and use the resource $r$ exclusively; that is, if the compound node is combined in parallel with another GCSR process, this latter can not communicate with $P_1$ and $P_2$ through the event $s$ and will not have access to the resource $r$ unless it requires it at a higher priority.

The overall behavior of a GCSR process can be formally described as a labeled transition system where each transition represents an execution step with the label being a communication event or a time and resource consuming action. As seen in the previous examples, there might be several execution steps that are simultaneously possible. The selection among such execution steps is done via a notion of priority that we describe in Section 3.3.

As mentioned in the introduction, a second way of defining the execution steps of a GCSR process is through a translation to the timed process algebra ACSR which also has an operational semantics. Each GCSR process is translated to an ACSR process. The translation is consistent with the direct semantics in the sense that the corresponding ACSR process has an equivalent labeled transition system to the one generated directly from the GCSR process; equivalence between the labeled transition systems is described in Section 3.4. The GCSR to ACSR translation which we have implemented in the GCSR tool set allows us to use the ACSR tool set VERSA to execute and analyze GCSR specifications.

### 3.2 The Syntax of GCSR

To define precisely the semantics of GCSR, either directly or via ACSR, we impose syntactic restrictions on how edges can connect nodes. For this, we next formally define GCSR nodes, edges and processes. We then define a set of rules that ensure the well-formedness of a GCSR process.

**Definition 3.2.1** Let $\mathcal{N}$ be the set of natural numbers, $\mathcal{R}$ be a set of resources and $\mathcal{L}$ be a set of event names. A GCSR node is a structure

$$[\text{name} : \text{type}, \text{attributes}]$$
where \textit{name} uniquely identifies the node and \textit{type} and \textit{attributes} are defined according to the following table:

<table>
<thead>
<tr>
<th>type</th>
<th>attributes</th>
</tr>
</thead>
<tbody>
<tr>
<td>instantaneous</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>time-consuming</td>
<td>${(r, p) \in \mathcal{R} \times \mathcal{N}}$ where each $r \in \mathcal{R}$ is listed at most once</td>
</tr>
<tr>
<td>nil</td>
<td>$\epsilon$</td>
</tr>
<tr>
<td>reference</td>
<td>\textit{name}</td>
</tr>
<tr>
<td>compound</td>
<td>${(G_1, \cdots, G_k), F, C}$ where $G_1, \cdots, G_k$ are GCSR processes, $F \subseteq \mathcal{L}$ and $C \subseteq \mathcal{R}$</td>
</tr>
</tbody>
</table>

For a node $n$, we refer to its type as $\psi_n(n)$ and to its attributes as follows:

1. For $[n: \text{time-consuming}, \{(r, p) \in \mathcal{R} \times \mathcal{N}\}]$, the action of $n$ is $\text{action}(n) = \{(r, p) \in \mathcal{R} \times \mathcal{N}\}$, and the set of resources used in $n$ is $\text{resrc}(n) = \{r \in \mathcal{R} | (r, p) \in \text{action}(n)\}$.

2. For $[n: \text{reference}, n']$, the name of $n$ is $\text{name}(n) = n'$.

3. For $[n: \text{compound}, \{(G_1, \cdots, G_k), F, C\}]$, the set of GCSR processes contained immediately inside $n$ is $\text{inside}(n) = \{G_1, \cdots, G_k\}$. The set of events restricted in $n$ is $\text{restrict}(n) = F$, and the set of closed resources is $\text{close}(n) = C$.

When the node does not have the correct structure, \textit{action}, \textit{resrc}, \textit{name}, \textit{inside}, \textit{restrict}, and \textit{close} are undefined. In the sequel, when the type of a node is irrelevant, we omit it and write $[n: \text{attributes}]$ for $[n: \text{type, attributes}]$. \hfill $\Box$

**Definition 3.2.2** Let $\mathcal{N}$ be a set of nodes. The function $\text{children} : \mathcal{N} \rightarrow 2^\mathcal{N}$ returns for each node $n \in \mathcal{N}$ the set of nodes that are immediately contained in $n$. It orders the set $\mathcal{N}$ into a forest where the leaves are either nil, instantaneous, time-consuming, or reference nodes, and intermediate nodes in a tree are compound nodes. We denote the transitive closure of $\text{children}$ by $\text{children}^*$.

**Definition 3.2.3** Let $\mathcal{N}$ be a set of nodes, $\mathcal{L}$ be a set of event names, and let $\mathcal{N}$ be the set of natural numbers. A GCSR edge is an element of $\mathcal{N} \times \text{Labels} \times \mathcal{N}$ where the set of labels, \textit{Labels}, is generated by the following grammar:

$$
\text{Labels} ::= (a!, p) \mid (a?, p) \mid (\tau, p) \mid t \mid \epsilon
$$
where $a \in \mathcal{L}$, $p \in \mathcal{N}$, $t \in \mathbb{N} \cup \{\infty\}$, and $\epsilon$ denotes the empty label, i.e., the edge is unlabeled.

For an edge $ed$, the type of $ed$ is denoted as $\psi_\mathcal{E}(ed)$ and defined as follows:

1. $\psi_\mathcal{E}(ed) = \text{event-labeled}$ if $ed$ is labeled with either $(\tau, p)$, $(a!, p)$ or $(a?, p)$;
2. $\psi_\mathcal{E}(ed) = \text{time-labeled}$ if $ed$ is labeled with $t$;
3. $\psi_\mathcal{E}(ed) = \text{unlabeled}$ if $ed$ is labeled with $\epsilon$; and
4. $\psi_\mathcal{E}(ed) = \text{exception}$ if $ed$ is labeled with $(a!, p)$ or $(a?, p)$. \hfill \Box$

We now have all the components to define a GCSR process.

**Definition 3.2.4** A GCSR process, $G$, is a five tuple $(\mathcal{N}, \mathcal{I}, \mathcal{E}, \mathcal{L}, \mathcal{R})$ where

1. $\mathcal{L}$ is a set of event names and $\mathcal{R}$ is a set of resource names such that $\mathcal{L} \cap \mathcal{R} = \emptyset$;
2. $\mathcal{N}$ is a finite set of nodes;
3. $\mathcal{I} \subseteq \mathcal{N}$ is a set of initial nodes with a distinctive initial node $n_0 \in \mathcal{I}$ that is the unique initial node at the highest level, i.e., it is not the child of any node. The function $\text{initial}(G)$ returns $n_0$; and
4. $\mathcal{E} \subseteq \mathcal{N} \times \text{Labels} \times \mathcal{N}$ is the set of edges with event names derived from $\mathcal{L}$.

For a given GCSR process $G$, we refer to its set of nodes as $\mathcal{N}(G)$, set of initial nodes as $\mathcal{I}(G)$, set of edges as $\mathcal{E}(G)$, set of event names as $\mathcal{L}(G)$, and its set of resource names as $\mathcal{R}(G)$. \hfill \Box$

The following definition describes well-formed GCSR processes that have semantics.

**Definition 3.2.5** A GCSR process $(\mathcal{N}, \mathcal{I}, \mathcal{E}, \mathcal{L}, \mathcal{R})$ is well-formed if its set of nodes $\mathcal{N}$ and set of edges $\mathcal{E}$ satisfy the following syntactic rules:

1. An instantaneous node must have at least one out-edge and only unlabeled or event labeled out-edges.
2. A time-consuming node must have one time-labeled edge and nothing else.
3. No edge out of a nil node.
4. An edge connects nodes that belong to the same GCSR process and at the same nesting level.
5. A reference and a compound node can have at most one time-labeled edge. \hfill \Box
Most of the above restrictions reflect the semantics of nodes. The restriction that at most one time-labeled edge out of a node requires a justification. It eliminates the following semantic ambiguity. Assume we have a node that has two time-labeled edges one with label 3 and the other with label 10. It becomes unclear where control is at time 3 relative to entering the node: did it exit the node through the edge labeled with 3, or did it remain in the node because the edge labeled with 10 allows it to do so?

**Short-hand notation.** Figure 3.5 shows a short-hand notation we adopt for GCSR processes to eliminate unnecessary nesting. The GCSR process which contains only the node \([n: \text{compound}, \{G\}, \emptyset, \emptyset]\) at the highest level is represented by \(G\) with the initial node of \(G\) tagged with the identifier \(n\). The identifier \(n\) can be seen as the *name* of the GCSR process \(G\). This short-hand notation allows us to construct a GCSR specification

![Short-hand notation in GCSR](image)

**Figure 3.5: Short-hand notation in GCSR**

as a collection of GCSR processes that refer to one another by name through the reference nodes.

In order to define the semantics of a GCSR process, all of its reference nodes must refer to defined GCSR processes. In the next section, whenever we use a GCSR process, we mean a *well-formed* GCSR process with all reference nodes have corresponding GCSR processes.

### 3.3 The Operational Semantics of GCSR

As mentioned in the introduction, the GCSR language has a correspondence to the ACSR language. It is therefore possible to obtain the behavior of a GCSR specification through translating it to an ACSR specification. Such a translation, however, loses the connection with the graphical entities of the specification. In the case of an error in the specification, this makes it hard to locate the node or edge where the error occurred. We therefore
develop the formal semantics of GCSR directly. This helps guide a simulation of a GCSR specification which gives a better feedback to the user about the active states of the system as well as its flow of execution. As we show in Section 3.4, the two ways to define the GCSR semantics are consistent with one another.

The operational semantics of GCSR is defined in terms of a labeled transition system together with a preemption relation that captures the notions of priorities [LBGG94, BG94].

**Definition 3.3.1** A labeled transition system is a four-tuple $(S, \Sigma, s_0, \rightarrow)$, where $S$ is the set of states, $\Sigma$ is the alphabet, $s_0 \in S$ is the start state, and $\rightarrow \subseteq S \times \Sigma \times S$ is the transition relation. We write $s \xrightarrow{\alpha} s'$ for a transition $(s, \alpha, s') \in \rightarrow$. □

The main difficulties in defining the states and transitions of the labeled transition system of a GCSR process stem from the time-labeled and the unlabeled edges. Time-labeled edges require a notion of “local clock” that keeps track of how long control can stay inside the source node of the edge. The local clock must be decreased with each “tick” of the system global clock and control must leave the source node when the local clock reaches zero. In addition, to ensure uniform passage of time in the system, all local clocks must be decreased at the same time. Since nodes do not have the information about how long control is allowed to remain in them, we had to consider GCSR edges that are extended with local clocks as part of the state definition; that is, a state in the labeled transition system is defined as a pair of a set of nodes together with a set of relevant edges. The nodes of a state can simultaneously have control. Each relevant edge has its source node in the nodes of the state and has a local clock associated with it to determine how long control is allowed to stay in the source node.

A transition in the labeled transition system represents two types of visible activities in the system: 1) time passage and resource consumption in all the nodes of the source state, and 2) communication events that label relevant edges. Unlabeled edges, however, do not contribute with visible activities. They must therefore be transparent; that is, the presence of an unlabeled edge must not change the behavior of the system. As an example, we would like the two GCSR processes in Figure 3.6 to have the same behavior since they basically differ in unlabeled edges. (The nodes in Figure 3.6 have their names listed inside for easier reference when we revisit this example to illustrate how transitions are defined.)

To handle GCSR unlabeled edges, we define the transitions of the labeled transition system for a GCSR process in two steps: First, we define the semantics of GCSR without
notions of parallel composition nor synchronization. At this level, we basically “prune out” the unlabeled edges and determine the sequential behavior of a process. We then define the (actual) GCSR semantics on top of the first labeled transition system, taking into account parallel execution and synchronization. The two labeled transition systems only differ in their transition relation.

In the following sections, we define the labeled transition system for a given GCSR process \( G = (N, I, E, L, R) \). We use node and edge to denote the (syntactic) GCSR entities, and use state and transition to denote the (semantic) labeled transition system entities.

### 3.3.1 Alphabet

The set of alphabet used to label the transitions is

\[
\Sigma \subseteq \{(a?, p)| a \in \mathcal{L} \land p \in \mathbb{N}\} \\
\cup \{(a!, p)| a \in \mathcal{L} \land p \in \mathbb{N}\} \\
\cup \{(\tau, p)| p \in \mathbb{N}\} \\
\cup 2\{|r, p| r \in \mathcal{R} \land p \in \mathbb{N}\}
\]

### 3.3.2 States

Informally, a state in the labeled transition system contains all GCSR nodes that simultaneously have control together with next possible activities that can be produced at some or all of these nodes. There are three possible types of activities: (1) take an unlabeled edge to move to another node, (2) take an event-labeled edge and produce an event, or
(3) consume time and resources for one time unit. We next introduce auxiliary definitions that we will use to define a state.

**Parallel nodes**: Two nodes $n_1, n_2 \in \mathcal{N}$ are parallel, $n_1 \parallel n_2$, if

$$\exists n \in \mathcal{N}. (\exists G_1, G_2 \in \text{inside}(n). n_1 \in \mathcal{N}(G_1) \land n_2 \in \mathcal{N}(G_2)).$$

Parallel nodes belong to distinct components inside a compound node.

**Consistent nodes**: A set of nodes $N$ is consistent if it satisfies the following condition:

$$\forall n, n' \in N. (n \neq n' \implies n \in \text{children}(n') \lor n' \in \text{children}(n) \lor n \parallel n').$$

The nodes in a consistent set are either related through children, or are parallel.

**Configuration**: For a given consistent set of nodes $N \subseteq \mathcal{N}$, the configuration of $N$, $conf(N) = q$, is a set of nodes constructed according to the following rules:

1. $N \subseteq q$

2. If a node has children in $q$, then it is also in $q$:

$$\forall n \in \mathcal{N}. (\text{children}(n) \cap q \neq \emptyset \implies n \in q)$$

3. If a compound node $n$ is in $q$ and any of its components doesn’t have a node in $N$, then the designated initial node of the component is added to $q$:

$$\forall n \in q. (\text{inside}(n) = \{G_1, \ldots, G_k\} \land \mathcal{N}(G_i) \cap q = \emptyset \implies \text{initial}(G_i) \in q \text{ for } i = 1, \ldots, k)$$

This rule ensures that whenever a compound node is in $q$, $q$ contains one node from each of the compound node’s components.

Note that by definition of a configuration, we can prove that nodes in a configuration are either related through the children* relation or belong to different components inside a compound node. A configuration can therefore be used to describe the set of nodes that simultaneously have control. The notion of configuration alone is however insufficient to define a state in the transition system since it lacks information about how long control can stay in any particular node. For this, we extend GCSR edges with a notion of local clock that determines how long control can stay in the source node of an edge; the extended edges together with a configuration form a state.
Extended edges: An extended edge \( ((s, l, d), c) \) is defined such that:

1. \( (s, l, d) \in E \) and
2. \( c = \begin{cases} 
  n & \text{if } 0 < n \leq l \in (\mathbb{N} \cup \{\infty\}) \\
  0 & \text{otherwise}
\end{cases} \)

The constant \( c \) represents a local clock value that describes how long control can stay in the source node of the edge. As we show shortly, when time passes while the control is in the source node, the clock is decreased.

We denote the set of extended edges of \( G \) as \( E^+ \subseteq E \times (\mathbb{N} \cup \{\infty\}) \). We denote the initially extended edges of \( E \) as \( \text{extend}(E) \subseteq E^+ \) where each time-labeled edge \( (s, t, d) \in E \) is extended as \( ((s, t, d), t) \).

Relevant edges: The set of relevant, extended edges for a set of nodes \( q \) is

\[
\text{relevant}(q) = \{(s, l, s'), c) \in E^+ | s \in q\}
\]

We now have the constituents of a state.

State: The set of states of the labeled transition system for \( G \) is

\[
S = \{(q, E) \in (2^N \times 2^{E^+}) | \text{conf}(q) = q \land E \subseteq \text{relevant}(q)\}
\]

Start state: The start state of the labeled transition system of \( G \) is

\[
s_0 = (\text{conf}(\{n_0\}), \text{relevant}(\text{conf}(\{n_0\})) \cap \text{extend}(E))
\]

where \( n_0 = \text{initial}(G) \), i.e., \( n_0 \) is the distinguished initial node of \( G \). We take the intersection of the relevant edges and the initially extended edges to ensure that the initial state \( s_0 \) is a state where control has just entered the set of nodes and no time has elapsed yet.

### 3.3.3 Transitions

The computation model of GCSR is such that a GCSR process, \( G \), can execute either an event instantaneously or an action which consumes time and resources. A transition in the labeled transition system of \( G \) therefore represents either event-labeled edges that are taken in \( G \), or resource consumption inside time-consuming nodes.
Unlabeled edges in GCSR do not produce a visible activity. The semantics of an unlabeled-edge is as follows: an unlabeled edge is taken, if it leads to a node where a visible activity can be produced. Such a semantics requires a “pre-processing” step where the unlabeled edges are pruned out. This step is done by the labeled transition system at the first level. At this level, we are interested in determining the sequential, visible activities which a GCSR process can produce possibly after taking unlabeled edges. Given the labeled transition system of the first level, we then define the labeled transition system for a GCSR process that may contain parallel composition and synchronization.

The transition relations of the labeled transition systems at both levels will make use of the following two auxiliary functions.

**Definition 3.3.2** The function \( \text{next} : S \times E^+ \rightarrow S \) returns the next state when a given edge is taken or after one time unit elapses in a given state.

The function \( \text{new} : S \times E^+ \rightarrow N \) returns the set of nodes which control enters when a given edge is taken out of a state or after one time unit elapses in a state:

\[
\begin{align*}
\text{next}((q, E), ((s, \alpha, d), c)) &= (q', E') \\
\text{new}((q, E), ((s, \alpha, d), c)) &= q''
\end{align*}
\]

where

\[
q' = (q - q_1) \cup q_2 \\
E' = (E - E_1) \cup E_2
\]

and

1. if \( \alpha \not\in (N \cup \{\infty\}) \) then

\[
\begin{align*}
q_1 &= \{n \in q | n \in \text{children}^*(s)\} \\
q_2 &= \text{conf}\{\{d\}\} \\
E_1 &= \{((n, l, n'), c) \in E | n \in q_1\} \\
E_2 &= \{((n, l, n'), c) \in \text{extend}(E) | n \in q_2\} \\
q'' &= q_2
\end{align*}
\]

2. if \( \alpha \in (N \cup \{\infty\}) \) then
\[ q_1 = \{ n \in q | \exists ((n', t, n''), 1) \in E. (s \in children^*(n') \land n \in children^*(n'')) \} \]
\[ q_2 = \{ n \mid \exists ((n', t, n''), 1) \in E. (n \in conf\{n''\} \land q_1 \subseteq children^*(n'')) \} \]
\[ E_1 = \{ ((n, l, n'), c) \in E \mid n \in q_1 \} \]
\[ \cup \{ ((n, t, n'), c) \in E \mid s \in children^*(n) \} \]
\[ E_2 = \{ ((n, l, n'), c) \in extend(E) \mid n \in q_2 \} \]
\[ \cup \{ ((n, t, n'), c - 1) \mid ((n, t, n'), c) \in E \land s \in children^*(n) \land n \notin q_1 \} \]
\[ q'' = \{ n \mid \exists ((n, l, n'), c) \in E_2 \} \]

When the edge is not time-labeled (case 1), the edge is taken, i.e., control transfers to the target node of the edge. The configuration is updated as follows: remove the source node of the edge and all its children (set \( q_1 \)), and add the configuration of the target node of the taken edge (set \( q_2 \)). Note that since the GCSR process \( G \) is well-formed, \( q' \) is a configuration. The set of relevant edges is updated as follows: remove all edges whose source nodes were removed (set \( E_1 \)), and add edges whose source nodes were added (set \( E_2 \)). The set of new nodes is the same as \( q_2 \), the newly entered nodes.

When the edge is time-labeled (case 2), the next function decreases by one time unit the local clocks of the given edge and those edges whose source nodes contain the node \( s \).

(Any time-labeled edge out of a parallel node will not be decremented at this level; it will be decremented at the second level.) Any edge whose local clock is 1 is taken and, thus, is removed from the set of relevant edges along with its source node and children (set \( q_1 \)).

The added nodes, set \( q_2 \), correspond to the configuration of the target node of a taken edge that is at the highest nesting level. The set of nodes new contains all nodes added through \( q_2 \) as well as those nodes which remained in \( q \) but whose local clocks are decremented.

**Notation.** To distinguish between the labeled transition systems at the two levels, we denote the transition relation at the first level as \( \xrightarrow{\omega} \) and at the second level as \( \xrightarrow{\cdot} \).

**Transitions at the First Level**

Transitions at the first level prune out unlabeled edges to produce sequential behavior of the GCSR process. More specifically, a transition at the first level is the result of taking unlabeled edges that will lead to a node where a visible (event or resource consumption)
activity can be produced. Transitions at this level can be either instantaneous or time-consuming. They are defined by the following two rules.

**Instantaneous transition:** An instantaneous transition is the result of taking unlabeled edges that lead to a node where an event-labeled edge can be taken. It is defined by the rule $\text{PreActI}$.  

\[
\text{PreActI} \quad ((s'_{k-1}, \epsilon, s_k), 0) \in E_{k-1}, \quad ((s'_k, (a*, p), s_{k+1}), 0) \in E_k \quad \text{cond}(\text{PreActI})
\]

\[
(q_0, E_0) \xrightarrow{[a*, a]} (q', E')
\]

where $a*$ denotes $\tau$, $a'$, or $a$ for $a \in \mathcal{L}$ and

\[
\text{cond}(\text{PreActI}) = \bigwedge_{i=1, k} s'_i \in \text{children}^*(s_i) \\
\land \exists q_1, \cdots, q_k \in \mathcal{N}. \ \bigwedge_{i=1, k} q_i, E_i = \text{next}((q_{i-1}, E_{i-1}), ((s'_i, \epsilon, s_{i+1}), 0)) \\
\land (q', E') = \text{next}((q_k, E_k), ((s'_k, (a*, p), s_{k+1}), 0))
\]

The condition $\text{cond}(\text{PreActI})$ ensures the following: 1) the event-labeled edge is produced at a node that is in the same component as the taken unlabeled edges. That is, the taken unlabeled edges sequentially lead to the node where the event is executed; and 2) the set of extended edges in the predicate are the result of taking the unlabeled edges according to the $\text{next}$ function. The first part of condition $\text{cond}(\text{PreActI})$ ensures that unlabeled edges are “transparent” in a GCSR process, i.e., they do not affect its behavior. To illustrate the necessity of this condition, consider the GCSR process $G_1$ in Figure 3.6 (a). The following transitions are illegal: first take the unlabeled edge from $s_1$ to $s_2$, second the unlabeled edge from $n_1$ to $n_4$, and then the $(a!, 2)$-edge to conclude that

\[
(\{n, s_1, n_1\}, \{(s_1, \epsilon, s_2), 0\}, ((s_1, \epsilon, s_4), 0), ((n_1, \epsilon, n_2), 0), ((n_1, \epsilon, n_4), 0))\]

\[
\xrightarrow{[a!, 2]} (\{n, s_3, n_4\}, \{((n_4, (b?, 2), n_5), 0)\}.
\]

In this case, after taking the second unlabeled edge, the event $(a!, 2)$ does not come from a sequentially connected node but rather comes from a parallel component. A legal transition requires taking the edge labeled with $(a!, 2)$ after taking the first unlabeled edge; that is, the following transition is legal:

\[
(\{n, s_1, n_1\}, \{(s_1, \epsilon, s_2), 0\}, ((s_1, \epsilon, s_4), 0), ((n_1, \epsilon, n_2), 0), ((n_1, \epsilon, n_4), 0))\]

\[
\xrightarrow{[a!, 2]} (\{n, s_3, n_1\}, \{((n_1, \epsilon, n_4), 0), ((n_1, \epsilon, n_2), 0)\}.
\]

Now, to see the effects of the illegal transition, let us take into account the fact that the events $a$ and $b$ are restricted in the GCSR process $G_1$ of Figure 3.6 (a); that is, in
G1 both components inside the compound node must synchronize on \( a \) and \( b \). In the illegal transition, allowing the second unlabeled edge (from \( n1 \) to \( n4 \)) to be taken creates a deadlock in \( G1 \). This makes \( G1 \) behave differently from the GCSR process \( G2 \) of Figure 3.6 (b) which does not have the unlabeled edges and does not deadlock.

**Time-consuming transition:** A time-consuming transition at the first level is the result of taking zero or more unlabeled edges that lead to a time-consuming node where time and resources can be consumed.

\[
\begin{align*}
\text{PreActT} & \quad ((s'_{k-1}, \epsilon, s_k), 0) \in E_{k-1}, \quad ((s'_{k}, t, s_{k+1}), c) \in E_{k-1} \\
\text{cond(PreActT)} & \quad (q_0, E_0) \xrightarrow{A} (q', E')
\end{align*}
\]

where

\[
\text{cond(PreActT)} = \bigwedge_{i=1}^{k} s'_{i} \in \text{children}^*(s_i) \quad \land \quad \psi_N(s'_k) = \text{time-consuming} \quad \land \quad A = \text{action}(s'_k) \quad \land \quad \exists q_1, \cdots, q_{k-1} \in N. \quad (\bigwedge_{i=1}^{k-1} (q_i, E_i) = \text{next}((q_{i-1}, E_{i-1}), ((s'_{i}, \epsilon, s_{i+1}), 0))) \quad \land \quad (q', E') = \text{next}((q_{k-1}, E_{k-1}), ((s'_{k}, t, s_{k+1}), c))
\]

The condition \( \text{cond(PreActT)} \) is similar to \( \text{cond(PreActI)} \), in that it ensures that the taken unlabeled edges sequentially lead to the time-consuming node where the resources are consumed for one time unit.

**Definition 3.3.3** For the transition rule **PreActI**, we say that the edge

\[
((s'_k, (\ast, p), s_{k+1}), 0) \text{ initiates the transition } (q_0, E_0) \xrightarrow{\{\ast, p\}} (q', E').
\]

Similarly, for the transition rule **PreActT**, we say that the edge \((s'_k, t, s_{k+1}), c\) initiates \((q_0, E_0) \xrightarrow{A} (q', E')\).

To be brief, when the states \((q_0, E_0)\) and \((q', E')\) are clear from their context, we simply write \(((s'_k, (\ast, p), s_{k+1}), 0) \text{ initiates } \xrightarrow{\{\ast, p\}}\), and \(((s'_k, t, s_{k+1}), c) \text{ initiates } \xrightarrow{A}\).

\[\square\]

**Transitions at the Second Level**

Given the transitions at the first-level, which filter out the unlabeled edges, the labeled transitions at the second level define the GCSR semantics, including parallel composition and synchronization.

**Notation.** In the sequel, we adopt the following notation: \( \overline{\alpha} \) denotes the inverse of event label \( \alpha \), e.g., \( \overline{a?} = a? \). We also overload the syntax of the function \( \text{new} \) and use \( \text{new}(q) \) for \( \text{new}((q, E), ((s, \alpha, d), c)) \) when \( E \) and \((s, \alpha, d), c\) are clear from their context.
**Instantaneous transition:** An instantaneous transition is labeled with an event. It is the result of three possible ways of taking relevant edges: 1) take an event-labeled edge that is labeled with an unrestricted event; 2) take two event-labeled edges with inverse event labels; or 3) take an event-labeled edge and an exception edge with inverse event labels. These transitions are described by the following three rules.

\[ \text{ActI} \quad \frac{(q, E) \xrightarrow{a \ast, p} (q', E')}{(q, E) \xrightarrow{[a \ast, p]} (q', E')} \quad \text{cond}(\text{ActI}) \]

where

\[ \text{cond}(\text{ActI}) = \langle s, (a \ast, p), d, 0 \rangle \text{ initiates } \xrightarrow{a \ast, p} \]

\[ \Rightarrow \quad \psi_E(\langle s, (a \ast, p), d, 0 \rangle) = \text{event-labeled } \land \]

\[ \forall n \in N.(\text{ } s \in \text{children}^* (n) \Rightarrow \]

\[ a \notin \text{ restricted}(n) \land \]

\[ \forall (n, (\overline{a \ast}, p'), n'), 0) \in E. \psi_E(\langle n, (\overline{a \ast}, p'), n' \rangle) \neq \text{ exception edge } \]

Condition \text{cond}(\text{ActI}) states that the event-labeled edge which initiates the first-level transition has a visible event \( a \); that is, the event \( a \) is not restricted in any ancestor node of \( s \) and does not synchronize with an exception edge from any ancestor node of \( s \). The second case is captured by rule \textbf{Exception} defined shortly.

The second type of instantaneous transitions is when events are synchronized. This is possible when there are two first-level, instantaneous transitions that can be executed in parallel and that are labeled with inverse events, i.e., \( a! \) and \( a? \). There are two possible ways to synchronize events: simultaneously taking two event-labeled edges, and simultaneously taking an event-labeled edge and an exception edge with an inverse event label. The two cases are described by the \textbf{ParCom} and \textbf{Except} rules, respectively.

\[ \text{ParCom} \quad \frac{(q, E) \xrightarrow{a! \ast, p} (q_1, E_1), (q, E) \xrightarrow{a? \ast, p'} (q_2, E_2)}{(q, E) \xrightarrow{\tau \ast, \overline{p} + p'} (q', E')} \quad \text{cond}(\text{ParCom}) \]

where

\[ q' = \text{conf}( (q_1 \cap q_2) \cup (\text{new}(q_1) \cup \text{new}(q_2)) ) \]

\[ E' = \{ (n, l, n'), c \in (E_1 \cup E_2) | n \in q' \} \]
To understand the new configuration $q'$, we note the following: $q_1$ and $q_2$ share compound nodes that contain 1) either $s_1$ or $s_2$ or both, and 2) nodes that are parallel to $s_1$ and $s_2$ but do not participate in the transitions. Furthermore, $q_1$ may contain nodes with unlabeled outedges that will lead to $s_2$; a similar comment applies to $q_2$ and node $s_1$. The latter nodes will not be in $q_1 \cap q_2$ and should not be in the final configuration. The new nodes in $q_1$ and $q_2$ however must be in the new configuration since control has just moved there. Finally, to construct $q'$, we take the configuration of the above sets of nodes to ensure that we include those nodes which are parallel to $s_1$ and $s_2$ but did not participate in the transitions at the first level.

Condition $cond(ParCom)$ ensures that 1) both edges are event-labeled edges, 2) they are parallel edges, and 3) their events can propagate from nested nodes up to a common ancestor node where they can synchronize; that is, the events are not restricted in an ancestor node that contains one edge but not the second and they do not synchronize with an exception edge at such an ancestor node. Synchronization with an exception edge is handled by the following rule.

\[
\text{Except} \quad \frac{q, E \xrightarrow{(s_{1}, p)} (q_1, E_1), (q, E) \xrightarrow{p, p'} (q', E')} {(q, E) \xrightarrow{p, p'} (q', E')} \quad cond(\text{Except})
\]
where

\[ Cond(Except) = ((s, (a^*, p), d), 0) \text{ initiates } \frac{[a^*, p]}{\rightarrow} \land ((s', (\overline{a^*}, p'), d'), 0) \text{ initiates } \frac{[\overline{a^*}, p']}{\rightarrow} \]

\[ \Rightarrow (\psi_\xi((s, (a^*, p), d), 0)) = \text{event-labeled } \land \psi_\xi((s', (\overline{a^*}, p'), d'), 0)) = \text{exception} \land \exists s \in \text{children}^*(s') \land \forall ((n, (\overline{a^*}, p'''), n'''), 0) \in E, (s \in \text{children}^*(n) \land n \in \text{children}^*(s') \Rightarrow n = s') \land \forall n \in N. (s \in \text{children}^*(n) \land n \in \text{children}^*(s') \Rightarrow a \notin \text{restrict}(n)) \]

Condition \( Cond(Except) \) ensures the following: 1) only one of the initiating edges is an exception edge; 2) the source of the exception edge, \( s' \), contains the event-labeled edge; 3) \( s' \) is the first ancestor with the exception edge; and 4) the event is not restricted before reaching the node \( s' \).

**Time-consuming transition:** The last type of labeled transitions is labeled with an action that represents the consumption of resources for one time unit.

\[
\begin{align*}
\text{ParT} & \quad \frac{(q, E) \xrightarrow{A} (q_1, E_1), \cdots, (q, E) \xrightarrow{A_k} (q_k, E_k)}{(q, E) \xrightarrow{A} (q', E')} \quad \text{cond(Pa}\text{rT)}
\end{align*}
\]

where

\[
\begin{align*}
q' & = \text{conf}(\bigcap_{i=1,k} q_i \cup \bigcup_{i=1,k} \text{new}(q_i)) \\
E' & = \bigcup_{i=1,k} \{(n, l, n'), c) \in E_i | n \in \text{new}(q_i)\} \\
A & = \bigcup_{i=1,k} A_i \\
\text{cond(Pa}\text{rT)} & = \forall i, j = 1, \cdots, k. (i \neq j \Rightarrow A_i \cap A_j = \emptyset) \\
& \quad \land \forall (q, E) \xrightarrow{A'} (q'', E''). ((s, t, d), c) \text{ initiates } \xrightarrow{A'} \\
& \quad \Rightarrow \exists ((s_i, t_i, d_i), c_i) \text{ initiates } \xrightarrow{A_{i\rightarrow}} (s = s_i \land A' = A_i) \lor s \nparallel s_i \\
& \quad \land \text{conf}(N) = N \text{ where} \\
& \quad N = \bigcup_{i=1,k} \{n| ((s_i, t_i, d_i), c_i) \text{ initiates } \xrightarrow{A_{i\rightarrow}} \land s_i \in \text{children}^*(n)\}
\end{align*}
\]

The new configuration \( q' \) is constructed in a similar way to the parallel, instantaneous transition, rule \textbf{ParCom}. To construct the new set of relevant edges, \( E' \), we cannot simply take the union of the relevant edges \( E_i \) for \( i = 1, \cdots, k \) because the \textit{next} function does not reduce the local clocks of nodes that are parallel to the source node of the time-labeled edge being considered. Thus, the sets of relevant edges \( E_i \) contain time-labeled edges of parallel nodes whose local clocks were not decreased. To make sure \( E' \) reflects synchronous time passage, it must select those edges in \( E_i \) which reflect time passage. These edges are easily tracked: their source nodes which have been selected by the function \textit{new}.
Condition \( Cond(ParT) \) ensures that a time consuming transition is allowed under three restrictions: 1) when the participating time-consuming nodes have disjoint sets of resources. This restriction stems from the assumption that all resources are serial, and thus at any time at most one action can execute on a particular resource; 2) the set of transitions at the first level is maximal, i.e., any other time-consuming transition at the first level is from a node that is not parallel to another participating node; and 3) the configuration from which the time consuming step is executed contains only the time-consuming nodes which initiate the transitions at the first level and their ancestor compound nodes. This ensures that the time-consuming nodes are parallel and that the time-consuming step occurs only if all parallel components can participate—thus synchronously.

To deal with the Close attribute of compound nodes in a GCSR process, the condition of rule \( ParT \) must be changed to construct the set \( A \) incrementally from the lowest level of nesting up as follows:

1. partition \( s_1, \ldots, s_k \) into sets \( C_1, \ldots, C_l \) (note that \( 1 \leq l \leq k \)) such that each set is immediately contained inside a compound node, i.e., \( C_j \subseteq \text{children}(n_j) \) for a compound node \( n_j \) and \( j = 1, \ldots, l \).

2. the label of the transition within each set \( C_j \) is

\[
A_j = \bigcup_{s_i \in C_j} A_i \cup \{(r,0) | r \in \text{close}(n_j) \land r \not\in \text{resrc}(\bigcup_{s_i \in C_j} A_i)\}
\]

The transition is possible if \( A_j \cap A_{j'} = \emptyset \) for \( j, j' = 1, \ldots, l \) and \( j \neq j' \).

3. repeat step 1 and 2 with \( n_1, \ldots, n_l \) until one fixed partition \( C_1 \) is reached.

To deal with the reference nodes in a GCSR process, whenever a reference node \( [n : \text{reference}, n'] \) is in a configuration of a state, the set of relevant edges of the corresponding state is computed by syntactically substituting the reference node with a new compound node \( [n' : \text{compound}, ([G], \emptyset, \emptyset)] \) where \( G \) is the GCSR process which corresponds to \( n' \).

### 3.3.4 Prioritized Operational Semantics

The prioritized operational semantics of GCSR is defined through the prioritized labeled transition system, \( \rightarrow_{\pi} \), which uses the preemption relation \( \prec \) of ACSR[BG94, LBGG94] to refine the unconstrained labeled transition system \( \rightarrow \).
Informally, the selection among two transitions that are simultaneously possible is either nondeterministic, or is arbitrated according to the following three rules for selecting a transition labeled with \( \alpha \) over a transition labeled with \( \beta \) (\( \beta < \alpha \)): 1) \( \alpha \) and \( \beta \) are events with the same label and \( \alpha \) has a higher priority; 2) \( \alpha \) and \( \beta \) are actions and \( \alpha \) uses a subset of the resources used in \( \beta \) at priorities no lower than in \( \beta \) and with at least one resource in \( \alpha \) at a higher priority; or 3) \( \alpha \) is a \( \tau \) event (i.e., synchronization of two events) with a non-zero priority and \( \beta \) is a time-consuming action. (The technical reasons behind these rules are rather complicated and can be found in [LBGG94].)

**Definition 3.3.4** The labeled transition system \( \rightarrow^\pi \) is defined as follows: \( q \xrightarrow{\alpha} q' \) if

a) \( q \xrightarrow{\alpha} q' \) is an unprioritized transition, and

b) There is no unprioritized transition \( q \xrightarrow{\beta} q'' \) such that \( \alpha < \beta \).

\[ \square \]

### 3.4 The GCSR-ACSR Correspondence

As mentioned in the introduction, the semantics of the GCSR formalism corresponds to the Algebra of Communicating Shared Resources (ACSR) [LBGG94] \(^1\). We next briefly review ACSR and present the translation from ACSR to GCSR. We then describe the translation from GCSR to ACSR. We finally show that the translations are *sound*, in that the semantics of an ACSR process and its corresponding GCSR process (and vice versa) are the same.

#### 3.4.1 ACSR

ACSR augments CCS [Mil80] with dense [BG94] and discrete [LBGG94] time and resource-consuming, prioritized actions. ACSR has two types of actions: time and resource-consuming actions, called *actions*, and instantaneous communication actions, called *events*. An action consists of a possibly empty set of pairs \( \langle r, p \rangle \) where \( r \) is the resource name and \( p \) is its priority, with the restriction that each resource is represented in the set at most once. The action \( \emptyset \) represents the idling action since no resources are used. An event in

\(^1\)In fact, the notion of exception in GCSR is more general than the end of scope in ACSR: GCSR allows multiple exception edges out of a node while ACSR allows one end of scope event only. For this, we restrict our attention to a subset of GCSR where a reference and compound node can have at most one exception edge.
A CSR consists of a pair \((e, p)\) where \(e\) is a label and \(p\) is its priority. The special event label \(\tau\) represents synchronization of two events with \emph{complementing} event labels \(e\) and \(\tau\).

Let \(P\) range over the domain of terms, \(A\) range over the domain of actions, \(e\) and \(b\) range over the domain of event labels or \(\tau\), \(F\) range over the set of event labels, \(I\) range over the set of resource names, and let \(C\) range over the domain of process names. The syntax of CSR is given by the following grammar:

\[
P ::= \text{NIL} \mid A^t : P \mid (e, p).P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P \triangledown_{\{b,p\}} (P_1, P_2, P_3) \mid [P]_I \mid P \setminus F \mid C
\]

The semantics of an CSR process is defined in terms of a labeled transition system together with a notion of prioritized transition represented by the preemption relation described in section 3.3.4. For completeness, we list in Appendix D the operational semantics rules to derive the labeled transition system for an CSR process. We next informally describe the CSR semantics; for a detailed description, please refer to [LBGG94].

NIL is a process that executes no action, e.g., it is initially deadlocked. There are two prefix operators that correspond to the two types of actions. The first, \(A^t : P\), executes a timed, resource-consuming action \(A\) for \(t \in \mathbb{N}^+\) time units (i.e., \(t\) is a positive integer) and proceeds to the process \(P\). The second prefix operator, \((e, p).P\), executes the instantaneous event \(e\) at priority level \(p\) and proceeds to \(P\). The Choice operator \(P_1 + P_2\) represents possibilities—either of the processes may be chosen to execute, subject to the event offerings and resource limitations of the environment. The operator \(P_1 \parallel P_2\) is the concurrent execution of \(P_1\) and \(P_2\). Note that in CSR, execution of events can be interleaved while actions are executed synchronously by the processes in the parallel operator.

The Scope construct \(P \triangledown_{\{b,p\}} (P_1, P_2, P_3)\) binds the process \(P\) by a temporal scope [LG85], and incorporates both the features of timeouts and interrupts. \(P\) executes for a maximum of \(t \in \mathbb{N}^+ \cup \{\infty\}\) time units. The scope may be exited in three ways. First, if \(P\) successfully terminates within time \(t\) by executing an event labeled with \(\overline{b}\), then control proceeds to the “exception-handler” \(P_1\) (here, \(b\) may be any label other than \(\tau\)). Second, if \(P\) fails to terminate within time \(t\), then control proceeds to the “timeout-handler” \(P_2\). Lastly, at any time while \(P\) is executing it may be interrupted by \(P_3\)'s execution of an action or event, and the scope is then departed.

The Close operator, \([P]_I\), produces a process \(P\) that uses the resources in the set \(I\) exclusively. The Restriction operator, \(P \setminus F\), limits the behavior of \(P\): events with labels
in $F$ are permitted to execute only if they synchronize and become the internal event $\tau$.

Each process constant $C$ is associated with a process definition of the form $C \overset{\text{def}}{=} P$. This provides an alternative way of defining recursive operator $rec X.P$.

ACSR offers two basic notions of behavioral equivalence that are defined over the prioritized labeled transition system. The first equivalence relation is based on strong bisimulation [Mil80], $\sim_s$, which ensures that equivalent processes match each other’s labeled transitions; it is a congruence relation [Ger91]. The second is based on weak bisimulation, $\approx_s$, which ensures that equivalent processes match each other’s transitions that are labeled with non-$\tau$ events but allows one process to make transitions on $\tau$ that an equivalent process does not match.

### 3.4.2 From GCSR to ACSR

Figures 3.7 and 3.8 show the main steps to translate a GCSR process to an ACSR process, where the translation function $T_{GA}$ is denoted as T.

The translation starts from the initial node of a GCSR process and recursively traverses all reachable nodes. Step 1 binds the translation of an initial node to the ACSR process variable name $C$ and returns the ACSR process variable $C$; this step is done for each initial GCSR node. In step 2, the nil node is translated to the NIL ACSR process. In step 3, an instantaneous node is translated to an ACSR Choice process; each ACSR subprocess in the Choice process corresponds either to

1. the translation of the target node of an unlabeled outedge, or
2. an event-prefix ACSR process where the event is the label of an outedge and the next process in the prefix is the translation of the target node of the edge.

Note that in this translation the event syntax must be converted from GCSR to ACSR, i.e., $(a!, p)$ and $(a?, p)$ are converted to $(\pi, p)$ and $(a?, p)$, respectively. In step 4, a time-consuming node is translated to an action-prefix ACSR process where the duration of the action is the label of the time-labeled edge out of the time-consuming node.

In step 5, the box node can be either a reference or compound node. The translation of such a node produces a Scope ACSR process $P \triangleleft (P_1, P_2, P_3)$ where the main process $P$ is the translation of the box node without its outedges. Figure 3.8 shows the translation when the box node lacks some outedges. Basically, the corresponding process is replaced with the ACSR NIL process. We assume in these simplified translations that the event label $\text{dummy}$ is a special label that is not used anywhere except on exception edges in
\begin{enumerate}
  \item \[ C = T(N1) \]
  \[
  \text{return } C
  \]
  
  \item \[
  \text{return } \text{NIL}
  \]
  
  \item \[
  \text{return } e_1.T(N1) + ... + e_n.T(Nj) + T(M1) + ... + T(Mk)
  \]
  
  \item \[
  \text{return } A^1 : T(N1)
  \]
  
  \item \[
  \text{return } P \bigtriangleup^e \langle T(N1), T(N2), S \rangle
  \]
  \[
  \text{where}
  \]
  \[
  S = e_1.T(U1) + ... + e_l.T(Ul) + T(V1) + ... + T(Vn)
  \]
  \[
  \text{and}
  \]
  \[
  P = C
  \]
  \[
  \text{or}
  \]
  \[
  P = [ ( T(N01) \parallel \, \ldots \parallel \, T(N0k) ) \setminus F ] \setminus I
  \]
  \[
  \text{N01, \ldots, N0k are initial nodes of G1, \ldots, Gk}
  \]
\end{enumerate}

\textbf{Figure 3.7: GCSR to ACSR translation, } T_{GA}
GCSR and as end of scope events in ACSR.

Figure 3.8: Special cases of Step 5 of the GCSR to ACSR translation

Figure 3.7 does not show how to handle loops in a GCSR process. To handle loops, the translation function marks nodes being translated. When it starts the translation of a node, it first checks whether the node is marked. If the node is marked, it returns the name of the node; recall that a node is of the form [name:type, attributes]. If the node is not marked, it marks it in two cases: the node is an initial node and has an incoming edge, or the node is not an initial node and has more than one incoming edge. In the second case, if the node has one incoming edge, than this is the edge used to reach the node and translation will not revisit the node. After the marking step, the translation then proceeds according to the steps in Figure 3.7 to produce a process $P$. In the last step, if the node was marked, the node is unmarked and the recursive process $rec\ X.(P)$ is returned where $X$ is the name of the node. (Recall that $rec\ X.P$ is a syntactic sugar for $X \overset{\text{def}}{=} P$.)

The GCSR to ACSR translation is sound, i.e., the GCSR process and its corresponding ACSR process have the same behavior.
**Theorem 3.4.1** For each GCSR process $G$, we have $T_{GA}(G) \sim G$.

**Proof:** The proof is by induction on the structure of $G$. Since the LTS of $G$ is constructed starting from the initial node of $G$ towards reachable nodes, each step of the induction considers one possible type of initial node, examines its possible transitions and shows that $T_{GA}(G)$ has the exact same transitions. Appendix A.1 contains the detailed proof. □

### 3.4.3 From ACSR to GCSR

Figure 3.9 describes the ten steps of translating an ACSR process to a GCSR process, where the function $G$ represents $T_{AG}$, the translation function from ACSR processes to GCSR processes.

The ACSR to GCSR translation is symmetric to the GCSR to ACSR translation shown in Figure 3.7. Furthermore, it is sound and produces a subset of GCSR processes that we characterize next.

**Theorem 3.4.2** For each ACSR process $P$, we have $T_{AG}(P) \sim P$.

**Proof:** The proof proceeds by induction on the structure of the ACSR process $P$. It connects the operational semantics rules of ACSR to those of the semantics of GCSR. It is symmetric to the proof of Theorem 3.4.2 and thus is omitted as it does not bring any new insights. □

**Property 1** For a given ACSR process, the corresponding GCSR process, $T_{AG}(P)$, has nodes that satisfy the following properties:

1. every node has at most one incoming edge;
2. every instantaneous node has either one event-labeled or two unlabeled out-edges;
3. every compound node, $n$, can be one of three forms: $[n : (\{G\}, R, \emptyset)]$, $[n : (\{G\}, \emptyset, C)]$, or $[n : (\{G_1, G_2\}, \emptyset, \emptyset)]$;
4. every reference node has one exception out-edge, one time-labeled out-edge, and one unlabeled edge; and
5. every compound node with out-edges has the form $[n : (\{G\}, \emptyset, \emptyset)]$, one exception out-edge, one time-labeled out-edge, and one unlabeled edge. □
1. NIL

2. \( A : P \)

3. \((e,p).P\)

4. \( P_1 + P_2 \)

5. \( P_1 \parallel P_2 \)

6. \( P \bowtie F \)

7. \([P]I\)

8. \( P \bigwedge_t (a,p) (P_1, P_2, P_3) \)

9. \( C \)

10. \( C = P \)

Figure 3.9: ACSR to GCSR translation, \( T_{AG} \)

50
Definition 3.4.1 We call a well-formed GCSR process that satisfies Property 1 basic GCSR process.

3.4.4 Efficiency of the Translations: Tying it Together

In this section we show that applying the two translations back-and-forth produces related processes. When the result of the translations are ACSR processes, the relation is strong equivalence ∼. When the result of the translations are GCSR processes, the relation is graph isomorphism modulo some graphical transformations we define shortly.

Theorem 3.4.3 For every ACSR process P, we have TGA(TAG(P)) ∼ P.

Proof: By Theorem 3.4.1 we have TGA(TAG(P)) ∼ TAG(P); and by Theorem 3.4.2 we have TAG(P) ∼ P. Thus, we get TGA(TAG(P)) ∼ P. □

Transformation of GCSR Specifications. In addition to the analysis of a GCSR specification, another advantage of ACSR equivalence relations is their use as the basis to minimize or expand GCSR specifications. Figures 3.10 and 3.12 show transformations (minimization or expansion) that produce equivalent specifications according to the prioritized strong equivalence ∼. We denote each of these transformations as TGG.

Eliminate process nodes. There are two transformations that allow the elimination of process nodes: one to replace a reference to a process with a cyclic edge (Figure 3.10, 1) and the other to eliminate an unreachable nil node (Figure 3.10, 2.a and 2.b). (For transformation 2.b, recall that the event label dummy is a special label that is used on exception edges only. Since no normal event-labeled edge can use this label, the exception edge will never be taken.)

Eliminate edges. There are two transformations to eliminate edges in a GCSR process: one to remove unnecessary unlabeled edges (3) and the second to merge consecutive “identical” nodes (4). Note the required restriction that the removed instantaneous node does not have any incoming edge from the graph of P. Without this restriction, the simplified GCSR process may not be equivalent to the original GCSR process. To show the requirement of this restriction, consider the GCSR process in Figure 3.11. The original GCSR process can not execute event e after executing g; however, the
Eliminate process nodes

1. \[ C \] \[ \ast \] \[ \text{C} \]

2.a \[ \infty \] \[ \text{G(P)} \]

2.b \[ \text{(dummy*,p)} \] \[ \text{G(P)} \]

Eliminate edges

3. \[ \text{G(P)} \]

4. \[ A \] \[ t_1 \] \[ A \] \[ t_2 \] \[ \times \] \[ A \] \[ t_1 + t_2 \] \[ \times \]

If instantaneous node has one unlabeled incoming edge

Figure 3.10: GCSR transformations, \( T_{GG} \) (part 1 of 2)

simplified GCSR process can execute \( e \) after executing \( g \). Thus, the two processes are not equivalent.

Eliminate boxes. Extra nested compound nodes, which can be due to reference node binding, can be eliminated (Figure 3.12, 5, 6, and 7).

Eliminate duplicates. The last GCSR transformation merges “identical” portions in a GCSR process (Figure 3.12, 8).

\[ \text{P} = e.\text{NIL} + \text{rec} \ X. \ (f.g.X) \]

\[ \text{P} = e.\text{NIL} + f.g.P \]

Figure 3.11: Illegal GCSR transformation
Eliminate boxes

5. 

6. 

7a. 

7b. 

Eliminate duplicates

8. 

Figure 3.12: GCSR transformations, $T_{GG}$ (part 2)
Theorem 3.4.4 (Soundness of the transformations) For any valid GCSR processes $G$ and $G'$, if $T_{GG}(G) = G'$, then $G \sim G'$.

Proof: The proof is trivial and essentially compares for each transformation the ACSR processes that correspond to $G$ and $G'$ and uses the soundness of the GCSR to ACSR translation proven in Theorem 3.4.1.

We next define a notion of syntactic equivalence between GCSR processes that basically does not distinguish between GCSR processes that differ only in the choice of node names.

Definition 3.4.2 Two GCSR processes $G = (N, I, E, L, R)$ and $G' = (N', I', E', L', R')$ are isomorphic, $G \equiv G'$, if $L = L'$, $R = R'$ and there is a bijection $h : N \rightarrow N'$ such that

1. $\forall n \in I, h(n) \in I'$ and $h(\text{initial}(G)) = \text{initial}(G')$;

2. $\forall n \in N$, we have:
   \begin{align*}
   h([n: \text{nil}, e]) &= [h(n): \text{nil}, e] \\
   h([n: \text{reference}, n']) &= [h(n): \text{reference}, n'] \\
   h([n: \text{time-consuming}, R]) &= [h(n): \text{time-consuming}, R] \\
   h([n: \text{compound}, \{G_1, \ldots, G_k\}, F, C]) &= [h(n): \text{compound}, \{G'_1, \ldots, G'_k\}, F, C]
   \end{align*}
   where $G_i \equiv G'_i$ for $i = 1, \ldots, k$;

3. and $\forall n, n' \in N$ we have $(n, \alpha, n') \in E$ iff $(h(n), \alpha, h(n')) \in E'$ and $\psi_E((n, \alpha, n')) = \psi_E'(h(n), \alpha, h(n'))$.

It is easy to prove that for any GCSR processes $G$ and $G'$, if $G \equiv G'$ then $G \sim G'$, but the reverse does not always hold.

Theorem 3.4.5 For every GCSR process $G$, there exists a GCSR process $G' = T_{GG}(G)$ such that $T_{AG}(T_{GA}(G)) \equiv G'$.

Proof: The proof outlines the order in which the transformations defined earlier are applied on the GCSR process $T_{AG}(T_{GA}(G))$ to transform it to an equivalent GCSR process $G'$ that satisfies Property 1. Appendix A.2 contains the detailed proof.

The connection between a GCSR process $G$ and the GCSR process obtained from first translating $G$ to ACSR and then back to GCSR, i.e., $T_{AG}(g_t(G))$, is not surprising due to the soundness and symmetry of the translations. In practise, however, such a double translation is inefficient for two reasons. One reason is that the translations lack a check for
structural identity between parts of a process. For instance, a user may draw a compact
GCSR process $G$ by grouping structurally identical parts. The translations, however,
unfold any such optimizations and produce multiple copies of the identical parts.

A second reason for the inefficiency of the translation is due to the fixed cardinality
of the ACSR operators. More specifically, the Parallel and Choice operators in GCSR
generalize their binary counterparts in ACSR. This results, respectively, in unnecessary
nested boxes, and instantaneous nodes and unlabeled edges. In addition, the Restrict and
Close operators in GCSR also generalize the unary cardinality of their counterparts in
ACSR. Restrict and Close in GCSR can be combined together as well as with all of the
other ACSR operators except for the prefix. This also results in additional boxes in the
GCSR process obtained through translation.

3.5 Busy Railroad Crossing at an Intersection

In this section, we illustrate how to model and analyze an extended version of the standard
railroad crossing benchmark [HJL93]. The system is to control a railroad crossing that is
near a road intersection; see Figure 3.13. This version of the railroad crossing example was
motivated by the recent deadly accident in Illinois, USA where a train in a similar railroad
crossing collided with a school bus.

![Figure 3.13: Top view of the Railroad Crossing example](image)

The railroad crossing contains a single track and is guarded by a gate that moves up
and down and an entry and exit sensor that detect the presence of a train near the crossing,
The road intersection is guarded by a traffic light. When a train approaches the railroad crossing, it signals the traffic light to turn green so that the vehicles in the intersection can move away from the track. The train is also detected by the entry sensor and signals the gate to move down so that no vehicle can start crossing the track. When the train passes the crossing, it is detected by the exit sensor and signals the gate to move up.

The gate is initially up. When it receives a signal to move down, it moves down and waits for a signal to move up from the train when it passes the crossing. While up, the gate constantly signals its status to vehicles at the crossing.

When a vehicle approaches the railroad crossing, it checks whether the gate is up. If the gate is up and no other vehicle is crossing the track, it immediately crosses; otherwise, the vehicle waits until either the gate is up or the track is empty. After passing the crossing, the vehicle then waits at the intersection for the traffic light to turn green, at which time it leaves the area.

**Constraints.** The resource constraints are the following: Obviously a train and a vehicle cannot share the track at the crossing. In addition, only one vehicle can be crossing the track at any time. Furthermore, the distance between the traffic light at the intersection and the track can hold one vehicle at a time; however, drivers are not aware of this constraint and thus may think they have passed the track while the rear end of their vehicles remain too close to the track. (This assumption is motivated by one investigation scenario in the Illinois accident.) We denote the track at the crossing by the *crossing* resource, and the area between the crossing and the intersection by the *C* resource (see Figure 3.13). To simplify the example, we assume that the gate and traffic light run on dedicated resources (e.g. processor) and, therefore, we do not represent their resources.

The timing assumptions about the system are summarized in Table 3.1.
Table 3.1: Timing assumptions in the Railroad Crossing example

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Traffic Light</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tn = 5$</td>
<td>time to enter the crossing</td>
<td>$tg = 4$ : duration of green cycle</td>
</tr>
<tr>
<td>$tp = 5$</td>
<td>inside the crossing</td>
<td>$ty = 1$ : duration of yellow cycle</td>
</tr>
<tr>
<td>$d = 2$</td>
<td>minimum delay between trains</td>
<td>$tr = 3$ : duration of red cycle</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Gate</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>$td = 4$</td>
<td>time to move gate down</td>
<td>$tc1 = 1$ : time to pass crossing</td>
</tr>
<tr>
<td>$tu = 6$</td>
<td>time to move gate up</td>
<td>$tc2 = 2$ : time to pass intersection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Bus</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$tb1 = 3$</td>
<td>time to pass crossing</td>
<td></td>
</tr>
<tr>
<td>$tb2 = 2$</td>
<td>time to turn intersection</td>
<td></td>
</tr>
</tbody>
</table>

**Our design.** The high-level structure of our design contains five components: *train* and *gate* which are the components of the standard railroad crossing system, and *traffic light, car* and *bus* which are the added components. The five components execute concurrently and coordinate with one another through the synchronization events listed in Table 3.2.

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tn = 5$</td>
<td>time to enter the crossing</td>
</tr>
<tr>
<td>$tp = 5$</td>
<td>inside the crossing</td>
</tr>
<tr>
<td>$d = 2$</td>
<td>minimum delay between trains</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Traffic Light</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tg = 4$</td>
<td>duration of green cycle</td>
</tr>
<tr>
<td>$ty = 1$</td>
<td>duration of yellow cycle</td>
</tr>
<tr>
<td>$tr = 3$</td>
<td>duration of red cycle</td>
</tr>
</tbody>
</table>

Figure 3.14: High level GCSR specification of the railroad crossing

Table 3.2: Synchronization events in the Railroad Crossing example

<table>
<thead>
<tr>
<th></th>
<th>Train</th>
<th>Gate</th>
<th>lower : gate move down</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tn = 5$</td>
<td>time to enter the crossing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$tp = 5$</td>
<td>inside the crossing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$d = 2$</td>
<td>minimum delay between trains</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Traffic Light</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>$tg = 4$</td>
<td>duration of green cycle</td>
<td></td>
</tr>
<tr>
<td>$ty = 1$</td>
<td>duration of yellow cycle</td>
<td></td>
</tr>
<tr>
<td>$tr = 3$</td>
<td>duration of red cycle</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Gate</th>
<th>Car</th>
</tr>
</thead>
<tbody>
<tr>
<td>$td = 4$</td>
<td>time to move gate down</td>
<td>$tc1 = 1$ : time to pass crossing</td>
</tr>
<tr>
<td>$tu = 6$</td>
<td>time to move gate up</td>
<td>$tc2 = 2$ : time to pass intersection</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Bus</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$tb1 = 3$</td>
<td>time to pass crossing</td>
<td></td>
</tr>
<tr>
<td>$tb2 = 2$</td>
<td>time to turn intersection</td>
<td></td>
</tr>
</tbody>
</table>

Restrict=[green, lower, raise, up] Close=[C]

Restrict=[Green, lower, raise, up] Close=[crossing]
Figures 3.15 and 3.16 describe the GCSR specification of a car and bus, respectively. A car starts with two possible behaviors: it can idle to indicate it is far from the crossing, or it can be near the crossing checking whether the gate is up. If the gate is up, the car takes $t_{c1}$ time units to pass the crossing; this is described by the time-consuming node with resource $\{(\text{crossing,}1)\}$. Afterward, the car can either wait too close to the track, which is described by the second time-consuming node with the $\text{crossing}$ resource, or it can enter the intersection and waits for the traffic light to turn green. The choice depends on the availability of the intersection resource $C$. Once it enters the intersection, as soon as the light turns green, the car then departs in $t_{c2}$ time units. A car can then return to the crossing immediately. This allows us to describe a flow of cars at the crossing. The behavior of the bus is similar to the car’s.

![Figure 3.15: GCSR specification of a car](image1)

![Figure 3.16: GCSR specification of a bus](image2)
The traffic light component functions as follows: Initially, the light at the intersection is green. Any time during its regular green-yellow-red cycle of operation, the traffic light can be interrupted by the reception of the \textit{Green} signal from a train that is near the crossing. At this time, the cycle is restarted at the green stage (\textit{GR}).

We will model two versions of the train and gate components to illustrate potential safety violations in the system when either component fails to operate according to the specification.

### 3.5.1 Safe Design

Figures 3.18 and 3.19 show the GCSR specification of the gate and train, respectively. The gate is a simpler version of Example 3.1.1 where no resource and no observable event (i.e., unrestricted event) is used.

The train (Figure 3.19) functions as follows. It first delays its arrival for \( d \) time units; this is described by the initial time-consuming node. It then can either signal the traffic light to turn green (\textit{Green}!, 1), or further delay its arrival. The second alternative is described by the time-consuming node which is preceded by the \((\tau, 2)\)-labeled edge. The \( \tau \) event is required due to the preemption relation where a \( \tau \) event preempts all timed actions.
With the $(\tau, 2)$-labeled edge the choice between idling and signaling the traffic light is non-deterministic which makes the train inter-arrival times variable; without the $(\tau, 2)$-labeled edge, the train will be able to synchronize with the traffic light through the $(\text{Green!}, 1)$ event immediately after the original delay, which makes the train’s behavior periodic. After signaling the traffic light to turn green, the train signals the gate to move down, $(\text{lower!}, 1)$, and then takes $tn$ time units to actually enter the crossing. The train then spends $tp$ time units in the crossing (the time-consuming node with resource $\{(\text{crossing}, 2)\}$), after which it signals the gate to move up $(\text{raise!}, 1)$. Note that once the train is in the crossing, it can not be preempted, since it does not have an alternative action to execute. In particular, if a bus or car must be in the crossing at the same time, the operational semantics of the parallel execution in GCSR will lead the train and the whole system to deadlock. Such a scenario models a collision between a train and a bus or car.

To ensure that our design is safe, we need to check that there never is a collision between a train and a car or a bus. First, we note that each component can separately execute forever; this is due to the loops in the GCSR processes. Second, the concurrent behavior of the components consists of synchronized events (i.e. $\tau$’s) interleaved by timed usage of the $\text{crossing}$ and $C$ resources, possibly at priority zero. Thus, by operational semantics of the parallel execution, if all the events are synchronized and there is no resource contention, then the components will execute forever; in other words, the system will not reach a deadlocked state. Furthermore, if we ignore the synchronization events and conceal the identities of the resources, the concurrent behavior of the components consists of infinite sequences of idling actions, i.e., $\emptyset$. From this, we can conclude that our design is safe if the following bisimulation holds:

$$\text{System} \parallel \{\text{crossing}, C\} \approx_{\pi} \text{Wait}$$

where $\text{Wait} \overset{\text{def}}{=} \emptyset : \text{Wait}$ and the operation $\parallel$ hides the resource names in the behavior of the process.
For the timing assumptions shown in Table 3.1, we used our automated GCSR-to-ACSR translation and the VERSA toolset to prove that our design satisfies the above bisimulation; and thus our design is safe.

### 3.5.2 A Faulty Train

In this version, we assume that the train component is faulty: a train may miss to signal the traffic light to turn green. This version of the train is shown in Figure 3.20 where a second \((\text{lower!}, 1)\)-labeled edge is added to the train process of Figure 3.19.

We substituted the faulty train GCSR process \textit{FaultyTrain} for the \textit{Train} GCSR process in Figure 3.14. We then used the automated GCSR-to-ACSR translation and VERSA to check whether the resulting design is safe. We found that the design is unsafe due to two possible collisions. One collision is between a train and a car and is due to the following scenario: at time 20, a bus is waiting at the intersection for the traffic light to turn green, a car needs one more time unit to pass the crossing, and a train signals the gate to move down but fails to signal the traffic light to turn green. At time 24, the traffic light turns green and the bus starts turning while the car’s rear-end is still too close to the track. (This is described by the control being 1) for the bus process in the time-consuming node with an out-edge labeled with \(lb2\), 2) for the car in the time-consuming node which is the alternative of the compound node, and 3) for the train in the time-consuming node with out-edge labeled with \(tn\).) One time unit later, the bus is still making its turn and, thus, the car could not move away from the track, while the train enters the crossing. The second collision is between a train and a bus and is similar to the first one.

### 3.5.3 A Faulty Gate

In this version, we assume that the gate does not move but rather sets flashing lights on when it must be down and off when it must be up. Furthermore, we assume that the gate
is faulty: when it receives a signal to move down (i.e., turn on flashing lights), the gate does not respond and rather keeps signaling vehicles that it is up. This version is shown in Figure 3.21 where the time-consuming action which represents the closing activity in Figure 3.18 is replaced by the GCSR process $FGC$. This latter can send the event $(up!, 1)$ which could be received by a vehicle that is near the crossing.

We substituted the faulty gate GCSR process $FaultyGate$ for the $Gate$ GCSR process in Figure 3.14 and verified the safety of the resulting design. We found that the design is not safe due to two possible collisions. One is due to allowing a bus at time 2 and then a car at time 5 to cross the track despite receiving a signal $lower$ from a train approaching the crossing at time 2. In this case, when the bus passes the crossing at time 5, the traffic light is not green. When the car finishes passing the crossing at time 7, its rear-end remains too close to the track since the bus is still in the intersection. However, at this time, the train enters the crossing and collision occurs. The second possible collision occurs between the train and a bus at time 10. It is the result of allowing the bus to pass the crossing at time 8 despite the fact that a train signaled its approach at time 5. In this case, the bus needed the crossing for one more time unit when the train enters the crossing.

**Faulty Train and Gate.** When both the train and the gate are faulty, the resulting design has 18 possible collisions, four of which are the ones described earlier. (VERSA allows the user to get statistics about the labeled transition system of a process, e.g., number of states, transitions, deadlocked states.)
3.6 Summary

We have presented the Graphical Communicating Shared Resources (GCSR). The GCSR language is distinguished from other graphical languages for real-time systems by its syntax and precise semantics. GCSR offers several types of nodes and edges for a modular and structured hierarchical specification. The semantics of GCSR through a translation to ACSR allows us to offer both a graphical and textual specification language and to analyze GCSR specifications using process algebraic techniques such as bisimulation checking and state exploration. The equivalence relations of ACSR makes it possible to minimize a GCSR specification or expand it to include more details. In addition, the direct, operational semantics of GCSR facilitates the implementation of a simulator for GCSR, which gives a better visual analysis of a GCSR specification.

To make GCSR suitable for the hierarchical design of a real-time system, it must be augmented with a notion of refinement that allows the ordering of GCSR specifications. In the next Chapter we augment ACSR and thus GCSR with a refinement theory.
Chapter 4

Refinement in ACSR

A common approach to specifying a complex system is to decompose it into components that may execute concurrently and communicate among one another to produce the system’s activities such as external communication and computation. While the internal structure of a component is unimportant, completion of each activity in the order specified is. As the system is refined and specified more completely, each component may itself become a collection of subcomponents. A refinement is correct if it preserves the ordering of the system’s activities.

In this paper, we augment the discrete-time version of ACSR [LBGG94] with a refinement theory that is the basis for the top-down development of distributed real-time systems in ACSR. With this refinement theory, we envision the top-down development of a complex system to begin with an ACSR specification that describes the behavior of the object system at a certain level of abstraction. At this stage, the system’s resource requirements may not be exactly known and are therefore estimated. Subsequent stages of the development process gradually describe the system at more detailed levels towards a specification that is appropriate to permit the system implementation. These stages may use different notation, e.g., different event names, describe subcomponents of an abstract component, and tighten the resource requirements.

Top-down development in ACSR is accommodated through an ordering relation over process terms, called ACSR refinement, whose semantics is defined by an ordering relation over traces, called trace refinement. The trace refinement relation maps traces from the refined specification to traces of the abstract specification while preserving timed occurrences of the specification events. This is essential for real-time systems since properties
about a real-time system behavior, e.g., safety property are generally expressed in terms of the occurrences of particular specification events. Another important property of trace refinement is that it guarantees that ACSR refinement is \emph{compositional}. ACSR refinement can therefore be applied in a modular fashion in the case of a large system.

ACSR refinement consists of transformation rules that syntactically rewrite an ACSR process to another. The transformation rules combine notions of action refinement [NEL88, vGG92, AM93, CvGG93, Jat93] and relabeling [Mil89, Hoa85] together with a notion of resource refinement. A refining process may introduce new events and use fewer resources than the abstract process. In addition to their soundness with respect to trace refinement, the transformation rules augment ACSR with a relational refinement theory where a process can be refined to multiple processes as opposed to a unique process. This allows a designer to derive several refinements from a specification, each of which may reflect different design emphases, e.g., optimal structure and resource usage.

**Chapter organization.** Section 4.1 briefly reviews ACSR and uses a simple router example to illustrate the proposed types of refinement. Section 4.2 first defines trace refinement and its properties; it then examines possible extensions of trace refinement to a relation over sets of traces, a possible semantics of ACSR processes. Section 4.3 uses the unprioritized semantics of ACSR to present two ACSR process refinements: the Hoare ordering and Egli-Milner ordering extensions of trace refinement. For each of these extensions, the section first shows the conditions for a compositional trace-set refinement, and then defines the set of transformations laws for it. Section 4.4 reports the effects of priorities on refinement. Section 4.5 uses a simple example to illustrate the use, advantages, and limitations of the presented notion of refinement. Section 4.6 compares our approach to refinement with others and discusses the limitations of our approach. Appendix B contains selected proofs of lemmas and theorems stated in various sections of this chapter.

**4.1 Motivation**

Before covering the details of ACSR process refinement, it is important to understand the practical motivation for our form of refinement. We therefore first review the syntax of ACSR and then describe a simple example of how we envision refinement to occur.

Let \( P \) range over the domain of ACSR process terms \( \text{Proc} \); \( r_1, r_2, \cdots \) range of the
domain of resources $\mathcal{R} : p_1, p_2, \cdots$ range over the domain of priorities; $t$ range over the discrete time domain $\mathbb{N} \cup \{\infty\}$; $a$ range over the set of event labels $\mathcal{L} \cup \mathcal{T} \cup \{\tau\}$; the set $I$ range over the power set resources $\mathcal{P}(\mathcal{R})$; and let $F$ range over the power set of non-$\tau$ labels $\mathcal{P}(\mathcal{L} \cup \mathcal{T})$. Additionally, we assume an infinite set of free term variables, $FV$, which is ranged over by $X$. The syntax of ACSR is given by the following grammar

$$P ::= \text{NIL} \mid \{(r_1, p_1), \cdots, (r_n, p_n)\}^t : P \mid (a, p).P \mid P_1 + P_2 \mid P_1 \parallel P_2 \mid P \triangle_i (P_1, P_2, P_3) \mid [P]_j \mid P \backslash F \mid \text{rec } X.P \mid X$$

To make the example more readable, we will associate names with events as well as actions, and drop the priorities from the events and actions; this is a temporary syntactic deviation from ACSR where actions do not have names. Throughout the example, we will use the following notation: action names start with a capital letter, process names are in bold, event and resource names start with a lower case letter, and an overlined event name signifies sending the event.

**Example 4.1.1** A router is a node of a communication network, that receives data and tries to forward it to the next node in the network. The router originally sits idle. Once it receives data (indicated by the event $\text{ready}$), it reads it ($\text{Read}$), extracts its destination address ($\text{Prepare}$), tries to forward the message ($\text{Send}$), then signals an acknowledgement ($\text{ack}$). The router has five time units to produce $\text{ack}$ from the time $\text{ready}$ is signaled. We assume that the implementation of this abstract specification will use two cpu’s, $\text{cpu}_1$ and $\text{cpu}_2$, one buffer, $\text{buffer}$, an access table, $\text{accessTable}$, and a communication channel, $\text{channel}$. Our initial specification is given below.

**Router** = $\text{rec } X. (\text{Idle}^1 : X + \text{ready} \cdot \text{Read}^1 : \text{Prepare}^3 : \text{Send}^3 : \text{ack} \cdot X)$

where

- $\text{Idle} = \emptyset$
- $\text{Prepare} = \{\text{buffer, cpu}_1, \text{cpu}_2, \text{accessTable}\}$
- $\text{Read} = \{\text{buffer, cpu}_1\}$
- $\text{Send} = \{\text{buffer, cpu}_1, \text{cpu}_2, \text{channel}\}$

and $\text{Prepare}^3$ stands for the action $\text{Prepare}$ executed three consecutive time units.

The first refinement of **Router**, **Imp1**, shows how $\text{Prepare}$ is executed. It reformats the received message and checks an access table for permission to forward the received message. If the router has permission to forward the data, it fills some header information necessary to transmit the data; if it does not have permission, it drops the data and frees
the buffer. Reformatting the message and checking the permission can execute concurrently with synchronization of activities. In the ACSR process, this introduces a process, \( Q1 \), that replaces the abstract action \( \text{Prepare} \).

\[
\text{Imp1} = \text{rec } X. (\text{Idle}^1 : X + \text{ready}.\text{Read}^1 : Q1 \triangleq_3 (\text{NIL}, \text{Send}^1 : \text{ack}.X, \text{NIL}))
\]

\[
Q1 = (Q11 \parallel Q12)\{\text{granted}, \text{denied}\}
\]

\[
Q11 = \text{Format}^1 : \text{rec } Y. \ (\text{Idle}^1 : Y
+ \text{granted}.\text{FillHdr}^1 : \text{Wait}
+ \text{denied}.\text{Drop}^1 : \text{Wait})
\]

\[
Q12 = \text{GetPermission}^2 : (\text{granted}.\text{Wait} + \text{denied}.\text{Wait})
\]

\[
\text{Wait} = \text{rec } X. \text{Idle}^1 : X
\]

where

\[\text{Format} = \{\text{buffer, cpu}_1\} \quad \text{Drop} = \{\text{buffer, cpu}_1\} \]

\[\text{FillHdr} = \{\text{buffer, cpu}_1\} \quad \text{GetPermission} = \{\text{cpu}_2, \text{accessTable}\}\]

One should note that the refining process \( Q1 \), out of its current context, may execute for more than three time units, the duration of the refined action \( \text{Prepare} \). However, within the Router context, it is restricted to execute for at most three time units after which \( \text{Send} \) is executed. \text{Imp1} also shows resource redistribution and reduction refinement: the \( \text{GetPermission} \) step uses \( \text{cpu}_2 \) and the access table but does not need to use the buffer; filling the header and dropping the data steps each uses one \( \text{cpu} \) and the buffer but do not use the access table.

The second refinement, \text{Imp2}, implements a router that does not tolerate transmission failures. This is done by refining the \( \text{Send} \) action: if the message can be successfully transmitted, the router signals \( \text{success} \) and carries the send; if it can’t, the router executes an error-handling routine, locally signals \( \text{failure} \) and stops functioning. Also, to reflect the fact that information will be provided only about a successful message delivery, the acknowledgement is renamed to \( \text{success} \). \text{Imp2} introduces one local event, \( \text{failure} \), and reduces resources, as it is discovered that two \( \text{cpu} \)'s are not needed in the send step.
\textbf{Imp2} = rec X. (\text{Idle}^1 : X \\
+ ready.\text{Read}^1 : Q_1 \triangleq_3 (\text{NIL}, \\
\text{CarrySend}^1 : \text{success}.X \\
+ Error^1 : \text{failure}.\text{NIL}, \\
\text{NIL}))

where \text{CarrySend} = \{\text{channel}\} and \text{Error} = \{\text{cpu}\}.

\section*{4.2 Trace Refinement}

In this chapter we use a trace semantics for ACSR processes. (See Appendix D for details on how to generate the prefix-closed set of traces of an ACSR process). To develop a notion of refinement, we therefore first develop a notion of trace refinement and then extend it to sets of traces to give a formal semantics for ACSR refinement.

Let the set of ACSR non-\(\tau\) events be \(E\) and the set of ACSR timed actions be \(A\); let the events \(e, g, (a, p), (b, p'), \ldots\) range over the set \(E\) and actions \(A^v, B^u, \ldots\) range over the set of actions \(A\).

\textbf{Definition 4.2.1} A trace is a possibly infinite string over the set \((E \cup A)\). The Greek letter \(\epsilon\) denotes empty string. For a finite trace \(r\) and a trace \(s\), \(rs\) denotes concatenation of the traces \(r\) and \(s\). The function \(\text{events}(s)\) returns the set of event labels in the trace \(s\). Note that \(\tau\) does not occur in any trace since it is an internal action while a trace describes observable actions.

For example, \((e, 1) (a, 3), (e, 1) ((r_1, 1))^3 (a, 2), \emptyset ((r_1, 1), (r_2, 2))^3 ((r_1, 1))^2 ((r_1, 1))^2 \ldots\) are ACSR traces where \(a, b, e\) are non-\(\tau\) event labels, and \(r_1\) and \(r_2\) are resources.

We next introduce two notions of trace refinement that only differ in the way resource usage is handled in related traces. The first notion of trace refinement allows the refining trace to use fewer resources. The second notion of trace refinement allows the refining trace to use more resources.

\textbf{Definition 4.2.2} Let \(r\) and \(s\) be two traces, the event relabeling function \(\rho : \text{events}(s) \rightarrow \text{events}(r)\), and let the set \(\text{Local} = \text{events}(r) - \text{range}(\rho)\). Then, we say \(r\) refines \(s\) under \(\rho\).
In the first, a designer can start with a loose estimate of the resource usage and they then

Informally, a trace \( r \) refines a trace \( s \) under a given relabeling function \( \rho \) if \( r \) and \( s \) are

the empty string \( \epsilon \) (1), or the events and actions in \( r \) are matched with events and actions

in \( s \) as follows. Each event in \( r \) is either a “local” event (2), or corresponds through \( \rho \) to an

event in \( s \) that happens at the same time (3). The event correspondence ensures that each

event in \( s \) is represented by an event in \( r \) and that the order of events in \( s \) is preserved in \( r \).

Each action in \( r \) corresponds to an action (or possibly a sequence of actions) in \( s \) with the

same duration, and the resources used in the \( r \)-action are a subset of the resources used in the

\( s \)-action—conditions (4) through (6).

The next notion of trace refinement differs from the above in relating the timed actions

only (conditions (4) through (6)): an action in the refining trace can use more resources.

**Definition 4.2.3** Let \( r \) and \( s \) be two traces, the event relabeling function \( \rho : events(s) \rightarrow

events(r) \), and let the set \( Local = events(r) - range(\rho) \). Then, we say \( r \) refines \( s \) under \( \rho \)

\((r \leq_{\rho} s)\) if one of conditions (1)-(6) holds:

\[
\begin{align*}
    r &= s = \epsilon \\
    r &= (a, p) r' \wedge a \in Local \wedge r' \leq_{\rho} s \\
    r &= (a, p) r' \wedge s = (b, p) s' \wedge a = \rho(b) \wedge r' \leq_{\rho} s' \\
    r &= A^u r' \wedge s = B^u s' \wedge A \subseteq B \wedge r' \leq_{\rho} s' \\
    r &= A^u r' \wedge s = B^u s' \wedge u < v \wedge A \subseteq B \wedge A^u r' \leq_{\rho} s' \\
    r &= A^u r' \wedge s = B^u s' \wedge v < u \wedge A \subseteq B \wedge A^{u-v} r' \leq_{\rho} s' \\
\end{align*}
\]

Practically, the two notions will allow us to provide two types of design methodologies:

In the first, a designer can start with a loose estimate of the resource usage and they then

69
tighten the estimate as more details are known about the system. In the second, a designer can ignore resource usage at the beginning and later introduce it as more details are available. Theoretically, the two notions of trace refinement do not have major differences. We therefore focus on the first relation $\leq_{\rho}$. In other words, whenever we mention trace refinement, we mean the first relation $\leq_{\rho}$. However, unless explicitly stated, all definitions and facts for the trace refinement relation $\leq_{\rho}$ also hold for the second trace refinement relation $\leq_{\rho}$.

**Definition 4.2.4** *(Trace refinement)* For every trace $r$ and $s$, we say $r$ refines $s$ ($r \leq s$) if there exists a function $\rho : events(s) \rightarrow events(r)$ such that $r \leq_{\rho} s$.

As an example, the process **Router** of the previous section has the trace

$$s = \text{ready}\{buffer, cpu_1\}\{buffer, cpu_1, cpu_2, accessTable}\{buffer, cpu_1, cpu_2, \text{channel}\}\text{ack},$$

and the process **Imp2** has the trace

$$r = \text{ready}\{buffer, cpu_1\}\{buffer, cpu_1, cpu_2, accessTable\}\{cpu_2, accessTable\}\{buffer, cpu_1\}\{channel\}\text{success}.$$  

For the relabeling function $\rho(\text{ready}) = \text{ready}$ and $\rho(\text{ack}) = \text{success}$, we have $r \leq_{\rho} s$ and thus $r$ refines $s$.

Trace refinement has four properties: 1) it is a pre-order relation; 2) it relates traces with equal duration; 3) it preserves the timed occurrence of related events; and 4) it preserves the order of events.

**Property 2** Trace refinement is a pre-order.

**Proof:** It is trivial to verify that using the identity function over events $Id$, we have $s \leq_{Id} s$ for every trace $s$; thus trace refinement is reflexive. To prove that $\leq$ is transitive, let $s_3 \leq r_2$ and $r_2 \leq s_1$; then (by definition) there exist functions $\rho_2 : events(s_2) \rightarrow events(s_3)$ and $\rho_1 : events(s_1) \rightarrow events(s_2)$ such that $s_3 \leq_{\rho_2} s_2$ and $s_2 \leq_{\rho_1} s_1$. It is easy to verify that $s_3 \leq_{\rho} s_1$ where the function $\rho : events(s_1) \rightarrow events(s_3)$ is the composition of $\rho_1$ and $\rho_2$; that is, $\rho(a) = \rho_2(\rho_1(a))$ for each $a \in events(s_1)$. Thus, we have $s_3 \leq s_1$. □

To state the other properties of trace refinement formally, we introduce the following operations on traces.
**Definition 4.2.5** The function \( \text{event}(s, t) \) returns the sequence of events in \( s \) that occur at time \( t \) for \( 0 \leq t \leq \text{dur}(s) \) and returns the empty sequence \( \epsilon \) for other values of \( t \). For a trace of events only \( \alpha \), the predicate \( \alpha \subseteq \text{event}(s, t) \) is true if the events in the event sequence \( \alpha \) occur in the same order in the sequence \( \text{event}(s, t) \), with \( \text{event}(s, t) \) possibly having additional events that interpose the events of \( \alpha \).

The function \( \text{dur}(s) \) returns the duration of the trace \( s \):

\[
\text{dur}(s) = \begin{cases} 
0 & \text{if } s = \epsilon \\
\text{dur}(s') & \text{if } s = \epsilon s' \\
\alpha + \text{dur}(s') & \text{if } s = A^\alpha s'
\end{cases}
\]

The function \( \text{time}(s, e) \) returns the set of times when the event \( e \) occurs in \( s \):

\[
\text{time}(s, e) = \{ t \mid 0 \leq t \leq \text{dur}(s) \land e \subseteq \text{event}(s, t) \}.
\]

\( \square \)

**Property 3** If \( r \) refines \( s \), then \( \text{dur}(r) = \text{dur}(s) \).

**Property 4** If \( r \) refines \( s \), then for each function \( \rho : \text{events}(s) \rightarrow \text{events}(r) \) such that \( r \leq_\rho s \), the following property holds:

\[
\forall a \in \text{events}(s) \forall p : t \in \text{time}(s, (a, p)) \leftrightarrow (\rho(a), p) \subseteq \text{event}(r, t).
\]

Informally: the timed occurrence of an event in \( s \) is represented through the occurrence of its refining event in \( r \) at the same time.

The above property can be generalized to a sequence of events. This is the topic of the next property where we use the notation \( \rho(a) \) to denote the sequence obtained by applying the function \( \rho \) to each event in the event sequence \( a \).

**Property 5** If \( r \) refines \( s \), then for each function \( \rho : \text{events}(s) \rightarrow \text{events}(r) \) such that \( r \leq_\rho s \), we have for any event sequence \( \alpha \) the following property holds:

\[
\forall 0 \leq t \leq \text{dur}(s). (\alpha \subseteq \text{event}(s, t) \iff \rho(\alpha) \subseteq \text{event}(r, t)).
\]

A few comments about the usefulness of these properties are in order. Transitivity allows a system to be refined in a stepwise manner while guaranteeing that the final system description is also a refinement of the original specification. Preserving the timed occurrence of original events is important for maintaining the validity of properties such as safety properties. That is, if a property expressed in terms of original events is proven about the original system, its validity is also guaranteed in the refined system.
Extending Trace Refinement to Sets of Traces. To use trace refinement as a formal semantics for ACSR refinement we must extend it to a relation over sets of traces which represent the behaviors of ACSR processes. During this extension, we should address two concerns. One is that the trace ordering and the trace-set ordering are consistent; that is, if a trace \( r \) refines a trace \( s \) in the trace ordering, then the singleton trace set \( \{r\} \) also refines the singleton trace set \( \{s\} \). The second concern is that the trace-set ordering is compositional with the ACSR operators. This will allow us to refine a process, i.e., its set of traces, by refining its subcomponents.

The first concern is automatically resolved by adopting an extension methodology from the powerdomain theory [Smy87, Gun92]. The three most commonly used extensions in powerdomain theory are the Hoare, Smyth and Egli-Milner orderings. For sets of traces \( R \) and \( S \), and for a function \( \rho : \text{events}(S) \rightarrow \text{events}(R) \), these orderings are defined as follows:

1. Hoare ordering: \( \leq^1_\rho S \) if and only if \( \forall r \in R : \exists s \in S : r \leq_\rho s \)
2. Smyth ordering: \( \leq^1_\rho S \) if and only if \( \forall s \in S : \exists r \in R : r \leq_\rho s \)
3. Egli-Milner ordering: \( \leq^1_\rho S \) if and only if \( R \leq^1_\rho S \) and \( R \leq_\rho S \).

In the case of our trace refinement relation \( \leq_\rho \), we would like to use the extended relation as an “implementation” relation; that is, \( R \) is considered to be an “implementation” of the “specification” \( S \). There are two common notions of an acceptable implementation relation on the trace semantics, and which we develop in the remainder of this chapter: 1) an implementation contains only behaviors that are related to behaviors of the specification—the Hoare ordering; and 2) an implementation contains all and only those behaviors that are related to behaviors of the specification—the Egli-Milner ordering. In addition, we would like the implementation to rename specification events and use fewer resources. We will therefore examine these two extensions as possible notions of process refinement.

**Definition 4.2.6** Trace-set refinement, \( \leq^1 \) (\( \leq^1_\rho \)), is a relation over sets of traces defined as follows: for all sets of traces \( R \) and \( S \), \( R \leq^1 S \) (\( R \leq^1_\rho S \)) if there exists a function \( \rho : \text{events}(S) \rightarrow \text{events}(R) \) such that \( R \leq^1_\rho S \) (\( R \leq^1_\rho S \)).

To reduce the complexity of refining a large system, it is important to be able to break it into components and refine the components in a modular fashion. That is, it is important
that whichever definition of trace-set refinement is adopted be compositional with the ACSR operators. In the definition for compositionality that follows, $\leq^o$ denotes either $\leq^1$ or $\leq^1$.

**Definition 4.2.7** We say trace-set refinement $\leq^o$ is *compositional* if for all processes $P$ and $Q$ and for every context $C$, $\text{traces}(P) \leq^o \text{traces}(Q) \Rightarrow \text{traces}(C[P]) \leq^o \text{traces}(C[Q])$.

**ACSR Process Refinement.** Since the semantics of ACSR processes is given in terms of sets of traces, two ACSR processes are related if and only if their sets of traces are related by a trace-set refinement. However, in the case of large systems, using this notion of process refinement is impractical to verify since most systems have large or infinite sets of traces. We therefore provide an alternative approach that is based on a set of *transformation rules* on process terms. The rules themselves depend on which trace-set ordering is used. Each transformation rule rewrites a specification process to an implementation process. Each transformation rule must be *sound* with respect to a trace-set refinement; that is, if process $Q$ can be rewritten to process $P$, then $\text{traces}(P) \leq^o \text{traces}(Q)$.

An interesting question is whether a set of transformation rules is complete; that is, if whenever two processes $P$ and $Q$ have sets of traces that are related by trace-set refinement, the set of transformation rules can be used to transform one process to the other or, more generally, transform it to an "equivalent" process.

With this background, we now turn our attention to the specifics of ACSR process refinement in the Hoare and Egli-Milner orderings. In the next section, we examine refinement in the *unprioritized* semantics. In Section 4.4, we discuss the effects of priorities on the two trace-set orderings.

### 4.3 ACSR Process Refinement in the Hoare Ordering

As discussed in the previous section, our goal is to develop a set of sound and complete transformation rules over ACSR processes for a given trace-set ordering. In this section, we first focus our attention on the Hoare ordering trace-set refinement, $\leq^1$, in the *unprioritized* semantics. We show that it is compositional with the ACSR operators under reasonable restrictions, and define its set of transformation rules. We then show that this set of transformation rules is sound and complete in the case of finite processes. Finally, we
discuss the differences in the sets of transformation rules between the Hoare and Egli-Milner orderings.

### 4.3.1 Compositionality

Trace refinement, $\leq^1$, is compositional with ACSR operators under certain restrictions on the relabeling function $\rho$ that is used in the trace refinement. The restrictions are due to the fact that trace refinement allows the refining process, $P$, to have local events. We define the set of events, and the set of local events for a process $P$ under a given relabeling function $\rho$ as follows.

**Definition 4.3.1** The set of events used in a process $P$, $\text{events}(P)$, is defined inductively as follows:

- $\text{events}(\text{NIL}) = \emptyset$
- $\text{events}(a, P) = \{a\} \cup \text{events}(P)$
- $\text{events}(A^i : P) = \text{events}(P)$
- $\text{events}(P \setminus F) = F \cup \text{events}(P)$
- $\text{events}([P]_i) = \text{events}(P)$
- $\text{events}(P_1 + P_2) = \text{events}(P_1) \cup \text{events}(P_2)$
- $\text{events}(P_1 \parallel P_2) = \text{events}(P_1) \cup \text{events}(P_2)$
- $\text{events}(P_1 \Delta_i (P_2, P_3, P_4)) = \{b\} \cup \bigcup_{i=1}^{4} \text{events}(P_i)$
- $\text{events}(\text{rec } X. P^\rho) = \text{events}(P^\rho)$
- $\text{events}(X) = \emptyset$

**Definition 4.3.2** The set $\text{Local}(P, \rho)$ of local events of a process $P$ under a relabeling function $\rho : \text{Evts} \rightarrow \text{Evts}$ is defined as follows:

$$\text{Local}(P, \rho) = \text{events}(P) - \text{range}(\rho).$$

The restrictions which are required for compositionality of trace-set refinement are expressed in terms of a notion of “compatibility” between the processes $P$ and $Q$ and the context $C[.]$ in which they could be inserted. Next, a context is represented as a process $E$.  

74
Definition 4.3.3 Given processes \( P, Q, \) and \( E, \) and given a relabeling function \( \rho : \text{events}(Q) \rightarrow \text{Events}, \) we say that \( P \) and \( Q \) are compatible with \( E \) under \( \rho \) if

\[
\forall a \in (\text{events}(Q) \cap \text{events}(E)) : \rho(a) = a \quad \text{and} \quad \text{Local}(P, \rho) \cap \text{events}(E) = \emptyset
\] (4.1)

\[
\text{Local}(P, \rho) \cap \text{events}(E) = \emptyset
\] (4.2)

Condition (4.1) says that \( \rho \) does not rename events that are common between \( Q \) and \( E. \) This ensures that, when \( Q \) and \( E \) are combined through the Choice, Parallel or Scope operators, the function \( \rho \) can be extended with the identity over the events of \( E \) to a function that maps \( \text{events}(Q) \cup \text{events}(E) \) to \( \text{events}(P) \cup \text{events}(E). \) This extended \( \rho \) function is used to prove that \( P \) combined with \( E \) refines \( Q \) combined with \( E. \)

Condition (4.2) ensures that local events of \( P \) remain local in the process \( P \) combined with \( E. \) Without this condition, trace-set refinement in the Hoare ordering may not be compositional. To illustrate the necessity of condition (4.2), consider the following example where we use the set notation for a function: Let \( e_1 = (a_1, p_1) \) and \( e_2 = (a_2, p_2) \) with \( a_1 \neq a_2; \) we have

\[
\text{traces}(e_1, e_2, \text{NIL}) \leq^1_{ \{ (a_2, a_2) \} } \text{traces}(e_2, \text{NIL})
\]

with \( a_1 \in \text{Local}(e_1, e_2, \text{NIL}, \{ (a_2, a_2) \}). \) However,

\[
\text{traces}(e_1, e_2, \text{NIL} + e_1, \text{NIL}) \not\leq^1_{ \{ (a_2, a_2), (a_1, a_1) \} } \text{traces}(e_2, \text{NIL} + e_1, \text{NIL})
\]

because the trace \( e_1 e_2 \in \text{traces}(e_1, e_2, \text{NIL} + e_1, \text{NIL}) \) can not be mapped to any trace in \( \text{traces}(e_2, \text{NIL} + e_1, \text{NIL}) \). This violation of the Hoare ordering is due to the fact that \( a_1 \in \text{Local}(e_1, e_2, \text{NIL}, \{ (a_2, a_2) \}) \cap \text{events}(e_1, \text{NIL}). \)

**More notation.** In the remainder of this chapter, we will write \( P \leq^1_\rho Q \) as a short hand notation for \( \text{traces}(P) \leq^1_\rho \text{traces}(Q). \) Also, for given ACSR processes \( E_1, \ldots, E_n, \) the function \( \text{Id}_{E_1 \ldots E_n} \) denotes the identity function over the events of \( E_1, \ldots, E_n; \) that is,

\[
\forall a \in (\text{events}(E_1) \cup \ldots \cup \text{events}(E_n)) : \text{Id}_{E_1 \ldots E_n}(a) = a
\]

In addition, to simplify the notation, since we will be examining refinement in the unprioritized semantics, we overload the syntax of the relabeling function and write \( \rho(e) \) for an event \( e = (a, p) \) instead of \( (\rho(a), p); \) we also write \( \pi \) for \( (\pi, p'). \)
Theorem 4.3.1 If $P \preceq^l Q$, then

1. $A^t : P \preceq^l A^t : Q$

2. if $P$ and $Q$ are compatible with $e.NIL$ under $\rho$ then
   $e.P \preceq^l \rho \cup \{e\} \cup e.Q$

3. if $P$ and $Q$ are compatible with $E$ under $\rho$ then
   (a) $(P + E) \preceq^l (Q + E)$
   (b) $(E + P) \preceq^l (E + Q)$

4. $(P \setminus \rho(F)) \preceq^l (Q \setminus F)$ if $F \subseteq \text{events}(Q)$ where $\rho(F) = \{\rho(e) : e \in F\}$

5. $[P].t \preceq^l [Q].t$

6. if $P$ and $Q$ are compatible with $E_i$ under $\rho$ for $i = 1, \ldots, 4$ and $E_4 = e.NIL$ and $\rho' = \rho \cup Id_{E_1 \ldots E_3}$ then
   (a) if $\rho'(\pi) = \overline{\pi}(e)$ and $\rho'(\{L \cup \mathcal{L}\})$ is one-one, where $L = \text{events}(Q) \cup \{e\}$, then
      $P \Delta^\prime_{\rho'} (E_1, E_2, E_3) \preceq^l Q \Delta^\prime_{\rho'} (E_1, E_2, E_3)$
   (b) $E_1 \Delta^\prime_{\rho'} (P, E_2, E_3) \preceq^l E_1 \Delta^\prime_{\rho'} (Q, E_2, E_3)$
   (c) $E_2 \Delta^\prime_{\rho'} (E_1, P, E_3) \preceq^l E_2 \Delta^\prime_{\rho'} (E_1, Q, E_3)$
   (d) $E_3 \Delta^\prime_{\rho'} (E_1, E_2, P) \preceq^l E_3 \Delta^\prime_{\rho'} (E_1, E_2, Q)$

7. $(\text{rec}X.P) \preceq^l (\text{rec}X.Q)$

Proof: See Appendix B.1. \qed

To have compositionality with the event prefix (1b), Choice (2), and Scope (6) operators, we restrict the processes $P$ and $Q$ to be compatible with the processes $e.NIL$, $E$, and $E_1, \ldots, E_4$ under $\rho$, respectively. Consider the following example that shows the necessity of this restriction in the case of the Choice operator; similar examples can be constructed for the event prefix and Scope operators. We have

$e_1.e_2.NIL \preceq^l e_2.NIL$

where $\rho = \{(f, f)\}$. Let the process $E = e_1.NIL$ which shares the event $e_1$ with the local events $\text{Local}(e_1.e_2.NIL, \rho)$. The extension of the function $\rho$ is

$\rho \cup Id_E : \text{events}(e_2.NIL + e_1.NIL) \rightarrow \text{events}(e_1.e_2.NIL + e_1.NIL).$
This function however makes $e_1$ a non local event in $e_1.e_2.NIL + e_1.NIL$, and thus

$$e_1.e_2.NIL + e_1.NIL \not\preceq_{\rho \cup Id_E} e_2.NIL + e_1.NIL$$

because the trace $e_1 e_2$ does not refine any trace of $e_2.NIL + e_1.NIL$ using $\rho \cup Id_E$.

Compositionality with the Scope operator in its first argument requires two additional restrictions on the function $\rho'$ which extends $\rho$ over the events of the context: The function $\rho'$ must preserve the complement of $e$ and it must map the events $events(Q) \cup \{e\}$ and their complements in a one-to-one fashion. The first restriction ensures that any potential end of scope synchronization in $Q$ is preserved in $P$. The second restriction disallows unexpected end of scope synchronization in $P$. These restrictions will be addressed further in Lemma 4.3.1.

Another comment about Theorem 4.3.1 is that trace-set refinement is not in general compositional with the Parallel operator. This is because the process $Q$ when run in parallel with another process $E$ may lead to a deadlock due to resource sharing. However, refinement may release those resources in process $P$ (which refines $Q$); therefore $P \parallel E$ would not deadlock and produce more traces, a violation of the Hoare ordering. In the restricted case where $Q$ does not share resources with $E$, trace-set refinement is compositional with the Parallel operator. We state this result in Lemma 4.3.1, and separate it from Theorem 4.3.1 because the restriction is stronger than the restrictions for the other operators. In addition, there is no simpler, syntactic way of characterizing potential deadlocks in a parallel process.

Before stating Lemma 4.3.1, we need to define the set of resources of a process. In the definition, we use $res(A)$ to denote the set of resources in the action $A$.

**Definition 4.3.4** The set of resources of a process $P$, $res(P)$, is defined as follows:

- $res(NIL) = \emptyset$
- $res(e.P') = res(P')$
- $res(A' : P') = res(A) \cup res(P')$
- $res(P' \setminus F) = res(P')$
- $res([P']_I) = I \cup res(P')$

77
Lemma 4.3.1 If $P \leq_{\rho}^1 Q$, then for every process $E$ such that $P$ and $Q$ are compatible with $E$ under $\rho$,

if $\text{res}(Q) \cap \text{res}(E) = \emptyset$ and  

$\rho \cup \text{Id}_E$ is a one-to-one, complement-preserving over $L \cup \overline{L}$,  

where $L = \text{events}(Q) \cup \text{events}(E)$, then  

(a) $(P \parallel E) \leq_{\rho \cup \text{Id}_E}^1 (Q \parallel E)$  

(b) $(E \parallel P) \leq_{\rho \cup \text{Id}_E}^1 (E \parallel Q)$

Proof: Appendix B.2 contains the proof for (a); the proof for (b) is similar. □

The additional restriction on the set of events of $E$, that $\rho \cup \text{Id}_E$ be one-to-one, complement-preserving function preserves potential communication between $P$ and $E$ and disallows “unexpected” synchronization between $P$ and $E$. As an example of unexpected synchronization, consider

$Q \parallel E = \overline{e_1}, e_2, \text{NIL} \parallel e_3, \text{NIL}$ and $P \parallel E = \overline{e_5}, e_2, \text{NIL} \parallel e_3, \text{NIL}$

and let $\rho(e_2) = e_2$, $\rho(\overline{e_1}) = \overline{e_3}$ and $\rho(e_3) = e_3$. Then, we have

$P \leq_{\rho}^1 Q$, but $P \parallel E \not\leq_{\rho} Q \parallel E$

since the trace $e_2 \in \text{traces}(P \parallel E)$ does not refine any trace in $\text{traces}(Q \parallel E)$. This problem is due to the synchronization between $e_3$ and $\overline{e_3}$ (refining $\overline{e_1}$), which is unexpected in $Q \parallel E$ since $e_3$ and $\overline{f_1}$ can not synchronize. This problem is eliminated when $\rho$ is restricted to be a one-to-one function and to preserve complements, i.e. $\overline{\rho(e)} = \rho(\overline{e})$.

We note that in defining trace refinement, we needed to define the relabeling function $\rho$ to map events between related traces but did not require such a mapping between actions. The notion of action refinement over traces is defined as resource set inclusion through conditions (4)-(6) of Definition 4.2.2.
We also note that in the case of the second notion of trace refinement, \( \preceq \), where the refining trace can introduce new resources, compositionality with the Parallel operator is obtained without any condition on the resources used. The proof is straightforward.

### 4.3.2 Refinement Transformation Rules

The above notion of trace-set refinement can be captured over the syntax of ACSR processes by the following transformations: A process \( Q \) is refined to a process \( P \) by relabeling the events of \( Q \), introducing new events, and replacing the actions of \( Q \) with processes. We define these transformations by an auxiliary transformation relation, \( ref \), that is defined recursively on the structure of the process \( Q \). The transformation relation \( ref \) uses a function \( \rho_1 \) that describes how the events of \( Q \) are relabeled and a relation \( \rho_2 \) that describes how actions of \( Q \) can be refined to processes. (The set of actions of a process \( Q \), \( actions(Q) \) can be recursively defined in a similar way to \( events(Q) \).)

An important property of the transformation rules is their soundness with respect to trace-set refinement. This is a relatively easy property to enforce. A desirable property of the transformations is their completeness with respect to trace-set refinement. This, however, is rather not a straightforward property to enforce. In particular, since in the Hoare ordering trace-set refinement is not in general compositional with the Parallel operator, we need to find a way around the restrictive condition that the subprocesses do not share resources. For this, we use the Close operator to ensure that the transformation rules rewrite a process \( Q_1 \parallel Q_2 \) by rewriting \( Q_1 \) and \( Q_2 \) such that the transformed \( Q_1 \) and \( Q_2 \) do not release shared resources. We therefore define the auxiliary transformation relation \( ref \) in two steps. First, we define a transformation relation \( cref \) that takes as an argument a set of resources that must not be released. We then use this transformation relation to define \( ref \).

#### Definition 4.3.5

Given a function \( \rho_1 : \text{Evts} \rightarrow \text{Evts} \), and a relation \( \rho_2 \subseteq \text{Act} \times \text{Proc} \); the resource-closed transformation relation \( cref^1_{(\rho_1, \rho_2)} \subseteq (\text{Proc} \times R) \times \text{Proc} \) is defined as follows:

1. \( cref^{1}_{(\rho_1, \rho_2)}(Q, I) \leadsto \text{NIL} \)
2. \( cref^{1}_{(\rho_1, \rho_2)}(Q, I) \leadsto \sum_{\ell \in L} cref^1_{(\rho_1, \rho_2)}(Q, I) \) for finite \( L \), \( L \cap \text{range}(\rho_1) = \emptyset \)
(3) \( \text{cref}_{(p_1, p_2)}^{\delta}(e, Q, I) \sim \rho_1(e) \text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \)

(4) \( \text{cref}_{(p_1, p_2)}^{\delta}(A^t : Q, I) \sim [R]_t \triangleq (\text{NIL}, \text{cref}_{(p_1, p_2)}^{\delta}(Q, I), \text{NIL}) \)
for \((A^t, R) \in p_2\) and \(I' = I \cap (\text{res}(A) - \text{res}(R))\)

(5) \( \text{cref}_{(p_1, p_2)}^{\delta}(Q||Q_2, I) \sim \text{cref}_{(p_1, p_2)}^{\delta}(Q_1, I) || \text{cref}_{(p_1, p_2)}^{\delta}(Q_2, I') \)
for \(I' = I \cup (\text{res}(Q_1) \cap \text{res}(Q_2))\)

(6) \( \text{cref}_{(p_1, p_2)}^{\delta}(Q_1 + Q_2, I) \sim \text{cref}_{(p_1, p_2)}^{\delta}(Q_1, I) + \text{cref}_{(p_1, p_2)}^{\delta}(Q_2, I) \)

(7) \( \text{cref}_{(p_1, p_2)}^{\delta}(Q \triangleq (R, S, T), I) \sim \text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \triangleq (\text{cref}_{(p_1, p_2)}^{\delta}(R, I), \text{cref}_{(p_1, p_2)}^{\delta}(S, I), \text{cref}_{(p_1, p_2)}^{\delta}(T, I)) \)

(8) \( \text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \sim \text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \backslash F \)

(9) \( \text{cref}_{(p_1, p_2)}^{\delta}(Q \backslash F, I) \sim \text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \backslash \rho_1(F) \)

(10) \( \text{cref}_{(p_1, p_2)}^{\delta}([Q]_v, I) \sim [\text{cref}_{(p_1, p_2)}^{\delta}(Q, I)]_v \)
where \((U' - (\text{res}(\text{cref}_{(p_1, p_2)}^{\delta}(Q, I)) - \text{res}(Q))) \subseteq U'\)

(11) \( \text{cref}_{(p_1, p_2)}^{\delta}(\text{rec} X.Q, I) \sim \text{rec} X.\text{cref}_{(p_1, p_2)}^{\delta}(Q, I) \)

(12) \( \text{cref}_{(p_1, p_2)}^{\delta}(X, I) \sim X \)

\[\square\]

Note that in the above transformation rules since \(p_2\) is a relation, different occurrences of an action can be refined to different processes. The set of resources \(I\) represents those resources that can not be released. It is essentially used in Rules (4) and (5) and indirectly in Rule (10). In Rule (4), an action-prefix process is transformed. Since the process \(R\) which refines the action \(A^t\) may release some resources, the Close operator is used so that only those resources not listed in \(I\) can be released. In Rule (5), the set of resources \(I\) is augmented by those resources shared between the two processes running in parallel. Thus, any refinement of these processes will not release shared resources and eliminate any deadlock from the abstract process. In Rule (10), which deals with the Close operator, the refining process can release those resources that are neither in \(I\) nor in the original closed process \(Q\). Note that in this rule refinement can close more resources than in the abstract process.

An on the side note about Rule (8) is that when we use the above transformation rules
along with an equivalence relation (e.g., bisimulation), Rule (8) can be derived from Rule (1) after restructuring the abstract process to an equivalent process. We include it in the set of transformation rules for practical reasons; this rule is often used as we show through the examples in this chapter and chapter 6.

We next use the resource-closed transformation rules to define the transformation rules for the Hoare ordering extension of trace refinement.

**Definition 4.3.6** Let \( Q \) be a process, let the function \( \rho_1 : \text{events}(Q) \rightarrow \text{Evts} \) be one-to-one, complement preserving, and let the relation \( \rho_2 \subseteq \text{actions}(Q) \times \text{Proc} \) satisfy for each \((A^i, R) \in \rho_2\)

\[
(\text{events}(R) \cap \text{range}(\rho_1) = \emptyset) \land (\text{res}(R) \subseteq \text{res}(A)).
\]

Then, \( Q \) can be transformed using the following transformation scheme

\[
\text{ref}_{(\rho_1, \rho_2)}^1(Q) = \text{cref}_{(\rho_1, \rho_2)}^1(Q, \emptyset).
\]

Condition (4.3) states that an action is refined to a process that may contain "local" (new) events only and that uses at most the same resources as the refined action\(^1\). Note that, in this ordering, a process that refines an action does not have any restriction on its duration. The transformation rules ensure that the process executes for at most the same duration as the refined action through the Scope operator—transformation (4). A shorter process is accepted in the Hoare ordering due to the fact that, in this ordering, any specification can be refined to the NIL process—transformation (1). While this seems an unreasonable notion of implementation in the extreme case, it however allows designers to reduce behaviors from an abstract specification for instance to reduce non-determinism and to eliminate unimplementable behaviors.

Note also by defining \( \text{ref} \) in terms of the resource-closed transformation relation \( \text{cref} \), we can rewrite a parallel process by rewriting its sub-processes. In addition, we ensure that each of the transformed sub-processes will not release any shared resource. Thus, any potential deadlock within the original process will not be released in the transformed process.

In the sequel, whenever we say "a process \( Q \) is transformed to process \( P' \), we mean \( Q \) is transformed using a transformation scheme \( \text{ref} \) of Definition 4.3.6.\(^2\)

---

\(^1\)In the case of the second trace refinement relation \( \preceq \), condition (4.3) differs in the direction of the resource inclusion only; that is, we have \( \text{res}(A) \subseteq \text{res}(R) \).
Example 4.3.1 In Example 4.1.1, recall that the original specification process \texttt{Router} was refined to the process \texttt{Imp1} by showing how the action \texttt{Prepare} was executed. We can transform the process \texttt{Router} to the process \texttt{Imp1} as follows: First, we use the compositionality result and hence focus on refining the process fragment of \texttt{Router} which contains the action \texttt{Prepare}. The process fragment to be refined is

\[ Q \overset{\text{def}}{=} \text{Prepare} : \text{Send} : \text{ack} \times . \]

Second, we can use the transformation rule (4) to refine the process \( Q \) where the function \( \rho_1 \) is the identity function over events and the relation \( \rho_2 \) defined as follows:

\[ \rho_2 = \{ (\text{Prepare}, Q), (\text{Send}^1, \text{Send}^1 : \text{NIL}) \} \]

We then can apply the transformation rules on \( Q \) as follows:

\[
\begin{align*}
\text{ref}_{(\rho_1, \rho_2)}(Q) & \leadsto Q1 \triangle_3 (\text{NIL}, \text{ref}_{(\rho_1, \rho_2)}(\text{Send}^1 : \text{ack} \times \text{NIL}), \text{NIL}) \quad \text{by rule (4)} \\
& \leadsto Q1 \triangle_3 (\text{NIL}, \text{Send}^1 : \text{NIL} \triangle_1 (\text{NIL}, \text{ref}_{(\rho_1, \rho_2)}(\text{ack} \times \text{NIL}), \text{NIL}), \text{NIL}) \quad \text{by rule (4)} \\
& \leadsto Q1 \triangle_3 (\text{NIL}, \text{Send}^1 : \text{ref}_{(\rho_1, \rho_2)}(\text{ack} \times \text{NIL}), \text{NIL}) \\
& \leadsto Q1 \triangle_3 (\text{NIL}, \text{Send}^1 : \rho_1(\text{ack}).\text{ref}_{(\rho_1, \rho_2)}(X), \text{NIL}) \quad \text{by rule (3)} \\
& \leadsto Q1 \triangle_3 (\text{NIL}, \text{Send}^1 : \text{ack} \times \text{NIL}, \text{NIL}) \quad \text{by rule (11)} \\
\overset{\text{def}}{=} & \texttt{Imp1}
\end{align*}
\]

We can further refine process \texttt{Imp1} to a process where the router sends an acknowledgement only when it successfully forwards the message; otherwise, the router executes an error recovery, signals a failure then stops. Again we make use of the compositionality of refinement and focus on applying the transformation steps the process fragment of \texttt{Imp1}

\[ Q' \overset{\text{def}}{=} \text{Send} : \text{ack} \times . \]

Let the relabeling function be defined as \( \rho_1(\text{ack}) = \text{success} \) and the action-refinement relation be

\[ \rho_2 = \{(\text{Send}^1, \text{CarrySend}^1 : \text{NIL}), (\text{Send}^1, \text{Error}^1 : \text{NIL})\} \]

We then apply the transformations as follows:
One point to make here is that the fact that our action refinement is a relation allowed us to refine the two occurrences of the action Send$^1$ differently. Another interesting point to make here is that our transformation rules do not allow us to transform the router to a faulty router that would keep functioning even if it does not manage to forward a message. In such an implementation, we would have the process fragment $Q'$ transformed as follows:

\[
\text{ref}^i_{(p_1,p_2)}(Q') \sim \text{ref}^i_{(p_1,p_2)}(Q' + Q')
\]

\[
\sim \text{ref}^i_{(p_1,p_2)}(Q') + \text{ref}^i_{(p_1,p_2)}(Q')
\]

\[
\equiv \text{ref}^i_{(p_1,p_2)}(\text{Send}^1 : \text{ack}.X) + \text{ref}^i_{(p_1,p_2)}(\text{Send}^1 : \text{ack}.X)
\]

\[
\sim \text{CarrySend}^1 : \text{NIL} \triangle_1 (\text{NIL}, \text{ref}^i_{(p_1,p_2)}(\text{ack}.X), \text{NIL})
\]

\[
+ \text{Error}^1 : \text{NIL} \triangle_1 (\text{NIL}, \text{ref}^i_{(p_1,p_2)}(\text{ack}.X), \text{NIL})
\]

\[
\sim \text{CarrySend}^1 : \text{ref}^i_{(p_1,p_2)}(\text{ack}.X) + \text{Error}^1 : \text{ref}^i_{(p_1,p_2)}(\text{ack}.X)
\]

\[
\sim \text{CarrySend}^1 : \rho_1(\text{ack}_1).\text{ref}^i_{(p_1,p_2)}(X) + \text{Error}^1 : \text{failure}.\text{ref}^i_{(p_1,p_2)}(\text{ack}.X)
\]

by rules (3) and (2)

\[
\sim \text{CarrySend}^1 : \text{success}.X + \text{Error}^1 : \text{failure}.\text{NIL}
\]

by rules (12) and (1)

\[
\equiv \text{Imp2}
\]

Theorem 4.3.2 (Soundness of transformation rules.) For all processes $P$ and $Q$,

\[
\text{if } \text{ref}^i_{(p_1,p_2)}(Q) \sim P, \text{ then } P \leq^i_{p_1} Q.
\]

Proof: The proof is by induction on the structure of $Q$ and uses the compositionality of trace refinement with the ACSR operators as shown in Theorem 4.3.1 and Lemma 4.3.1. The detailed proof is in Appendix B.3. □
A desirable property of the transformation rules is completeness with respect to trace-set refinement. We found out that the transformation rules are complete, modulo strong bisimulation \(\sim\), in the case of finite processes.

**Theorem 4.3.3** *(Completeness of transformation rules.)* For all finite processes \(P\) and \(Q\), if \(P \preceq^\rho Q\) for some one-to-one, complement preserving function \(\rho\), then there exist processes \(Q' \sim Q\) and \(P' \sim P\), a function \(\rho_1\), and a relation \(\rho_2\), such that \(\text{ref}^\rho_{(\rho_1,\rho_2)}(Q') \sim P'\).

**Proof:** Appendix B.4 contains the proof. \(\square\)

We conjecture that by augmenting the transformation rules with the rule

\[
\text{eref}^\rho_{(\rho_1,\rho_2)}(\text{rec } X.Q, I) \sim \text{eref}^\rho_{(\rho_1,\rho_2)}(Q[\text{rec } X.Q / X], I)
\]

the above theorem also holds for finite-state, non-Zeno processes.

The reason strong bisimulation is needed is that since trace-set refinement does not impose any particular structure on \(P\) (while the transformation rules preserve the structure of \(Q\)) we may need to transform either \(Q\) or \(P\) to a strongly bisimilar process with the right structure. We elected to use strong bisimulation, as opposed to trace equivalence, for two reasons. One reason is that a complete set of axioms for strong bisimulation is already available (see [BGCL93]) and thus any ACSR process can be transformed using these axioms. The second is that strong bisimulation implies trace equivalence and thus we indirectly get completeness of the transformation rules modulo trace equivalence.

For the second notion of trace refinement relation, \(\preceq\), we can also define a set of transformation rules that is sound and complete. The set basically differs from the one defined earlier in the transformation rules (4) and (5): in this case, there is no need to preserve shared resources. For the sake of completeness, we next list the set of transformation rules in the case of the trace refinement \(\preceq\) of Definition 4.2.3. The proof of the soundness and completeness of these transformation rules are left to the reader since they can be proven in a way similar to Theorems 4.3.2 and 4.3.3.

**Definition 4.3.7** Let \(Q\) be a process, let the function \(\rho_1 : \text{events}(Q) \rightarrow \text{Evts}\) be one-to-one, complement preserving, and let the relation \(\rho_2 \subseteq \text{actions}(Q) \times \text{Proc}\) satisfy for each \((A', R) \in \rho_2\)

\[
(\text{events}(R) \cap \text{range}(\rho_1) = \emptyset) \land \ (\text{res}(A) \subseteq \text{res}(R)).
\] (4.4)
Then, \( Q \) can be transformed using the following transformation rules:

\[
\begin{align*}
(1) \quad & \text{ref}^i_{(p_1,p_2)}(Q) \quad \sim \quad \text{NIL} \\
(2) \quad & \text{ref}^i_{(p_1,p_2)}(Q) \quad \sim \quad \sum_{l \in L} l.\text{ref}^i_{(p_1,p_2)}(Q) \quad \text{for finite } L, L \cap \text{range}(p_1) = \emptyset \\
(3) \quad & \text{ref}^i_{(p_1,p_2)}(e.Q) \quad \sim \quad p_1(e).\text{ref}^i_{(p_1,p_2)}(Q) \\
(4) \quad & \text{ref}^i_{(p_1,p_2)}(A^I : Q) \quad \sim \quad R \triangle_\tau (\text{NIL},\text{ref}^i_{(p_1,p_2)}(Q),\text{NIL}) \quad \text{for } (A^I, R) \in p_2 \\
(5) \quad & \text{ref}^i_{(p_1,p_2)}(Q_1 || Q_2) \quad \sim \quad \text{ref}^i_{(p_1,p_2)}(Q_1) || \text{ref}^i_{(p_1,p_2)}(Q_2) \\
(6) \quad & \text{ref}^i_{(p_1,p_2)}(Q_1 + Q_2) \quad \sim \quad \text{ref}^i_{(p_1,p_2)}(Q_1) + \text{ref}^i_{(p_1,p_2)}(Q_2) \\
(7) \quad & \text{ref}^i_{(p_1,p_2)}(Q \Delta^*_\tau (R, S, T)) \sim \quad \text{ref}^i_{(p_1,p_2)}(Q) \Delta^*_{\tau} (\text{ref}^i_{(p_1,p_2)}(R),\text{ref}^i_{(p_1,p_2)}(S),\text{ref}^i_{(p_1,p_2)}(T)) \\
(8) \quad & \text{ref}^i_{(p_1,p_2)}(Q) \quad \sim \quad \text{ref}^i_{(p_1,p_2)}(Q) \setminus F \\
(9) \quad & \text{ref}^i_{(p_1,p_2)}(Q \setminus F) \quad \sim \quad \text{ref}^i_{(p_1,p_2)}(Q) \setminus p_1(F) \\
(10) \quad & \text{ref}^i_{(p_1,p_2)}([Q]_U) \quad \sim \quad [\text{ref}^i_{(p_1,p_2)}(Q)]_U \quad \text{for } U \subseteq U' \\
(11) \quad & \text{ref}^i_{(p_1,p_2)}(\text{rec } X.Q) \quad \sim \quad \text{rec } X.\text{ref}^i_{(p_1,p_2)}(Q) \\
(12) \quad & \text{ref}^i_{(p_1,p_2)}(X) \quad \sim \quad X
\end{align*}
\]

\( \square \)

### 4.3.3 Process Refinement in the Egli-Milner Ordering

In the previous section, we examined trace-set refinement in the Hoare ordering, \( P \preceq_P^i Q \). This gave us the weaker notion of an implementation relation where an implementation may eliminate some behaviors of the specification. We also have examined a stronger notion of an implementation relation that is obtained from the Egli-Milner ordering extension of trace refinement, \( P \preceq_P^i Q \).

Unlike the Hoare ordering extension of trace refinement, the Egli-Milner ordering forces the refining process to contain all of the behaviors that are related to the specification behaviors; that is, it cannot eliminate behaviors. More formally, Egli-Milner trace-set refinement is defined as:

\[
P \preceq_P^i Q \quad \text{if and only if} \quad P \preceq_P^i Q \quad \text{and} \quad P \preceq_P^i Q.
\]

Thus, in order to prove the compositionality of \( \preceq_P^i \) with respect to the ACSR operators, we only need to prove it for the Smyth ordering \( \preceq_P^i \). This result is proven in Appendix B.5. The significant difference from the compositionality of the Hoare ordering is the Parallel operator where the resource restriction is switched between the first and second notions.
of trace refinement. More specifically, for the first notion of trace refinement, \( \leq \), compositionality with the Parallel operator does not require any restriction on the resources used by the parallel processes. However, for the second notion of trace refinement, \( \leq \), compositionality with the Parallel operator must be restricted such that the added resources are not shared between the two processes running in parallel.

The set of transformation rules for the Egli-Milner ordering, \( \text{ref}_{(\rho_1, \rho_2)} \), also vary slightly from those for the Hoare ordering. The significant three changes are: a stronger condition is required on the relation \( \rho_2 \); transformation rule (1) must be stricter; transformation rule (8) is not included; and in transformation rule (10) the refining process can not close more resources than the abstract process. More specifically, in the Egli-Milner ordering the relation \( \rho_2 \subseteq \text{actions}(Q) \times \text{Proc} \) is defined such that for each \((A^t, R) \in \rho_2 \)

\[
(\text{events}(R) \cap \text{range}(\rho_1) = \emptyset) \land (\text{res}(R) \subseteq \text{res}(A)) \land (\text{dur}(R) \geq t),
\]

where \( \text{dur}(R) \) is the maximal duration of the process \( R \) in all contexts. Formally,

\[
\text{dur}(R) \overset{\text{def}}{=} \min_{s \in \text{maxtraces}(R)} \text{dur}(s).
\]

Intuitively, a trace of process \( R \) is maximal if \( R \) produces the trace and then stops, i.e., becomes the NIL process. The set of maximal traces, \( \text{maxtraces}(R) \), is defined in Appendix D.

The conditions on \( \rho_2 \) are stronger than those in the Hoare ordering since the refining process is required to have at least the same duration as the refined action. This ensures that if the specification has a trace that extends beyond the execution of the refined action, the implementation must be able to match such a trace. In the Hoare ordering, however, the implementation had the option of stopping.

Transformation rule (1) is also stricter since the only process that can be transformed to the NIL process is the NIL process itself. Transformation rule (8) is not allowed since by forcing synchronization, this rule may eliminate behaviors from the specification. Transformation rule (10) does not allow the refining process to close more resources than in the specification, since resource closure can also eliminate behaviors from the specification.

The soundness of the transformations in the Egli-Milner can be proven in a similar way to the Hoare ordering. In this case, we conjecture that the completeness of the transformation rules is modulo trace equivalence.
4.4 Refinement in the Prioritized Semantics

In this section, we consider refinement in the prioritized semantics of ACSR. The prioritized trace semantics $\xrightarrow{\alpha}_\pi$ is defined as an extension of the prioritized transition system $\xrightarrow{\alpha}_\pi$; see Appendix D for the definition of the prioritized trace semantics.

**Theorem 4.4.1** In the prioritized semantics, trace-set refinement in the Hoare ordering ($\leq^1$) is compositional with the Prefix, Restriction, and Scope in its second and third arguments (end of scope and timeout), under the same conditions as compositionality in the unprioritized semantics.

**Proof:** The proof is similar to compositionality in the unprioritized semantics. \(\square\)

For the remaining ACSR operators, we next give counter examples to prove that trace-set refinement is not compositional in the Hoare ordering.

**Choice.** The Choice operator is not compositional in the prioritized semantics for two reasons. One is that adding a local event may make two transitions in a process incomparable by the preemption relation. The second is that a resource that decides the preemption of a transition may be released which results in two transitions that are incomparable.

To illustrate how local events and resource release affect compositionality with the Choice operator, consider the following processes:

\[
Q \overset{\text{def}}{=} (a, 2).NIL + \{(r_1, 1), (r_2, 2)\} : \text{NIL}
\]
\[
P \overset{\text{def}}{=} (b, 1).(a, 2).NIL + \{(r_1, 1)\} : \text{NIL}
\]
\[
E \overset{\text{def}}{=} (a, 1).NIL + \{(r_1, 1), (r_2, 1)\} : \text{NIL}
\]

For the identity event relabeling function, \(Id_Q\), we have \(P \leq^1_{Id_Q} Q\). Once we combine \(Q\) and \(P\) with \(E\) in a choice, we have the following prioritized traces:

\[
Q + E \xrightarrow{(a,2)}^\pi E \xrightarrow{\{(r_1,1),(r_2,2)\}}^\pi
\]
\[
P + E \xrightarrow{(b,1)(a,2)}^\pi E \xrightarrow{\{(r_1,1)\}}^\pi \xrightarrow{\{(r_1,1),(r_2,1)\}}^\pi
\]

In \(Q + E\), the trace \((a, 1)\) from \(E\) was preempted by the trace \((a, 2)\) of \(Q\). This trace, however, remained in \(P + E\) because the added event \((b, 1)\) and event \((a, 1)\) are incomparable.
by the preemption relation. In addition, the trace on the timed action in $E$ was preempted in $Q + E$. This trace remained in $P + E$ because the resource $r_2$ which decided the preemption in $Q + E$ was released in $P$. These additional traces in $P + E$ do not refine any trace in $Q + E$. Thus, $P + E \not\leq_{1dQ} Q + E$. In other words, trace-set refinement in the Hoare ordering and prioritized semantics is not compositional with the Choice operator.

**Parallel.** As a counter example to the compositionality of the trace-set refinement with the `Parallel` operator, consider the following processes:

$$
\begin{align*}
Q & \overset{\text{def}}{=} \emptyset : \text{NIL} + (a,1),\text{NIL} \\
P & \overset{\text{def}}{=} \emptyset : \text{NIL} \\
E & \overset{\text{def}}{=} \emptyset : \text{NIL} + (\pi,1),\text{NIL}
\end{align*}
$$

It is easy to check that $P \leq_{1dQ} Q$. However, in the prioritized semantics $P \parallel E \not\leq_{1dQ} Q \parallel E$: we have $P \parallel E \xrightarrow{\emptyset} E^*; \; E$; however, the transition on $\emptyset$ is preempted in $Q \parallel E$ by the synchronization on $a$ which produces $(\tau,2)$. Thus, $P \parallel E \not\leq_{1dQ} Q \parallel E$.

**Close.** Releasing resource makes trace-set refinement in the Hoare ordering not compositional with the `Close` operator in the prioritized semantics. As an illustrating example, consider the following refinement:

$$
\begin{align*}
\emptyset : \text{NIL} & \leq_{1} \{(r,1)\} : \text{NIL} \\
[\emptyset : \text{NIL}]_{(r)} & \not\leq_{1} [\{(r,1)\} : \text{NIL}]_{(r)}
\end{align*}
$$

**First and fourth arguments of `Scope.** It is not a surprise that the `Interrupt` does not behave well with trace-set refinement since it can be interpreted as a Choice operator. As an example, consider the following processes where the refinement releases the resource $r_2$, which caused the incomparability:

$$
\begin{align*}
Q & \overset{\text{def}}{=} \{(r_1,2), (r_2,2)\} : \text{NIL} \\
P & \overset{\text{def}}{=} \{(r_1,2)\} : \text{NIL} \\
E & \overset{\text{def}}{=} \{(r_1,1), (r_3,1)\} : \text{NIL}
\end{align*}
$$

We have $P \leq_{1} Q$, however,

$$
\begin{align*}
P \triangle^*_t (\text{NIL}, \text{NIL}, E) & \not\leq_{1} Q \triangle^*_t (\text{NIL}, \text{NIL}, E) \\
E \triangle^*_t (\text{NIL}, \text{NIL}, P) & \not\leq_{1} E \triangle^*_t (\text{NIL}, \text{NIL}, Q)
\end{align*}
$$
Recursion. The main reason for recursion not to behave well in the prioritized semantics is the Close operator. As an example, consider the following processes:

\[ Q \overset{\text{def}}{=} \emptyset : \text{NIL} + \{ (r_1, 1) \} + \left[ X \right]_{(r_1, r_2)} \]

\[ P \overset{\text{def}}{=} \{ (r_1, 1) \} : \text{NIL} + \emptyset : \left[ X \right]_{(r_1, r_2)} \]

We have \( P \leq^1 Q \), however, when we unfold the recursion in \( P \) once, \( rec X.P \) will have the following type of traces only

\[ rec X.P \xrightarrow{\emptyset}^* \{ (r_1, r_2, 0) \} \{ (r_1, 1), (r_2, 0) \} \xrightarrow{\tau}^* \ldots \]

which do not refine any trace of \( rec X.Q \) where after unfolding the recursion once we get the following type of traces

\[ rec X.Q \xrightarrow{\{ (r_1, r_2, 0) \}}^* \{ (r_1, 1), (r_2, 0) \} \{ (r_1, 2), (r_2, 0) \} \xrightarrow{\tau}^* \ldots \]

Thus, \( rec X.P \not\leq^1 rec X.Q \).

**Theorem 4.4.2** In the prioritized semantics, trace-set refinement in the Egli-Milner ordering (\( \leq^1 \)) is compositional with the Prefix, Restriction, and Scope in its second and third arguments (end of scope and timeout), under the same conditions as compositionality in the unprioritized semantics.

**Proof:** The proof is straightforward and similar to the unprioritized semantics. □

The examples given above in the Hoare ordering for the Choice, Close, Scope (first and fourth arguments), and recursion operators can be used to prove that trace-set refinement is not compositional with these operators in the prioritized semantics. For the Parallel operator, consider the following processes:

\[ Q \overset{\text{def}}{=} \emptyset : \text{NIL} + (a, 1).\text{NIL} \]

\[ P \overset{\text{def}}{=} (b, 1), \emptyset : \text{NIL} + (a, 1).\text{NIL} \]

\[ E \overset{\text{def}}{=} \emptyset : \text{NIL} + (\tau, 1).\text{NIL} \]

It is easy to check that \( P \leq^1_{dQ} Q \). The process \( Q \| E \) does not have any time consuming trace since the synchronization trace \( (\tau, 2) \) preempts the trace \( \emptyset \). However, we have \( P \| E \xrightarrow{(b,1)\emptyset}^* \) which does not refine any trace of \( Q \| E \). Thus, \( P \| E \not\leq^1_{dQ} Q \| E \)
4.5 Example: Task System

In this section, we use an example of a task system to illustrate the use, advantages, and limitations of the presented notion of refinement. In order to make the example more readable, we use indexed processes, and process constants instead of the recursion operator \( \text{rec} \).

At the most abstract level of description, one would think of a task system as a system where a set of resources, e.g., processors, I/O devices and ready queues, are continuously used to produce a certain functionality. Let us consider a system with one processor, \( \text{cpu} \), and one queue of length 3, represented by \( q_1, q_2, \) and \( q_3 \). Furthermore, assume that the system executes in periods of length \( p \) time units. The essential property at this level of abstraction is that the system never deadlocks, i.e., it does not stop. This property is represented by the set of infinite traces of the process \( \text{Spec} \) defined below.

\[
\text{Spec} \overset{\text{def}}{=} \{ \text{cpu}, q_1, q_2, q_3 \}^p : \text{Spec}.
\]

The above system description abstracts out several details about the resource usage; in particular, it does not distinguish between a task that uses the processor and a task that is waiting in a queue, nor does it describe the order in which the system tasks are executed.

The process \( \text{Imp}_1 \), below, provides a more accurate resource utilization and forces an order of execution among the system tasks:

\[
\begin{align*}
\text{Imp}_1 & \overset{\text{def}}{=} \text{OrdExec} \triangle_p (\text{NIL}, \text{Imp}_1, \text{NIL}) \\
\text{OrdExec} & \overset{\text{def}}{=} \{ \text{cpu}, q_2, q_3 \}^{c_1} : \{ q_1, \text{cpu}, q_3 \}^{c_2} : \{ q_1, q_2, \text{cpu} \}^{c_3} : \text{Wait} \\
\text{Wait} & \overset{\text{def}}{=} \{ q_1, q_2, q_3 \} : \text{Wait}
\end{align*}
\]

The process \( \text{Imp}_1 \) shows that at most one task can use the \( \text{cpu} \) resource at any time while the other tasks are waiting in the queues. In addition, \( \text{Imp}_1 \) describes how long each task uses the \( \text{cpu} \) resource. This process can be syntactically derived from the abstract specification process \( \text{Spec} \) by using the transformation rule for action-prefix processes, Rule (4). Using the soundness of the transformation rules, we can conclude that \( \text{Imp}_1 \) refines \( \text{Spec} \). We can not however infer from the fact that \( \text{Spec} \) is deadlock free that the design \( \text{Imp}_1 \) is also deadlock free. Instead, we need to verify explicitly this property on the design \( \text{Imp}_1 \). It is easy to inspect the process \( \text{Imp}_1 \) and see that it is a sequential, recursive, and non-terminating process.
The above derived designs provide a "functional" description of the system; in particular, they do not reflect any software structure. The next design instance of the system focuses on the system structure in terms of its task set and provides a detailed resource usage. In this design, the system consists of three tasks $T_1$, $T_2$, and $T_3$, each of which either uses the processor $cpu$ for $c_i > 0$ time units or waits in a queue $q_i$ for the processor to become available; in addition, at most one task can use the processor at any time.

Let $Imp_2 = \text{TaskSet} \triangleq (\text{NIL}, Imp_2, \text{NIL})$

$\text{TaskSet} = T_1 \parallel T_2 \parallel T_3$

$T_i = \text{Exec}_i[0] + \text{Wait}_i[0]$ for $i = 1, 2, 3$

$\text{Exec}_i[t] = \{cpu\} : \text{Exec}_i[t+1] + \text{Wait}_i[t]$ for $0 \leq t < c_i$

$\text{Wait}_i[t] = \{q_i\} : \text{Wait}_i[t] + \text{Exec}_i[t]$ for $0 \leq t < c_i$

$\text{Exec}_i[c_i] = \{q_i\} : \text{Exec}_i[c_i]$

It is easy to see that one can rewrite $Spec$ to $Imp_2$ using the rewrite rule for an action-prefix process, Rule (4). By the operational semantics of the Parallel operator, the mutual exclusive use of the processor is guaranteed in the design instance $Imp_2$. To verify the deadlock freeness of $Imp_2$, we again can not rely on inheriting this property from the specification. Instead, we can use weak bisimulation and resource hiding and verify that

$$Imp_2 \angle \{cpu, q_1, q_2, q_3\} \sim Idle$$

where the process $Idle = \emptyset : Idle$.

Unlike the design solution $Imp_1$, the above design abstracts out the order of execution among the three tasks and lacks efficiency since, by operational semantics, the three tasks may never execute while the processor sits idle. A more realistic design, $Imp_3$, associates a scheduler with the set of tasks to force an order of execution and a better resource utilization. We next illustrate how $Imp_2$ can be transformed to a design that implements a round-robin scheduling algorithm with time slice $c$. We basically proceed according to the following refinement steps: 1) refine each task $T_i$ to add events to synchronize with the dispatcher; 2) construct a dispatcher process that implements the round-robin scheduling algorithm; 3) replace the $TaskSet$ process in $Imp_2$ with the refined set of tasks and the dispatcher process. Using compositionality of the transformation rules, we are guaranteed that the resulting process refines the process $Imp_2$ and, thus, $Spec$. 
Task refinement. Each task $T_i$ is refined to a task $T'_i$ (for $i = 1, 2, 3$) that contains communication events to synchronize with the dispatcher:

$$T'_i \overset{\text{def}}{=} \text{Exec}_i[0] + \text{Wait}_i[0] \quad \text{for } i = 1, 2, 3$$
$$\text{Exec}_i[t] \overset{\text{def}}{=} \text{start}_i; \{\text{cpu}\} : \text{Exec}_i[t + 1] + \text{stop}_i; \text{Wait}_i[t] \quad \text{for } 0 \leq t < e_i$$
$$\text{Wait}_i[t] \overset{\text{def}}{=} \{\text{q}_i\} : \text{Wait}_i[t] + \text{Exec}_i[t] \quad \text{for } 0 \leq t < e_i$$
$$\text{Exec}_i[e_i] \overset{\text{def}}{=} \text{end}_i; \text{Done}_i$$
$$\text{Done}_i \overset{\text{def}}{=} \{\text{q}_i\} : \text{Done}_i$$

It is easy to see that each process $T_i$ (for $i = 1, 2, 3$) can be transformed to a process $T'_i$ using the transformation rules for the Eglı-Milner ordering (as well as the Hoare ordering). Note also that each $T'_i$ is obtained by basically introducing new events, and in particular the $\text{cpu}$ resource was not released in $T'_i$ when it was used in $T_i$. We therefore have

$$\text{cref}_{\rho_1, \rho_2}(T_i, \{\text{cpu}\}) \leadsto T'_i \quad \text{for } i = 1, 2, 3$$

where $\rho_1$ is the null function and $\rho_2$ maps each action $A$ to the process $A : \text{NIL}$.

Dispatcher design. The process $\text{Dispatcher}$, below, implements a round robin scheduling algorithm with time slice $c$. It uses processes $D[k, j, i]$ for $i, j, k = 0, 1, 2, 3$ to identify the dispatcher when the FIFO ready queue contains task $T_i$ at its head and task $T_k$ at its tail; a zero index indicates no task is in the corresponding queue position.

$$\text{Dispatcher} \overset{\text{def}}{=} D[3, 2, 1]$$
$$D[k, j, i] \overset{\text{def}}{=} \text{start}_i; \text{Idle} \triangle (\text{NIL}, \text{Stop}, D[i, k, j], \text{end}_i, D[0, k, j])$$
$$\quad \text{for } i, j, k = 1, 2, 3 \quad \text{and } i \neq j \neq k$$
$$D[0, j, i] \overset{\text{def}}{=} \text{start}_i; \text{Idle} \triangle (\text{NIL}, \text{Stop}, D[0, i, j], \text{end}_i, D[0, 0, j])$$
$$\quad \text{for } i, j = 1, 2, 3 \quad \text{and } i \neq j$$
$$D[0, 0, i] \overset{\text{def}}{=} \text{start}_i; \text{Idle} \triangle (\text{NIL}, \text{NIL}, \text{end}_i, \text{Idle}) \quad \text{for } i = 1, 2, 3$$
$$D[0, 0, 0] \overset{\text{def}}{=} \text{Idle}$$
$$\text{Idle} \overset{\text{def}}{=} \emptyset : \text{Idle}$$

The process $\text{Dispatcher}$ basically either idles or sends and receives events. It can therefore be seen as the refinement of the $\text{Idle}$ process; that is, for $\rho_1 = \emptyset$ and $\rho_2$ a relation that maps each action $A$ to the process $A : \text{NIL}$, we have

$$\text{Dispatcher} \preceq^\circ_{\rho_1} \text{Idle}$$
In other words, we can use the transformation rules to rewrite the process \textit{Idle} to the process \textit{Dispatcher} as the next transformation steps illustrate:

\[
\begin{align*}
\text{Idle} & \triangleq \emptyset : \text{Idle} \\
\sim & \emptyset^c : \text{Idle} \\
\sim & \text{Idle} \triangleq (\text{NIL, Idle, Idle}) \\
\sim & \begin{array}{c}
\text{start}_{i}, \text{ref}^o_{(\rho_1, \rho_2)}(\text{Idle}) \triangleq (\text{NIL, \text{stop}_{i}, \text{ref}^o_{(\rho_1, \rho_2)}(\text{Idle}), \text{end}_{i}, \text{ref}^o_{(\rho_1, \rho_2)}(\text{Idle})) \\
\text{for } i = 1, \text{ by Rule}(2)
\end{array}
\end{align*}
\]

Strong bisimulation axioms along with the transformation Rules (2) and (4) can be further applied, in a similar fashion, on the two occurrences of \text{ref}^o_{(\rho_1, \rho_2)}(\text{Idle}) to get \text{D}[i, k, j] and \text{D}[0, k, j].

**Putting it together.** For \( \rho_1 = \emptyset \) and \( \rho_2 \) which maps each action \( A^i \) to \( A^i : \text{NIL} \) we have the following:

\[
\begin{align*}
T_1 \parallel T_2 \parallel T_3 & \sim T_1 \parallel T_2 \parallel T_3 \parallel \text{Idle} \\
\text{cref}^o_{(\rho_1, \rho_2)}(T_i, \{\text{cpu}\}) & \sim T_i^f \text{ for } i = 1, 2, 3 \\
\text{cref}^o_{(\rho_1, \rho_2)}(\text{Idle}) & \sim \text{Dispatcher} \\
\text{ref}^o_{(\rho_1, \rho_2)}(T_1 \parallel T_2 \parallel T_3) & \sim T_i^f \parallel T_j^f \parallel T_k^f \parallel \text{Dispatcher}
\end{align*}
\]

Using the Hoare-ordering transformation rule (8), we can get

\[
\text{ref}^o_{(\rho_1, \rho_2)}(T_i^f \parallel T_j^f \parallel T_k^f \parallel \text{Dispatcher}) \sim (T_i^f \parallel T_j^f \parallel T_k^f \parallel \text{Dispatcher}) \setminus F
\]

where \( \rho_1 \) is the identity function, \( \rho_2 \) maps each action \( A^i \) to \( A^i : \text{NIL} \) and

\[
F = \{\text{start}_1, \text{start}_2, \text{start}_3, \text{stop}_1, \text{stop}_2, \text{stop}_3, \text{end}_1, \text{end}_2, \text{end}_3\}
\]

As a final result, we again make use of the compositionality of the transformation rules, and the transformation rule (7) for refining a Scope and rules (11) and (12) for refining a recursion, we have

\[
\begin{align*}
\text{ref}^o_{(\rho_1, \rho_2)}(\text{Spec}) & \sim \text{Imp}_2 \\
& \triangleq \text{rec } X. (T_1 \parallel T_2 \parallel T_3) \triangleq_p (\text{NIL, X, NIL}) \\
& \sim \text{rec } X. (T_i^f \parallel T_j^f \parallel T_k^f \parallel \text{Dispatcher}) \setminus F \triangleq_p (\text{NIL, X, NIL})
\end{align*}
\]

In other words, we presented a detailed description of the task system as a set of tasks and a dispatcher that are instantiated periodically, and we have described a more realistic resource usage.
The above refinement steps have been carried out in the unprioritized semantics where we can refine a parallel process by refining its subprocesses. Similar refinement steps are, however, not allowed in the prioritized semantics. While it might be possible to follow the above design scheme to implement a more complex priority-based scheduler within the unprioritized semantics, e.g., rate monotonic, a simpler and more natural design would use the priorities and preemption relation of the ACSR computation model.

4.6 Discussion

We have presented a refinement theory for timed process algebra ACSR. Refinement in ACSR combines two types of refinement operators, relabeling and action refinement, through transformation rules that syntactically rewrite one process to another. Relabeling is adopted for ACSR instantaneous events while action refinement is adopted for ACSR time-consuming actions. In addition, the transformation rules allow the addition of events and reduction or increase of resource usage in the refining specification. The soundness of the transformation rules is defined in terms of two possible preorder relations over sets of traces, a possible semantics of ACSR processes. The trace-set relations ensure that related traces preserve the timed occurrences of events of the abstract specification. This in turn allows the inheritance of properties, e.g., safety, between different levels of abstraction.

In the unpriortitized semantics, the trace-set relations are compositional with the ACSR operators under a reasonable set of conditions. This allows the modular application of the transformation rules. In the presence of priorities, there does not seem any set of conditions that make either of the trace-set refinements compositional with all of the ACSR operators. The major difficulties stem from the Choice operator. In the prioritized semantics, our notion of trace refinement does not preserve the comparability of actions with respect to the preemption relation. We conjecture, however, that a judicious application of the transformation rules can be used for a compositional refinement. For example, one has to ensure that two comparable actions in the specification are refined in such a way that they remain comparable in the implementation.

4.6.1 Advantages of Our Approach

Event refinement. In addition to event relabeling, our notion of event refinement allows a designer to add new events to an abstract specification. This in turn allows them to model
implementation-related communications and to add events that can be used as watch-dog steps to test the presence or lack of certain behaviors.

**Action refinement.** Our notion of action refinement differs from other action refinements in two ways: First, ACSR action refinement allows different occurrences of one action to be syntactically replaced by different processes, which makes it a relation over processes instead of a function. Second, ACSR action refinement has a notion of resource refinement which allows a refining process to use fewer (or more) resources than the abstract process. The motivation behind the differences are two-fold: the notion of an ACSR action and the way we envision ACSR action refinement to be used.

An ACSR action can be considered as an abstract representation of time and resource consuming system activities such as computations. Thus, two activities may use the same resources for the same amount of time yet they may represent different activities. Such activities would be represented in ACSR by two different occurrences of the same ACSR action. The refinement of the two system activities may lead to different processes. Thus, it is natural to allow the two occurrences of the action to be refined differently. An immediate question is why not make action refinement a function which takes into account an action’s occurrence number? While this is theoretically feasible, it may however complicate the application of refinement (as syntactic substitution) to account for the occurrence numbers of actions especially in the presence of recursion.

The second distinctive feature of ACSR action refinement is its notion of resource refinement: the refined process may use fewer (or more) resources than the abstract action. This notion of resource refinement is motivated by the fact that at the abstract level, designers may not know the specific resource requirements for an abstract activity. Thus, it is essential to allow designers to estimate these requirements. Later, as the system activities are refined, designers should be allowed to tighten the estimates and free unnecessary resources or add required resources.

**Syntactic approach.** The syntactic approach to refinement is a step towards providing designers with a set of tools to derive an implementation from a specification (or vice versa) syntactically and in a modular way. In fact, the set of transformation rules can be implemented within a design environment. This in turn bridges the gap between the specification and implementation stages of the development life cycle.
Property preservation. An important issue in a hierarchical design methodology is the preservation of properties between the different levels of abstraction. In other words, the refinement steps must maintain “correctness” with respect to the original specification. This eliminates the need to prove a system property twice: once for the specification and a second time for the implementation, or vice versa.

A system property can be expressed either in terms of an ACSR process, or more concisely, as a predicate on a typical behavior of the system [Hoa85, DS89, Sch90, MP91]. When properties are expressed as processes, we can use trace-set refinement in the Hoare ordering to define a notion of satisfiability: a system $P$ satisfies a property $S$ if $\text{traces}(P) \leq^* \text{traces}(S)$. This notion is equivalent to the common notion of satisfiability expressed in terms of trace set inclusion. The transformation rules give us the advantage of proving satisfiability syntactically—in the case of finite processes.

It is often more natural to express properties as predicates on a typical or desirable behavior of the system. In this case, a system property is commonly expressed in terms of the timed occurrences of certain events in the system traces. Given the properties we have proven about our notion of trace refinement (in particular Properties 3 through 5), it is straightforward to prove that trace refinement preserves an important set of properties: safety properties, e.g., “if event $e_2$ ever occurs, it is preceded by event $e_1$” [MP91].

In both forms of properties, our notions of refinement ensure that in the Hoare ordering the implementation inherits the specification properties, and in the Egli-Milner ordering the implementation and specification inherit each other’s properties.

4.6.2 Limitations of Our Approach

The presented refinement approach has limitations that stem from the syntax-based approach as well as some conceptual decisions. Some of these limitations are shared with work based on a syntactic approach to refinement, and others are particular to the ACSR model.

Limitations due to the syntactic approach. The syntax-based approach to refinement has the following limitations:

- Cannot refine an event to a process. A legitimate question is why not treat event refinement as action refinement, i.e., an event is uniquely refined to a process that executes events only? From a theoretical point of view, using action refinement for
ACSR events requires major changes to ACSR, semantic or both semantic and syntactic changes. One alternative is to change the semantics of the Scope operator, which can be used for prefixing processes, so that it executes the events of the main process before starting the timeout process. A second alternative is to change the ACSR event-prefix operator to denote process prefixing, along the lines of CSP-like process algebras [Hoa85]. This syntactic modification was required by all action refinements defined for CCS-like process algebras, e.g., [Ace92, Jat93]. The second alternative also requires the addition of a notion of a “successfully” terminated process; this would change the semantics of ACSR which does not distinguish between successful and unsuccessful process termination.

We elected not to pursue this approach to event refinement for several reasons. One is that the required changes would result in a different version of ACSR that is not convincingly justifiable. Another reason is that, from a practical point of view, the advantages of allowing an instantaneous event to be refined to several, instantaneous events are unclear. An event can be seen as a time marker and, thus, one representative time marker would suffice. Furthermore, we conjecture that our notion of event refinement, where new events can be added to the refined process, can be used to mimic action refinement for events.

- Can not add interactions between process components that are refined in different steps. This limitation is captured by the notion of compatibility (Definition 4.3.3) and is enforced by Condition (4.3) in the Hoare transformation scheme and Condition (4.5) in the Egli-Milner transformation scheme. It basically states that the local (or new) events in a process must be distinct from the events of its context. As an example, if an action A is refined to a process R, then R can not contain events that can synchronize with another process in the context where A was.

This limitation is shared with refinements based on relabeling as well as action refinement. For relabeling, the restriction that the relabeling function be one-to-one disallows any unintended interactions. For action refinement, α-conversion is used to guarantee that local events are truly “new” [Ace92, AH89]; in Petri nets with action refinement this restriction is guaranteed by disallowing new transitions from places in the refining net to its context Petri net [Jat93, vGG89].
Limitations due to conceptual decisions.

- *Refining process preserves the timing requirements of the abstract process.* It might be useful to support a notion of refinement where the refining process is faster or slower than the abstract process. Such a notion of refinement allows a designer, for instance, to develop an optimized faster, or fault-tolerant but slower implementation of a system. In such a refinement, the specification events are represented by events that occur earlier or later, respectively, in the implementation. Such a notion of refinement however is not compositional with the Parallel operator due to the potential introduction of deadlocks that result from missed synchronizations [BG94, BGBAL96].

The difficulty of making such a notion of refinement compositional can be related to guaranteeing synchronization in a distributed system with multiple, unsynchronized clocks. Without low level, detailed information about the behaviors of the communicating tasks and the different system clocks, a solution may never be found. In terms of ACSR processes, the detailed information is captured semantically and thus any syntactic approach would either be impossible or too restrictive.

- *The Hoare ordering and Egli-Milner ordering can be impractical.* Consider the case where a given specification, $Spec$, is formulated and analyzed. When a designer finds out that $Spec$ has a deadlock due to a loose estimate of the resource usage, the natural next step in the design is to tighten the estimate in order to eliminate the deadlock. Such a step could require releasing certain shared resources. Neither the Hoare ordering nor the Egli-Milner ordering allows such a modification, since by removing the deadlock more behaviors can be produced in the modified specification. Ideally, we would like to modify the specification to resolve the deadlock and inherit some properties from the original specification. In other words, the modified specification is not an arbitrary specification, as the Smyth ordering allows. The question is, therefore, whether such a notion of specification modification can be represented as an ordering between the Hoare ordering and Egli-Milner ordering. Such a notion of specification modification is beneficial since it reduces the design costs by saving the analysis efforts invested into the original specification.

It is obvious that the notions of refinement are numerous for ACSR. However, defining a notion of refinement that is compositional with respect to all of the ACSR operators in
the prioritized semantics is not trivial if possible at all. The presented refinement tries to accommodate a "reasonable" notion of hierarchical specifications that can be applied within a design environment. The syntactic transformation rules provide a set of tools for top-down design. They adopt practical solutions to the difficulties due to synchronization events and resource sharing to simplify their implementation within a design environment. The next chapter describes a prototype environment where the transformation rules for the Hoare ordering have been implemented within a tool set for GCSR.
Chapter 5

The GCSR Tool Set

In order to experiment with the GCSR language and refinement theory, we have augmented the VERSA system [CLX95b] with a tool set for the use of GCSR for real-time systems modeling, refinement, and analysis. Figure 5.1 shows the overall structure of the GCSR/VERSA system. The user's view of the tool is provided by the GCSR and XVERSA user interfaces. The analysis of GCSR (and ACSR) specifications is carried out by the VERSA system which is accessed through these two interfaces.

Chapter organization. We next briefly review the VERSA system and its XVERSA interface. In Section 5.2, we present a tutorial of the GCSR tool set. In Section 5.3, we describe how the ACSR refinement presented in Chapter 4 is supported in the GCSR tool set. We conclude the chapter with a summary of the implemented features and future work.
5.1 The VERSA System

The Verification, Execution, and Rewrite System for ACSR (VERSA) is a tool set for specifying and analyzing system models in ACSR. The XVERSA interface is an X-Windows based, textual interface to the VERSA system.

The syntax in VERSA (and thus XVERSA) is as close to the pure ACSR syntax as the keyboard allows. The VERSA syntax is also enhanced with the facility to define macros (e.g., to define manifest constants) and parametrized process terms.

The three main features of the VERSA system are: syntax checking, compilation, and functional analysis of processes. Syntax checking is supported through a Lex/Yacc based parser. Compilation uses the semantic rules of ACSR to translate ACSR processes into their underlying labeled transition systems (LTS) representation. The LTS for one or more processes is produced by an algorithm that expands the process to produce a labeled transition system representing all possible executions. The LTS construction algorithm also prunes edges made unreachable by the semantics of the prioritized transition system, in most cases reducing the size of the resulting LTS. Functional analysis of processes consists of four major areas: term rewriting, state space exploration, equivalence testing, and interactive execution.

The rewrite system facilitates the rewriting of ACSR process expressions according to sound algebraic laws that preserve prioritized strong equivalence, a bisimulation relation that respects priority. At the direction of the user, the rewrite system applies pre-defined algebraic laws to one or more processes, producing a new process that may be bound to a new, or pre-existing process variable. In this way, algebraic proofs of the equivalence of process expressions may be developed.

State space exploration, equivalence testing and interactive execution operate on the LTS representation of the system being analyzed. State space exploration analysis can be used to determine key properties of a system’s LTS. These include

1. number of states and transitions;
2. presence of deadlocked states;
3. states capable of Zeno behaviors (i.e., infinite sequences of instantaneous events);
4. states that require synchronization to take place before time can progress; and
5. reachability of specific externally observable events.

Process equivalence can be tested using a number of different notions of equivalence
including syntactic equivalence, a weaker syntactic equivalence which allows renaming of process variables and simple changes in structure, prioritized strong equivalence, and prioritized weak equivalence. In the order listed, these notions of equivalence increase in computational complexity and decrease in “strength” (i.e., equate more terms).

The interactive execution feature of VERSA allows user-directed execution of process specifications. The user may interactively step through the LTS one action at a time, produce traces from random executions of the LTS, save process configurations to a stack for later analysis while an alternate path is explored, and analyze the size and deadlock characteristics of the LTS resulting from their process.

5.2 The GCSR Tool Set

To facilitate the use of the GCSR language, we have augmented the VERSA system with a graphical tool set that manages the input/output as well as refinement of graphical GCSR descriptions. We have implemented the the GCSR tool set using the GNU project C++ compiler, the Library of Efficient Data types and Algorithms (LEDA) class library, and the OSF X/Motif toolkit. We next describe the three components of the GCSR tool set: language, views, and operations.

5.2.1 Language

While displayed, GCSR descriptions are maintained in an intermediate representation based on an object-oriented data structure. In addition, to facilitate incremental and modular design, the GCSR tool set uses a simple textual language for storing GCSR descriptions into files that can be loaded and displayed.

GCSR Database Objects. Displayed GCSR descriptions are maintained as one database object of the class

```cpp
class GcsrDBObject {
    GcsrEditor *editor;

public:
    GcsrNode *nodeObjects;
    list<GcsrEdge *> edgeObjects;
    list<GcsrProc *> gcsrProcList;
}
```
The editor points to the window where the database is currently displayed; the nodeObjects is a tree of GCSR nodes; the edgeObjects is a list of GCSR edges; and the gcsrProcList is a list of GCSR processes in the database. The tree structure of the nodeObjects field reflects the nesting of the GCSR nodes: The root of the nodeObjects tree is a GCSR node drawn at the highest level of nesting. Each GCSR node has one pointer to a node drawn at the same level and a second pointer to a node nested inside of it.

The definition of the object classes for nodes and edges augments the syntactic definitions presented in Section 3.2 with graphic attributes (e.g., location) and information to facilitate navigation and updates in the database.

**GCSR Storage Language.** The GCSR tool set uses a simple text-based language to store GCSR descriptions for a later usage. Figure 5.2 illustrates the syntax of the storage language in the case of parts of the gate example of Chapter 3. The storage language syntax contains constructs to save the minimal information required to reconstruct the GCSR database. It essentially represents each GCSR node and edge through its identifier, type, graphic location, along with its GCSR defined attributes, e.g., the resource attribute for a time-consuming node and the label for an edge.

### 5.2.2 Views

The GCSR tool set is composed of two windows that the user can use: a **textual editor** window and a **graphical editor** window.

**Textual Editor.** The XVERSA text-based interface is used as the textual editor for the GCSR tool set. Within XVERSA, the user enters textual information, e.g., defines manifest constants, interactively steps through the behaviors of GCSR descriptions after they are translated to ACSR processes, and sees analysis results obtained from the VERSA system.

**Graphical Editor.** The user composes GCSR specifications using an icon-based graphical user interface editor. Figure 5.3 shows a session of the graphical editor displaying the gate example of Chapter 3. The graphical editor is composed of four components:

1. a **canvas** for drawing GCSR specifications;
Figure 5.2: Example in the GCSR storage language
Figure 5.3: A session of the GCSR tool set
2. a control menu with pull-down menus for file operations, editing operations, accessing a set of tools, option setting operations and help operations;

3. a palette of drawing modes and fast-access editing modes; and

4. a message area.

The drawing area canvas is equipped with horizontal and vertical scroll bars that allow the user to work with specifications whose size exceeds the size of the main window. The zooming facility makes it possible to view the whole GCSR specification or concentrate on a part of it.

To make modular design of complex systems easier, the graphical editor allows the user to spawn secondary graphical editor windows. These windows can be used to specify either different parts of one system or independent systems. A secondary graphical editor window is similar to the main graphical editor window except that it does not contain a message area (all messages are directed to the main window) and several control menu items are modified to reflect the inferior status of a secondary window.

The message area displays textual feedback to the user about operations they perform—e.g., successful loading of a file, consistency violation of the syntax when drawing an object.

The operations related to the items in the control menu and the palette of drawing and fast-access editing modes are described in the next section.

5.2.3 Operations

Drawing Modes

The left side of the graphical editor window contains a palette of drawing and fast-access editing modes. The editor provides separate drawing modes for all types of GCSR nodes, parallel separators and initial markers. All normal edges are drawn in the same mode. There is a separate drawing mode for exception edges. If an object requires textual attributes, a custom dialog box, specific to the object type, is brought up to enter the attributes.

Figure 5.4 illustrates a sequence of user actions needed to enter a time-consuming node into the graphic editor.

First, the Action drawing mode is selected (a). Then, the ellipse boundary is drawn by pressing the left mouse button and dragging the mouse pointer in the free space of the
Figure 5.4: Drawing a time-consuming node
drawing area. When the mouse button is released, a dialog box is displayed (b) with the entry field for the Resource attribute of the node. After the attribute is entered, the new node is displayed on the screen (c). Before the internal database of objects is updated, the system runs consistency checks to ensure that the node does not overlap other nodes or parallel separators. If a syntax violation is found, an error message is printed into the message area of the graphical editor and the drawn object is raised. If no syntax violations are detected, the system updates the internal database of objects.

Editing Modes

The most commonly used editing modes are included on the mode palette for fast access. These provide for moving, resizing and deleting objects, as well as modification of their
textual attributes. The connectivity of the specification is preserved during these graphical editing modes. When a node is moved or resized, its incident edges are stretched to move along with it. Resizing an edge allows the user to reattach one of its end points from one node to another. Moving an edge gives the ability to adjust both end points at the same time, but without changing the source and target nodes of the edge.

When a node is deleted, its incident edges are deleted as well. In the current implementation, deleting a compound node also removes its children.

The Select mode produces composite objects by treating a composite node together with its children as a single object. In this case, editing operations such as move, resize, and delete are applied to all components of a composite object.

The Refine mode implements simpler syntactic versions of the ACSR refinement transformation rules that we described in Chapter 4. We discuss in Section 5.3 the correspondence between the ACSR refinement and GCSR refinement and the effects of the Refine operation.

Other editing modes can be accessed through the Edit menu. The editing modes which are not implemented in the current version have been deactivated from selection. Most of the editing modes in the menu are self-explanatory, however, the Show Process mode deserves special attention.

In the Show Process mode, the user can click on a displayed reference node and see the definition of the referenced GCSR process. Two of the attributes to a reference node are the name of a process it represents and the flag local which indicates whether the referenced process is in the same database as the node. If the referenced process is in the same database as the reference node, a local process has its initial node highlighted. If a reference node refers to a non-local GCSR process, the referenced name is treated as a file name that contains the process specification, and a secondary editor window is created where the process specification is loaded.
The Zoom buttons at the bottom of the mode palette allow the user to increase and decrease the size of objects displayed in the drawing area. All objects in the drawing area are scaled proportionately. This operation is a display facility since it does not modify the internal representation of objects.

Control Menu

The menu bar of the editor window is designed in compliance with the recommendations of the *OSF/Motif Style Guide* with respect to positioning and names for the menus. In addition to the Edit menu described earlier, the menu bar offers four selections: File, Tools, Options, and Help.

The File menu contains the usual facilities to load and save objects. The New item deletes the objects in the current database, while the New Window item allows one to work on a new specification in a secondary editor window, keeping the current one. The new window is completely separate from all other editing windows, and specifications can be loaded and saved independently of operations in other windows. The Exit menu item is available only through the main editor window. It ends the session of the editor, discarding all secondary windows. Secondary windows, instead, have an Exit Window item in their File menus that disposes only of the current secondary window.

Items in the Tools menu provide connections with other tools in the GCSR tool set. The Check Syntax item verifies syntactic consistency of one selected GCSR process or all of the displayed GCSR descriptions. All of the syntactic rules defined in Section 3.2 have been implemented within the current version of the GCSR tool set. Syntactic error messages are printed in the message area.

The Translate item from the Tools menu allows the user to translate a GCSR specification into an ACSR specification in the VERSA syntax. The Analyze spawns the
XVERSA window, translates the contents of the graphical editor window into the VERSA syntax and passes it via a temporary file to XVERSA.

Other tools provided in the Tools menu include refinement. The user can select a specific type of refinement operation through the Refine menu; they can use the Verify Refinement item to verify the validity of refinement transformations that they perform on the displayed GCSR description; and they can confirm refinement operations to update the graphics and the internal database through the Apply Refinement. These items will be discussed in more details in the next section.

The Request Attributes option allows the user to choose between interactively entering the textual attributes of GCSR nodes and edges as they draw them, or entering the attributes at the user’s request through the Update editing mode. The Refinement options determine the type of refinements the user will be applying. The current version only supports the Hoare ordering refinement option.

5.3 Refinement in GCSR

As described in Chapter 4, there are two basic types of refinement in ACSR: event refinement and action refinement. These basic refinements can be applied repeatedly on a given ACSR process to refine it according to the transformation rules described in Chapter 4. Furthermore, a refinement must adhere to a set of conditions, e.g., event relabeling is a one-to-one, complement preserving function, and resources used in the refining process are a subset of those used in the abstract process.

Refinement in the GCSR tool set represents the two basic types of ACSR refinement
in terms of graphical transformations. In addition, the graphical transformations are supported by semantic checks to ensure that the conditions of the corresponding ACSR refinement are satisfied. The GCSR refinement operations can be selected either through the **Refine** mode in the palette, or the **Refine** menu item which offers three specific types of refinement operations: event refinement, action refinement, and process refinement.

**Event Refinement**

Event refinement consists of either relabeling a given event in a GCSR process or adding a new event. To relabel an event \( e \), the user first selects either the **Refine** mode (from the mode palette) or the **Refine Event** menu item. The user then clicks on any edge that is labeled with the event \( e \). They are then prompted with a dialog box where they can enter the new event name \( e' \). Afterwards, the system first runs a consistency check to ensure that if \( e \) is not a new event no other event has the label \( e' \). If the consistency check succeeds, the system updates the drawing and database by relabeling *all* occurrences of the event \( e \) to the event \( e' \). The consistency check and total relabeling ensure that GCSR event relabeling is a one-to-one, complement preserving function, as it is in ACSR.

If the relabeled event \( e \) is a new event, the system first checks that \( e' \) is also new. If \( e' \) is in fact new, the system adds it to the internal database and updates only the label of the selected edge.

The second type of ACSR event refinement consists of adding a *new* event. Adding an event can be done in two ways: label an unlabeled edge or insert a new event-labeled edge. The first way is supported in both the generic **Refine** mode in the editing mode palette and the specific **Refine Event** menu item. The second type, inserting a new event-labeled edge, is supported only through the **Refine Event** menu item when the user clicks on a node where they want to insert the new event. (This design decision was a result of allowing the generic **Refine** mode implement a simpler notion of action refinement through attribute updates as we see shortly.)

If the user selects an unlabeled edge, the system prompts them with a dialog box where they can enter the new event \( e \). Afterwards, the system checks that the event \( e \) has a new label and and updates the drawing of the selected edge.

If the user selects a node during a **Refine Event** operation, the system performs the following six steps:
1. A new dot node is inserted at a user selected position,
2. An unlabeled edge connects the dot node to the selected node,
3. A dialog box prompts the user for the new event $e$,
4. A consistency check is run to ensure the event is in fact new,
5. The newly added unlabeled edge is labeled with the new event $e$, and
6. All edges incoming to the selected node are redirected to point to the new dot node.

In each of the above two types of event refinement, when a violation is detected during the consistency check step, the user is prompted with a warning message that reports the consistency violation and no updates are done. In addition, the current version of the GCSR tool set implements interactive consistency checks for event refinement; that is, the user does not have to select the Verify Refinement menu item from the Tools pull-down menu. The current version, however, limits the application of event refinement to GCSR descriptions where all reference nodes are defined locally, i.e., in the same file.

When the user selects the Apply Refinement menu item, all events, including those that have been added during this round of refinement, are considered as part of the specification.

**Action Refinement**

Action refinement in the GCSR tool set is supported through two types of operations. One is by refining the resource attribute of either an action or a compound node; another is by refining an action node to a process.

The first and simpler type of action refinement is provided via the generic Refine mode in the palette. In this mode, when the user clicks on an action or compound node, they are prompted with a dialog box where they can refine the resource requirements of the node. The semantic checks for this simpler notion of action refinement are automatically conducted by the system the moment the user enters the new resource attribute. They consist of two parts:

1. Ensure that the new resources used are a subset of the original set of resources; and
2. Ensure that no released resources are shared by a parallel process.

The second consistency check makes use of the fast access information stored in a node.

The second type of action refinement implements the ACSR transformation to refine an action-prefix process, where more details are added to an action. It is supported only for action nodes and via the Refine Action mode in the Tools menu. In this case, when
the user clicks on an action node, \( A \), the system spawns a secondary GCSR editor window, \( W_{\text{new}} \), where the user can enter a refining process \( R \). At the window \( W_{\text{new}} \), the user can either load a previously drawn GCSR process from a file or they can draw a new GCSR process.

When the user selects the Verify Refinement from the Edit menu of the \( W_{\text{new}} \) window, the system runs consistency checks that depend on the type of refinement being applied. The type of the current refinement is determined by the Refinement option in the Option menu at the window where the node \( A \) resides. The current version of the GCSR tool set implements the Hoare ordering action refinement. In this case, the system runs the following two consistency checks on the refining process \( R \):

1. \( R \) uses at most as many resources as the node being refined, \( A \); and
2. \( R \) contains events that are new with respect to the events in the database where \( A \) resides.

(To support action refinement in the Egli-Milner ordering, a third consistency check must be added: the duration of \( R \) is no less than that of \( A \).)

When the user selects the Apply Refinement from the Edit menu of the \( W_{\text{new}} \) window, the system first runs the above described consistency checks. If the process \( R \) meets the refinement constraints, the system saves the process \( R \) into a file and replaces the node \( A \) with a compound node that has one nested process consisting of a reference node to the process \( R \). The Restrict attribute of the compound node is set to the \( \emptyset \), and its Close attribute is set to

\[
\text{Close} = I \cap (\text{res}(A) - \text{res}(R))
\]

where \( I \) is the parallel resource context in which \( A \) resides. This set is collected through a traversal of the nodes that contain the node \( A \).

To summarize, the Refine and Refine Action operations implement for GCSR the following ACSR refinement transformation rules defined in Chapter 4:

\[
\text{ref}^1_{(p_1, p_2)}(A': Q, I) \leadsto [R]_U \bigtriangleup_1 (\text{NIL}, \text{ref}^1_{(p_1, p_2)}(Q, I), \text{NIL})
\]

\[
\text{for } I' = I \cap (\text{res}(A) - \text{res}(R))
\]

\[
\text{ref}^1_{(p_1, p_2)}([Q]_U, I) \leadsto [\text{ref}^1_{(p_1, p_2)}(Q, I)]_U,
\]

\[
\text{where } (U - (\text{res}(\text{ref}^1_{(p_1, p_2)}(Q, I)) - \text{res}(Q))) \subseteq U'
\]

in addition to a simpler version of the refinement of an action-prefix process.
Process Refinement

Process refinement applies to either a whole GCSR process when the user clicks on the initial marker of the process, or to intermediate nodes in the GCSR description. It is provided via both the Refine mode and the Refine Process option.

When the user selects an initial marker to refine, the system spawns a new graphical editor where the user can restructure the GCSR process to an equivalent process. In the current version, the user has to draw the equivalent process from scratch. The user also has to ensure that the original and refining processes are bisimilar; they can use the Translate and Analyze options and ensure the equivalence of the two processes within the VERSA system. Automatic verification of this type of refinement is left for future work. To apply this type of process refinement, the system erases the old copy of the process and replaces it with the new process.

When the user selects a reference node to refine, a new refinement window $W_{new}$ is spawned where the definition of the referenced process is loaded. The user can then refine the process inside $W_{new}$. When the user chooses to apply the refinement, the system saves the refined GCSR process as a new GCSR process $G'$, and updates the Name attribute of the selected, abstract reference node. (Note that this strategy of creating a new copy of the referenced process allows the user to refine different occurrences of a reference node differently.) The consistency checks during the refinement of a reference node are done with respect to the database in the editor where the reference node resides.

In the generic Refine operation from the palette, refining a compound node consists of adjusting the restrict and close attributes of the compound node. This type of refinement implements the following two ACSR refinement transformation rules:

\begin{align}
& (8) \quad \text{cref}^i_{(\rho_1,\rho_2)}(Q, I) \leadsto \text{cref}^i_{(\rho_1,\rho_2)}(Q, I)_F \\
& (10) \quad \text{cref}^i_{(\rho_1,\rho_2)}([Q]_U, I) \leadsto [\text{cref}^i_{(\rho_1,\rho_2)}(Q, I)]_U, \\
& \text{where } (U - (\text{res}(\text{cref}^i_{(\rho_1,\rho_2)}(Q, I)) - \text{res}(Q))) \subseteq U' \end{align}

In the Refine Process operation from the Tools menu, refining any node except a reference node implements the following ACSR refinement transformation rule:

\begin{equation}
(1) \quad \text{cref}^i_{(\rho_1,\rho_2)}(Q, I) \leadsto \text{NIL} \end{equation}

In this case, the result of the refinement replaces the refined node with a NIL node and removes all of the outgoing edges from the refined node to maintain the well-formedness of the GCSR description.
5.4 Summary

We have presented a brief tutorial to the GCSR tool set. In addition to the regular input/output operations, the GCSR tool set includes functions to support drawing, syntax checking, refinement (in the Hoare ordering), and automated translation of GCSR descriptions to ACSR processes. The GCSR tool set provides connection to the VERSA system for analysis of GCSR descriptions.

There are several enhancements that should be added to the current tool set. One is a more elaborate diagnostic facility that would allow the user to interactively examine error messages pertinent to syntax violations, e.g., list the problematic objects and highlight them under the user's request. Other enhancements include the implementation of semantic checks for the Egli-Milner refinement; a graphical representation of ACSR equivalence preserving rewrite rules; a layout algorithm for “pretty” drawing and displaying of ACSR processes that are translated to GCSR; a simulator for GCSR descriptions; and a Help menu.
Chapter 6

Production Cell: a Case Study

The “Production Cell” example was a case study of the KorSo project [Li95]. It aimed at comparing different approaches of formal and semi-formal development methods, and checking their suitability for safety-critical and reactive systems.

In addition to being a representative of safety-critical and reactive applications, the production cell example represents a realistic, industrial real-time application, where safety requirements are essential and can be met by the application of formal methods. Furthermore, on one hand, its functionality is complex enough to require several levels of abstraction; on the other hand, its size is moderate and thus could be treated within our tool set and timing constraints.

We used the GCSR tool set to describe a distributed controller for the production cell as well as models for each of its machines. We used the refinement transformations, described in Chapter 4, to reach a detailed design solution in a modular and top-down as well as bottom-up fashion. We also made use of the modularity and hierarchy of the design solution during the analysis process. We used VERSA to verify all of the safety and liveness (or time bounded) requirements through testing, state exploration, equivalence checking, as well as informal arguments.

Chapter organization. Section 6.1 describes the production cell as reported in [Li95]. Section 6.2 overviews our design strategy. Section 6.3 presents parts of our design solution using the GCSR specification language and refinement theory. Section 6.4 analyzes the presented design solution. Section 6.5 summarizes our experience and evaluates the suitability of the GCSR design environment for the production cell’s class of applications.
Appendix C.1 discusses our design assumptions; Appendix C.2 contains the remaining parts of our design solution; and Appendix C.3 lists the analysis results from the VERSA system.

### 6.1 Informal Description

The informal description of the production cell in this section has been adapted from [Lin95]. Figure 6.1 depicts the top level view of the production cell. The system has two conveyor belts that are connected by a traveling crane, a positioning table, a two-armed robot, and a press. In addition, the system is equipped with a set of sensors (e.g., photoelectric cells) and actuators.

The main functionality of the system is to forge metal plates in the press. This is accomplished through the following cycle of operation:

1. the deposit belt conveys unforged metal plates towards the traveling crane;

2. a photoelectric cell informs the controller when a plate arrives to the end of the deposit belt;

3. the traveling crane picks up the metal plate from the deposit belt and moves it towards the feed belt where it unloads it;

4. the feed belt conveys the metal plate towards the elevating rotary table;

5. a photoelectric cell informs the controller when a metal plate arrives to the end of the feed belt;

6. the elevating rotary table receives the metal plate and rotates towards the robot;

7. the robot extends its first arm, picks up the metal plate, retracts its first arm, rotates to position its first arm in front of the press, extends its first arm, places the plate in the press, and retracts its first arm away from the press;

8. the press closes, forges the metal plate, and opens again;

9. the robot extends its second arm, picks up the forged metal plate, retracts its second arm, and rotates to position its second arm on the deposit belt where it unloads the metal plate.
The system functionality is complicated by physical constraints pertinent to the hardware: The conveyer belts and robot are positioned at different vertical heights. This requires the traveling crane and the elevating rotary table to move vertically in addition to horizontally.

To achieve its functionality autonomously, the system receives sensory information and controls the behavior of the machines through a set of actuators. In addition, to achieve a better utilization of the press, several metal plates can be in the system at different stages. To model a repetitive behavior, we assume that a metal plate that is placed on the deposit belt is conveyed again towards the traveling crane and the processing cycle is restarted.

**Problem Statement** The goal of this case study is to develop a software controller for the production cell. The controller basically serves as a system monitor and scheduler. It collects sensory information from the set of sensors in the system and instructs the machines through a set of actuators to react in such a way that the system realizes its predefined functionality.

The behavior of the system must satisfy two types of requirements: safety and liveness. **Safety requirements**— The software controller of the production cell must
• disallow a machine from colliding with another;
• limit the motion of a machine within its allowed ranges;
• never allow a metal plate to be dropped in unsafe areas, e.g., off the two belts; and
• guarantee that at most one metal plate is in the press at any time.

Liveness requirements—The production cell can have several liveness properties. A very strong liveness property requires that the software controller must guarantee that each metal plate on the feed belt eventually arrives at the end of the deposit belt after it has been forged at the press.

6.2 Design Strategy

Our design produces a distributed controller for the whole production cell. In this design, the desirable behavior is achieved through arbitration of the usage of shared resources and synchronization through the set of sensors and actuators. The design is constructed according to the following strategy.

Structured Design Solution. To make the design solution easier to understand, we follow the hardware structure of the system. We model each machine as a process and the overall system as the machine processes running in parallel. In addition, to facilitate analysis, we follow a modelling methodology that is common in Control Theory: each machine is represented by a model whose desirable behavior is dictated by a controller.

Minimal Assumptions. To make the design solution more realistic and portable, we make minimal assumptions about the hardware environment. In particular, a machine controller fully relies on the information it receives from the machine sensors. Similarly, a machine model has no built-in “intelligence” and tries to behave according to the instructions it receives from the controller even if this leads to a safety violation. A machine model, however, encompasses the timing requirements for its actions. The one basic assumption we make about a machine model is that it cannot communicate while executing an action, e.g., moving. This assumption simplifies the description of the example but can be easily removed from our design.
Modular and Hierarchical Development. To facilitate the specification and analysis process, we use our refinement theory to produce a design solution in a hierarchical and modular fashion. That is, we describe the system components at different levels of abstraction where each level show more implementation details than another and augments it in the support of the safety and liveness requirements. Overall, we followed the next design steps:

Level 1. This highest level of abstraction reflects the hardware structure of the production cell. Each machine in the production cell is modelled as one process. The overall system consists of the parallel composition of the processes which model the machines in the system.

Level 2. The second level of abstraction shows a structure for each machine process, that allows us to verify the requirements. Each machine process is refined to two communicating processes that describe the machine model and the machine controller. Each machine specification at this level is obtained from the corresponding machine specification at Level 1 essentially through the following syntactic transformations. First we use strong bisimulation to introduce the parallel structure. Second, we apply refinement transformation rules to 1) refine the parallel process by refining the machine model and controller processes separately; 2) introduce new events while refining the two processes of a machine; and 3) force synchronization on the new events between the refining model and controller processes to obtain the desirable behavior.

Level 3 and higher. From this level on, the specification reflects a better resource usage, existence of potential risks, and risk avoidance, e.g., the convoyer belts are stopped so that metal plates are not dropped, and metal plates cannot be piled up on the feed belt. A specification at any of these levels is obtained by refining the processes at the immediately preceding level.

Analysis. We use four basic techniques to verify that our design satisfies the safety and liveness requirements: equivalence checking, testing, deadlock detection, and predicate inspection.

Certain requirements are easy to express as an abstract specification that can be inspected for correctness—e.g., liveness in the traveling crane and deposit belt. When
possible, we describe the correctness specification of a machine and verify that our design for the machine is equivalent to the correctness specification. For those properties where a correctness specification is not easy to construct, we use testing to verify the correctness of our design. A tester either confirms the requirement or shows a requirement violation.

In addition, our design solution uses deadlock to model out of range motion and machine collisions. We verify these safety violations by searching for deadlocked states in our design solution. We note that all of the above three verification techniques require that the processes involved be non-Zeno and finite state. This is the case of the processes in our design solution.

The fourth method we used to verify the correctness of our design is by formulating the requirement as a predicate and then inspecting the design for its satisfiability. This method was natural for safety properties that deal with the order of occurrences of a few events. However, since it is informal, this method can not replace the other three, more rigorous analysis methods.

To make the analysis of our design solution manageable, we exploit its modularity and hierarchy. When a safety requirement involves two or more machines, e.g., machine collisions, we verify the correctness of the relevant machines grouped together. Furthermore, we try to verify requirements at a level of abstraction that provides enough details to model the behavior pertinent to the requirements. Given the properties of our refinement theory, detailed levels inherit the correctness of more abstract levels.

6.3 Design Solution

Before we present a design solution for the production cell, we first introduce our naming convention. We then summarize the functional and resource dependencies as well as timing requirements of the machines in the system.

Naming conventions. To facilitate the reading of the example, we adopt the following notations. A timing constant starts with the capital letter $T$. An event and resource name start with a lower case letter. A process name starts with a capital letter. In addition, to distinguish between specifications at different levels of abstraction, we use the suffix notation $P_i$ to denote the specification of process $P$ at the $i$th level of abstract, where level $i$ is more abstract than level $i + 1$ for $i \geq 1$. When we do not need to distinguish
between levels of abstraction, we simply use $P$.

**Design assumptions.** Our design uses two types of communication events: those that represent information from the sensors and actuators, and those that we added to synchronize between the distributed components of the controller. Figure 6.2 illustrates the inter-controller communication events. Appendix C.1 describes the two types of events and discusses how we modelled continuous information from the sensors as events that signal critical sensor values.

In addition, to model the resource usage and arbitration between the various components in the production cell, our design represents several dedicated resources, such as electric motors, as well as all those resources that are shared between components in the system. Figure 6.2 shows the shared resources. These resources are used to detect potential collisions between the sharing components. Appendix C.1 reviews the two types of resources in detail.

![Diagram of component dependencies in the production cell]

**Figure 6.2:** Component dependencies in the production cell

The original description of the production cell [LL95] did not specify any timing requirement, despite the fact that this is a *real-time problem*. We therefore introduced our
timing assumptions which represent two types of activities in the system. One is to carry out a system function such as “move the robot arm”, the second is to reserve shared resources before using them. The reservation activity allowed us to avoid using unnecessary software communication events between the various parts of the controller. Appendix C.1 describes in detail these two types of timing assumptions.

With the above notation and assumptions, we can now describe our design solution for the production cell.

6.3.1 System Structure

The high level structure of the production cell is specified by the following ACSR process

\[
ProdCell \overset{\text{def}}{=} [(\text{Press}) \parallel \text{Robot} \parallel \text{ElevRotTable} \parallel \text{FeedBelt} \parallel \text{Crane} \parallel \text{DepositBelt}) \setminus F],
\]

where the restricted events represent the communication events exchanged between the machines

\[
F = \{\text{crCanLoad, crUnloaded, rtCanLoad, rtUnloaded, prUnloaded, prLoaded}\}
\]

and the closed resources represent all of the resources in the production cell and which are listed in Table C.2. We next describe various levels of abstraction in the design of the deposit belt, traveling crane and feed belt. We included the detailed specification of the remaining machines in Appendix C.2.

6.3.2 Deposit Belt

Figures 6.3 (a), (b) and (c) show three levels of abstraction for the deposit belt. At the first level, Figure 6.3 (a), the GCSR process DepositBelt_1 does not commit to any software structure for the deposit belt. It basically describes the resource usage and communication with other system components as follows: The deposit belt is constantly running; this is described by the fact that the motor of the deposit belt, the \text{dbMotor} resource, being constantly used. Initially, the deposit belt is ready to receive the event (blEndBl?, 1) from the photoelectrical cell, which indicates that an unprocessed metal plate reached the end of the belt. When the deposit belt receives this event, it tries to send the event (crCanLoad!, 2) to the crane after which it moves back to its initial state, ready to communicate with the photoelectrical cell again.
At this first level of abstraction, since the belt is constantly running, there is a risk of dropping a metal plate that reaches the end of the deposit belt. Such a risk happens when the deposit belt does not manage to send the event `crCanLoad` to the crane. This risk will be eliminated by refining the abstract specification `DepositBelt_1` to one where the belt is stopped if communication with the crane is not possible when needed.

Figure 6.3: Deposit belt at levels 1 (a), 2 (b) and 3 (c)

The second level of abstraction, Figure 6.3 (b), shows the software structure of the deposit belt: it consists of a process for the controller, `DBController_1`, that is running in
parallel with a process for the belt model, $DBModel$. At this level, process $DepositBelt_2$ is obtained from the process $DepositBelt_1$ through strong bisimulation. Note that since the process $DepositBelt_2$ is strongly equivalent to the process $DepositBelt_1$, it inherits the potential safety violation of dropping a metal plate.

At the third level of abstraction, the process $DepositBelt_2$ is refined by 1) refining each of its two parallel processes separately (Figures 6.3 (d) through (g)) and 2) forcing synchronization between the two refined parallel processes (Figure 6.3 (c)). This level uses the Hoare ordering transformation rules to 1) introduce new events, 2) eliminate behaviors, 3) reduce resource usage, and 4) force synchronization.

The controller of the deposit belt, process $DBController_1$ of Figure 6.3 (b), is transformed to the strongly equivalent process $DBController_2$ in Figure 6.3 (d) which allows us to distinguish two operation modes of the belt in the next level of abstraction. The process $DBController_2$ is then refined to the detailed controller, process $DBController_3$ in Figure 6.3 (e), to eliminate the risk of dropping a metal plate. It is derived as follows: 1) the two local events $dbStop$ and $dbStart$ are added to communicate with the deposit belt’s model; and 2) the loop on event $blEnddb$ is eliminated. The event $dbStop$ is sent to the model when the metal plate reaches the end of the deposit belt but the event $crCanLoad$ can not be sent to the crane. (Note that the priorities are used to arbitrate the choice between the two events as described.) Once the event $dbStop$ is sent, the event $blEnddb$ can no longer be received; in this case, when the belt is not running, the sensory event $blEnddb$ is redundant since it would indicate the presence of the same metal plate at the end of the belt. The event $dbStart$ is sent to the belt model to restart the deposit belt, if it was not running, right after the event $crCanLoad$ can be sent to the crane.

The model of the deposit belt, GCSR process $DBModel$, is constructed in a fashion similar to the controller. We note here that the fact that our notion of action refinement is a relation is essential. It allowed us to refine the two occurrences of the action $\{(dbMotor, 1)\}$ differently: one instance is refined to itself and the second to the $\emptyset$ action which indicates that the belt is stopped.

The last refinement step of the deposit belt, consists of forcing synchronization on the $dbStop$ and $dbStart$ events between the processes $DBController_3$ and $DBModel_3$. The resulting, detailed specification is described by the GCSR process $DepositBelt_3$ in Figure 6.3 (c).
6.3.3 Traveling Crane

Figure 6.4 shows three levels of abstraction for the traveling crane. At the first level,

Figure 6.4 (a), the traveling crane is described as the process *Crane_1* where the crane is initially moving and ready to receive the event *crCanLoad* from the feed belt. When the event is received, the traveling crane may move further before it sets its gripper magnet on (event *crMagOn*) which indicates that the crane has picked up the metal plate. The crane then may move further until it reaches the deposit belt (event *crAtFb*). At this time, the crane tries to reserve the resource *fb*, the end of the feed belt where the metal plate will be dropped. Any time afterwards, the crane sets its gripper magnet off (event *crMagOff*) and sends the event *crUnloaded* to the deposit belt to indicate that a metal plate has been unloaded. At this first level of abstraction, the behavior of the crane is approximated; in particular, process *Crane_1* gives no details about the time required to travel between the belts nor which direction is the crane moving. The essential property at this level is that the crane sets its gripper magnet off only after reaching the feed belt; that is, the crane
will not drop the metal plate on its way to the deposit belt. This is one of the crane’s safety requirements.

The second level of abstraction, Figure 6.4 (b), introduces a software structure to the crane that is strongly bisimilar to the first level of abstraction. The final, detailed description of the traveling crane, process $Crane_3$ of Figure 6.6 (c), forces synchronization between the controller and model processes of the crane; it is obtained through the Hoare ordering transformations. Figures 6.5 and 6.6 show, respectively, the specifications of the controller of the traveling crane at the first and third levels of abstraction$^1$. Table 6.3.3 shows a few steps in the transformation of $CrController_1$ to the process $CrController_3$ using the Hoare ordering transformation rules together with the strong bisimulation axioms.

The detailed description of the crane controller, process $CrController_3$ in Figure 6.6, satisfies an additional safety property we have imposed in our design: the crane controller sets the gripper magnet off only after actually reserving the resource $fb$. This property ensures that metal plates are not piled up on the end of the feed belt near the crane. As we will see in the next section, the resource $fb$ is available only when the feed belt has conveyed the metal plates away from the portion near the crane.

Figure 6.5: Controller for the traveling crane at level 1

Figure 6.7 shows the specifications of the traveling crane’s model at the third and fourth level of abstraction. The process $CrModel_4$ of Figure 6.7 (b) refines the process $CrModel_3$ (a) by introducing the new event $crCrash$ which is used to test one of the safety requirements of the crane— the traveling crane should not move beyond its limits. The new event ($crCrash!, 1$) is sent whenever the crane model does not receive on time a signal to stop (i.e., events $crStoph$ and $crStopw$) from the controller process $CrController_3$. After

$^1$We have skipped the second level of abstraction and numbered the processes to match the levels of the overall crane description for simplicity.
the \textit{crCrash} event is sent, the crane model halts its execution; this is described by the Nil node in Figure 6.7 (b). By operational semantics, if this safety violation happens, the whole specification of the traveling crane, \textit{Crane}_4, will also halt. The overall crane specification at the fourth level is defined by the process

\[ \text{Crane}_4 \overset{\text{def}}{=} (\text{CrController}_3 \parallel \text{CrModel}_4) \setminus F \]

where the set of restricted events

\[ F = \{ \text{crLower}, \text{crLow}, \text{crLift}, \text{crHigh}, \text{crStopv}, \text{crToFb}, \text{crOnFb}, \text{crToDb}, \text{crOnDb}, \text{crStoph} \} \]

The process \textit{CrModel}_4 can be obtained from the process \textit{CrModel}_3 by the following transformation steps: 1) apply strong bisimulation to introduce Choice sub-processes at
the eight points where an event \( crCrash \) must be added; for each Choice subprocess, 2) refine one of its branches by first inserting the event \( (crCrash, 1) \) and then refining the remaining process to the NIL process; and 3) refine the remaining branch of the Choice subprocess to itself. These transformation steps are valid refinement steps in the Hoare ordering. We will revisit this process in Section 6.4 where we verify the safety requirement for the crane through testing for the occurrence of the event \( crCrash \).

**Design history.** It is worth reporting how we reached the current design since it affected the way we designed the robot controller in terms of using the shared resources \( press \) to synchronize with the press and \( er \) to synchronize with the elevating rotary table. In addition, the design of crane also allowed us to value the transformation rule for refining two parallel processes that share resources (rule (5) in Definition 4.3.5) as well as this rule’s implementation within the GCSR tool set.

Figure 6.8 describes the steps we originally followed to design a controller for the traveling crane. In this design, the crane is \textit{impatient} at the abstract level: it grabs the resource \( fb \) the moment it reaches the feed belt. At the second level, we relaxed this requirement by refining the time-consuming node to a process that can wait if the resource
is not available. In this design, the overall resource usage step still takes $Tcf$ time units even if the crane does not manage to grab the resource at all. At this stage of the design, we forgot the fact that the resource $fb$ is shared with the feed belt which is running in parallel with the crane and thought that our design looks as shown in Figure 6.8 (b).

Later when we started the analysis stage, we found out that despite the fact that we relaxed the usage of the resource $fb$ in the crane, the crane and the feed belt still deadlock. As we examined one of the sample deadlocked behaviors produced by VERSA, we found out that in the actual design the Compound node has the Close attribute set to $\{fb\}$, as opposed to the $\emptyset$ as shown in Figure 6.8 (b). This is due to the transformation rule (5):

$$\text{cref}_{\{p_1,p_2\}} (Q_1 || Q_2, I) \sim \text{cref}_{\{p_1,p_2\}} (Q_1, I') \parallel \text{cref}_{\{p_1,p_2\}} (Q_2, I')$$

where $I' = I \cup (\text{res}(Q_1) \cap \text{res}(Q_2))$

This rule ensures that shared resources are not released accidentally. Recall that this
condition was necessary to ensure that separate refinements of parallel processes do not release deadlocks in the original specification. Our forgetful refinement was caught due to the automated enforcement of rule (5) within the GCSR tool set. The clarification forced us to go back and revise the original design.

6.3.4 Feed Belt

Figure 6.9 shows the first three levels of abstraction for the feed belt. Figure 6.10 shows the controller and model for the feed belt at level 3. The detailed specifications can be derived using transformation steps similar to the deposit belt design. Note that the controller of the feed belt at level 3 has the event rtCanLoad which is used to communicate with the elevating rotary table.

The detailed controller of the feed belt, process FBController_3 in Figure 6.10, satisfies two safety requirements: 1) no metal plate is dropped on the floor, and 2) metal plates are not piled up at the end near the crane. The first safety requirement is part of the original problem and we added the second one.

Process FBController_3 initially idles until it receives the signal that the crane has
unloaded a plate on the belt (event \textit{crUnloaded}). At this time it acquires the end of the feed belt for $T_f$ times units, the time to convey the metal plate away from the end near the crane. Afterward, the controller waits until a metal plate reaches the belt’s end near the rotary table; this is marked by the reception of event \textit{blEndFb}. At this time, the controller stops the belt by sending the event \textit{fbStop} and does not restart the belt until it can communicate with the rotary table through the event \textit{rtCanLoad}. This event indicates that the rotary table can intercept the metal plate, after which the belt can be restarted to convey more metal plates. (The reader is referred to Appendix C.2 where they can see that the rotary table will be able to receive the event \textit{rtCanLoad} only when it is facing the feed belt.) This way, the controller guarantees that no metal plate is dropped on the floor.

During the above cycle of operation, the controller can receive from the crane controller several events \textit{crUnloaded} which indicate that several metal plates are unloaded by the crane. Note, however, that such events are received only when the the belt controller is in an idle state, i.e., a \textit{Wait} node in Figure 6.10. This way, the controller guarantees that no metal plates get piled up near the crane side of the feed belt. To verify this safety requirement formally, recall that the crane controller must reserve the end of the feed belt, resource \textit{fb}, before sending the event \textit{crUnloaded}. Thus, a violation of this
safety requirement can be detected by a deadlock due to an attempt to use the \textit{fb} resource simultaneously by the processes \textit{CrController$_3$} and \textit{FBController$_3$} running in parallel. Appendix C.3 shows the formal analysis in VERSA.

### 6.4 Design Solution Analysis

Any design solution to the production cell must satisfy the four safety requirements and one liveness requirement described in Section 6.1. We verified the first two safety requirements using testing and state exploration, and verified the remaining two safety properties by inspecting the designs for simple predicates over the timed occurrences of relevant events. We verified liveness requirements using equivalence to correctness specifications, and verified bounded delay requirements through testing. Appendix C.3 contains all the analysis results from VERSA. We next elaborate on some of the analysis steps.
6.4.1 Safety analysis: Part 1

Recall that our design solution uses special events and resource related deadlocks to mark potential collisions between machines and out of range motions as follows:

- the event \(crCrash\) to detect any collision between the crane and the conveyer belts (by knocking against them from above or laterally);
- a deadlock due to the shared resource \(ert\) to detect any collision between the robot’s first arm and the elevating rotary table, and the resource \(press\) to detect any collision between the robot’s arms and the press;
- the events \(crCrash, rtCrash, rbCrash, ra1Crash, ra2Crash\) and \(prCrash\) to detect any out of range motion for the crane, elevating rotary table, robot, robot’s arms, and the press.

We verified this part of the safety requirements by developing tester processes that deadlock when they detect the occurrences of a crash event. This procedure can be done separately on the machines involved or collectively on the whole system as illustrated below:

\[
\text{SafetyTest} \triangleq \emptyset : \text{SafetyTest} \\
+ (rbCrash, 1).\text{NIL} + (ra1Crash, 1).\text{NIL} + (ra2Crash, 1).\text{NIL} \\
+ (crCrash, 1).\text{NIL} + (rtCrash, 1).\text{NIL} + (prCrash, 1).\text{NIL}
\]

We used the automated GCSR to ACSR translation and VERSA to verify that our detailed design running in parallel with the process \(\text{SafetyTest}\) is deadlock free for the timing assumptions listed in Table C.3. More specifically, the process

\[
(\text{ProdCell}_3 \parallel \text{SafetyTest})\{rbCrash, ra1Crash, ra2Crash, crCrash, rtCrash, prCrash\}
\]

is deadlock free. Appendix C.3 contains the VERSA analysis results for the crane in-range motion, robot and press collision-free operation, and the whole system in-range and collision-free operation.

6.4.2 Safety analysis: Part 2

We used deadlock freeness and predicate properties to verify the remaining two safety requirements—mainly never allow a metal plate to be dropped in unsafe areas and guarantee that at most one metal plate is in the press at any time.
Deposit Belt. The detailed specification of the deposit belt, process \textit{DepositBelt,3} of Figure 6.3 (c), ensures that no metal plate is dropped for two reasons: 1) as argued in Section 6.3.2, the controller obviously produces a safe behavior by stopping the belt whenever the crane can not pick up the metal plate at the end of the belt; and 2) as \textit{VERSA} confirms, the controller in parallel with the model is a non-Zeno and deadlock free process, i.e., the controller synchronizes with the model as needed. Thus, the controller in parallel with the model produces a safe behavior.

Traveling Crane. A metal plate will be dropped from the crane if this latter turns its magnet off (marked by sending event \textit{crMagOff} in Figure 6.4) before reaching the feed belt (marked by receiving event \textit{crAtFb}). This safety requirement can be described by the following predicate over a trace \(r\):

\[
S(r) \overset{\text{def}}{=} \forall 0 \leq t_3 \leq \text{dur}(r).
\]

\[
\left( \text{crMagOff} \subseteq \text{event}(r,t_3) \right) \Rightarrow
\]

\[
\text{crAtFb} \text{\ crMagOff} \subseteq \text{event}(r,t_3)
\]

\[
\lor
\]

\[
\exists 0 \leq t_1 < t_3, \text{crAtFb} \subseteq \text{event}(r,t_1) \land \forall t_1 \leq t_2 \leq t_3, \text{crMagOn} \not\subseteq \text{event}(r,t_2)
\]

Recall that the event \textit{crMagOn} indicates that the crane picked up a metal plate from the feed belt and it occurs before any occurrence of the event \textit{crAtFb}. This event is used in property \(S(r)\) to rule out behaviors where the crane starts operating safely and then fails, that is, behaviors where the sequence of events are in the order

\[
\text{crMagOn \ crAtFb \ crMagOff \ crMagOn \ crMagOff} .
\]

It is easy to inspect the abstract specification of the crane, process \textit{Crane,1} of Figure 6.4, and see that it satisfies property \(S(r)\). Since properties are inherited from the specification to the implementation in the Hoare ordering refinement, we can infer that the detailed specification of the traveling crane also satisfies the above safety property.

Appendix C.3 contains an alternative way to verify this property in \textit{VERSA}. It uses a tester process that crashes if the events are produces in an unexpected order.

Recall that we argued that our design also satisfies the additional safety requirement that no metal plates be piled up in the portion of the belt near the traveling crane. As
explained in Section 6.3.4, a violation of this requirement can be detected by a deadlock in the process

$$
[(\text{CrController}_3\|\text{FBController}_2)\setminus\{\text{crUnloaded}\}]\{\text{fb, crMotor, fbMotor}\} \cdot
$$

We used VERSA to verify that the above process is deadlock free for the timing assumptions listed in Table C.3. Note that we only needed to verify that the controllers of the crane and feed belt function properly in parallel, i.e., do not deadlock. As we saw in the first part of the safety requirements, since these processes also function properly with the machine models they control, we can infer that the overall design of the crane and feed belt is deadlock free.

**Robot.** A robot can drop a metal plate in two unsafe areas: between the elevating rotary table and the press, and between the press and the deposit belt. The first safety violation is detected by the occurrence of the event $m1\text{MagOff}$ before the event $r\text{AtPr}$. The second safety violation is detected by the occurrence of the event $m2\text{MagOff}$ before the event $r\text{AtDb}$. It is straightforward to express each of these safety properties as predicates that order the timed occurrences of the relevant events. The overall safety property is the conjunction of the two.

**Press.** The remaining two safety requirements for the press, besides no out of range motion, are: 1) the press can close only when no robot arm is positioned inside, and 2) the press can contain at most one metal plate at a time.

In our design, a violation of the first safety requirement is detected by a deadlock due to the $\text{press}$ resource. As described in Appendix C.2, when the press is moving or is pressing a metal plate, it uses the resource $\text{press}$. In addition, when a robot arm is inside the press, the robot controller holds the resource $\text{press}$. Thus, a violation of this safety requirement can be detected by a deadlock in our design and more precisely in the process

$$
[(\text{RController}_3\|\text{Press}_3)\setminus\{\text{prUnloaded, prLoaded}\}]\{\text{press, rMotor, db}\} \cdot
$$

Note that since the robot model does not use the $\text{press}$ resource and functions safely with the robot controller, we only needed to verify that the robot controller functions safely with the press.

To verify that the press can contain at most one metal plate, it suffices to inspect the press and robot controllers (Figure C.7 and Figures C.3 and C.4, respectively) where the
event \textit{prLoaded} is always preceded by the event \textit{prUnloaded}. (This is a safety property that can be expressed as a simple predicate.) Since the press and robot processes do not deadlock when running in parallel, the synchronization on these events is guaranteed in the correct order and, therefore, the safety property holds.

\subsection{6.4.3 Liveness Requirements}

We verify the liveness requirements on forging metal plates in a modular and incremental fashion. Since several machines in the system operate under timing assumptions, we verify bounded delay whenever a machine has timing assumptions. For those machines that operate under no timing assumptions, we verify liveness.

We adapt to our process algebraic setting the model checking methodology used in the CSL contribution in the production cell case study [NW95]. For each machine controller, we verify that

“when a metal plate starts at an initial position, some metal plate will eventually (or within a bounded time) reach a final position” and

“each machine controller will eventually (or within a bounded time) return to its initial state.”

The first part of the requirements does not specify that the same metal plate reaches the final position. This is due to the fact that in our design solution metal plates do not have identifiers. However, given the assumption that the rotary table, each robot arm, and the press can only hold one metal plate at a time, all metal plates will eventually get forged, i.e., the overall liveness property follows.

We illustrate our analysis method to verify liveness for the deposit belt and to verify bounded delay for the robot and press running together.

\textbf{Deposit Belt.} Liveness in the deposit belt is guaranteed as follows: whenever a metal plate reaches the end of the feed belt, it is eventually picked up by the traveling crane. A correctness specification for the deposit belt is described by the ACSR process

\[
DPLiveness \overset{\text{def}}{=} \text{rec } X. \ (\emptyset : X \\
+ \text{blEndDb,1). } \ ((\text{crCanLoad},2).X \\
+ (r,2).\text{rec } Y. (\emptyset : Y + (\text{crCanLoad},2).X))
\]
In the above process, whenever the event \( (b E n d D b, 1) \) is received there are two possible behaviors. One is when the event \( (c r C a n L o a d, 2) \) is immediately sent and the process returns to its initial state; in this case, the liveness property is satisfied. The second behavior, which has a lower priority, is when the process idles for some time before it sends the event \( (c r C a n L o a d, 2) \) and returns to its initial state.

First we note that the above process is supposed to run in parallel with the crane controller with which it synchronizes through the event \( c r C a n L o a d \). Thus, when running in parallel with the crane controller, the prioritized semantics will ensure that when the event \( (c r C a n L o a d, 2) \) is possible, it will occur at the earliest time. This event is possible when the crane controller is ready to receive it. Thus, under the assumption that the crane controller eventually will be ready to receive the event \( c r C a n L o a d \), the process \( D B L i v e n e s s \) satisfies the liveness condition.

We used the automated GCSR-ACSR translation and VERSA to verify that the following equivalence holds:

\[
DepositBelt \& \setminus \{ dbMotor \} \approx_{\pi} DBLiveness
\]

We can therefore conclude that, under the assumption that the crane controller eventually will be ready to receive the event \( c r C a n L o a d \), our design of the deposit belt satisfies the liveness property. During the analysis of the traveling crane, we find out that this latter is initially ready to receive the event \( c r C a n L o a d \) and it eventually returns to its initial state, provided the feed belt is ready to accept the event \( c r U n l o a d e d \).

We carry out the above liveness analysis further to the feed belt and elevating rotary table. The last assumption about the liveness of these four components relies on the robot eventually sending the event \( r t U n l o a d e d \), which marks the fact that the robot picked up the metal plate from the table. As we see next, the robot can send this event within a bounded delay constraint. Thus, the whole design satisfies the liveness requirements.

**Robot and Press.** We prove that our detailed designs of the robot and press running in parallel satisfy a bounded delay between the time the robot picks up a metal plate from the elevating rotary table (indicated by the event \( r a i M a g O n \)) and the time the robot first arm returns in front of the elevating rotary table (marked by the event \( r A t R t \)). Under the assumption that the designs of the robot and press function safely and according to the required ordered steps of operation, this bounded delay property implies that the robot
and press running in parallel satisfy the liveness requirements.

First we note that it is easy to inspect that the designs of the controllers for the robot and press in deed function according to the required ordered steps of operation. (The GCSR specifications of these two controllers are shown in Figures C.3, C.4, and C.7.) Second, we have seen during the safety analysis that the robot controller is deadlock free when running in parallel with the robot model; so is the press controller when running in parallel with the press model. That is, the robot and press models will comply with the instructions of the controllers.

In addition, since the event \( rAtRt \) is internal to the process \( Robot_3 \), we refined the robot controller \( RController_3 \) by adding the new event \( rBackAtRt \) right after the event \( rAtRt \) is received. This event therefore describes the fact that the robot first arm is back facing the elevating rotary table. We used this event to verify the bounded delay property. Call the resulting robot specification \( Robot_4 \). Furthermore, let us abstract out irrelevant events by relabeling them at the highest level to \( \tau \) as follows:

\[
RPDesign \overset{\text{def}}{=} [(Robot_4 \parallel Press_3) \backslash F];
\]

\[
\%[\tau/rtUnloaded, \tau/ra1MagOn, \tau/ra2MagOff, \tau/ra2MagOn, \tau/ra2MagOn]
\]

\[
I = \{ert, press, rMotor, db\}
\]

\[
F = \{prLoaded, prUnloaded\}
\]

Let us examine the timing requirements and delays for the concurrent execution of the robot and press. The robot must use the press exclusively during three activities:

1) reserve the press; 2) extend and retract near the press; and 3) load and unload the press:

\[
Tx = 2 \ast (Trp + Trh1 + Trh2)
\]

The robot, on the other hand, can execute the following activities in parallel with those of the press: 1) rotate; 2) deposit the metal plates on the deposit belt; 3) pick up a new metal plate from the rotary table; and 4) extract and retract when not near the press. Thus, the robot must wait before it acquires the press resource each time its set of parallel activities are shorter than those of the press. Following the operational description of the robot and press, one can determine the following waiting time:

---

\( ^2 \)The \( \tau \) relabeling is a facility provided in VERSA. It is done over the labeled transition system of the process and is not part of our syntactic event refinement over processes.

139
\[ Tw = \max(Trh1 + Trr1, Tpr + Tpm + Tpb) \]
\[ \quad + \max(Trr2 + 2*Trh2 + Trr3, Tpb) \]
\[ \quad + \max(Trr2 + Trr2 + Trr1, Tpl) \]

We can therefore conclude that the total operation of the robot and press will be within the time bound of \( Tx + Tw \).

Now, to verify that the abstract design operates within a bounded delay, we first construct a tester process that enforces the property. We then verify that the synchronous execution of the tester process and the abstract design is deadlock free.

It is easy to inspect that the ACSR process \( RPTestDelay \), below, can receive the event \( rBackAtRt \) only within \( Tx + Tw \) time units after receiving the event \( rMagOn \):

\[
RPTestDelay \overset{\text{def}}{=} \text{Wait} \triangleq \infty (\text{NIL, NIL, (rMagOn, 1)}, SDelay) \\
SDelay \overset{\text{def}}{=} \text{Wait} \triangleq _{Tx+Tw} (\text{NIL, NIL, (rBackAtRt, 1)}, RPTestDelay)
\]

We used VERSA to verify that the process \( \{RPDesign \parallel RPTestDelay\} \setminus \{rMagOn, rBackAtRt\} \) is non-Zeno and deadlock free; thus, our design of the robot and press, together, satisfy a bounded delay requirement.

### 6.5 Evaluation

We conclude our attempt to the design of the production cell example by evaluating the GCSR/ACSR specification, refinement, and analysis methodology, as well as the GCSR/VERSA tool sets.

#### 6.5.1 Modeling

The temporal constructs, resources as well as priorities allowed us to design a solution that encompasses run-time resource constraints, and thus more dependable and realistic. In particular, the notions of resources allowed us to analyze certain safety requirements in a natural way; for example, to verify the absence of a collision between the robot and the press, we verified that our design does not have a deadlock due to an attempt by the robot and the press to use simultaneously the press resource.
The graphical notation did help us visualize the overall structure of the system, its component dependencies (in terms of resources and communication), and the flow of control within each component. The semantically tight connection between the graphical and textual notations allowed us to mix GCSR and ACSR descriptions freely. Overall we feel that GCSR is suitable for real-time systems in the class of the production cell.

**Semantic difficulties.** The main difficulty we encountered using the GCSR/ACSR paradigm is how to choose between synchronization through resource usage versus communication events. This difficulty arose due to two reasons. One is that we chose to develop a distributed controller for the production cell. A second reason is that we tried to model the system as a close analogue to the real system; in particular, we tried to minimize software communications, i.e., communications that are not provided through the set of sensors.

**Syntactic difficulties.** The two main notational inconveniences we encountered using GCSR are: 1) the lack of a way to designate entrance to different initial nodes in a GCSR process, and 2) the lack of a template notation.

As we experienced with the production cell example, it is common that a specification gets large and cumbersome. One way to simplify such a specification is to divide it into subprocesses and use the Reference node to produce a visually more compact and structured specification. However, in a few cases of the production cell components, we could not divide a specification due to control flow dependencies (i.e., edges) that result in a process being entered at more than one initial node. It would be therefore fruitful to investigate an extension of the exception edge feature to designate starting a process at different nodes depending on a synchronization through a specific event.

From our experience with the production cell, we also noticed that several parts of the specification only differ in the resource and event names or timing assumptions, e.g., the two robot arms. With a template notation in the language, we could have defined one generic component and instantiated it with different parameters, i.e., resource and event names, timing assumptions, and priorities.
6.5.2 Refinement

**Syntactic transformation rules.** The syntactic approach to the refinement of the production cell facilitated the hierarchical design since, at any time, we only had to work on a concise description of the system behavior. The transformation rules are straightforward to apply whenever the specification has the structure of the final design. However, in most cases, the axioms of strong bisimulation are necessary to transform a highly abstract specification to a more structured design. These axioms are however difficult to implement graphically. Currently within the GCSR tool set, the user constructs the equivalent implementation from scratch. Ideally, the user should be allowed to apply specific graphical transformations to restructure the abstract specification to an equivalent design.

A special note about the usefulness of the Hoare ordering transformation rule which rewrites any process to the NIL process: this rule allowed us to eliminate unimplementable behaviors, for example, to force a delay between the time the crane picks up a metal plate and the time it reaches the feed belt. This delay was not represented in the abstract level.

**Action refinement.** In this case study we frequently used a simple notion of action refinement where an action is refined by refining its resource attributes—as oppose to replacing it with a process. This is due to the simple individual activities in the production cell. The production cell example confirmed our claim that action refinement as a relation, as opposed to a function, is important within the context of GCSR/ACSR. In addition, we used resource refinement to model a better or safer resource utilization, e.g., to avoid dropping a metal plate a conveyer belt is stopped and its motor is not used.

While we used action refinement infrequently in this case study, we still believe that it is an intuitive representation of the concept of adding details to an abstract, time consuming activity. This is at the heart of top-down development methodologies.

**Event refinement.** We have extensively used refinement through the addition of new events. This type of refinement allowed us to introduce 1) design dependencies between components and 2) “watch-dogs” to test for safety violations, e.g., to detect an out of range motion.

In this case study, we did not need event relabeling during the refinement steps. This is essentially due to the fact that we were constructing the whole specification from the beginning. Event relabeling is however more of a tool for software component reuse where
the interface of a component must be relabeled to fit a new context, i.e., specification in which it is reused.

In addition, the case study confirmed our intuition that event refinement through relabeling and addition of new events is a sufficient concept of refinement for instantaneous activities. In other words, we did not find a need to refine an event to a process, as action refinement would provide.

Refinement semantics. During the design of the production cell, we relied on refinement in the Hoare ordering. This ordering was sufficient as we were interested in preserving safety properties. However, since the Hoare ordering extension does not preserve deadlock freeness, we had to verify this property at the lowest level of detail.

Because of the limited compositionality of our notion of refinement in the prioritized semantics, we carefully applied our refinement steps. In particular, to get around the lack of compositionality of refinement with the Choice operator, we made sure that when a newly added event determines a Choice, the two behaviors after the Choice refine the same abstract behavior.

While we managed to describe the production cell in a modular and hierarchical fashion, we feel that our design was developed in an ad hoc way. Initially, we started following the detailed, informal specification of the production cell. This gave us an idea about how a couple of the controllers are supposed to be structured. While designing these controllers, we often found ourselves lost in the details. We therefore went back to the informal specification, bypassed most of the details, and formulated the simplest specification for each machine. Later when we started refining the abstract specifications, we used the detailed controllers as our guidelines to apply the refinement as well as equivalence preserving transformations. This design strategy took more time than necessary, as we spent some time undecided about which details can be abstracted and still have a specification that portrays an "essential" behavior of a component. The notion of "essential" behavior may, on one hand, dependent on the properties which must be verified at a particular design stage; on the other hand, it may depend on the interface with other components in the system. It would be fruitful to look into a structured methodology that encompasses such information and guides the application of the refinement rules.
6.5.3 Analysis

To verify that our design solution satisfies the safety and liveness requirements, we have used testing, state exploration, and equivalence checking. In addition, we found it natural to express several safety requirements as predicates that we informally argued hold by inspecting the design components; a rigorous proof of the predicates requires a model checker.

During the analysis stage, we tried to take advantage of the modularity and hierarchy of the design solution as much as possible. In this process, we often found it hard to determine which property is better verified at one level of abstraction as opposed to another. It is often easy to determine when one level of abstraction lacks details to verify a certain property, e.g., the absence of events used to express the property. However, it is difficult to determine when one level of abstraction provides for, say, a more efficient or sufficient analysis.

6.5.4 GCSR/VERSA Tool Sets

The GCSR/VERSA tool sets have been essential in this case study during the specification and analysis stages. Refinement within the production cell example motivated us to implement the simpler version of action refinement through refinement of the resource attributes. As we noticed in the production cell example, most actions are simple and, thus, do not require refinement through a process. The simple actions are instead refined by updating the resource attribute of their time-consuming nodes. This simplifies the graphical description, as opposed to going through the regular action refinement and substituting a compound node for the refined time-consuming node.

The connection between the GCSR and VERSA tool sets through the automated translation of GCSR descriptions to ACSR processes was vital during the analysis stage. However, a gap remains between the analysis results in VERSA and the GCSR descriptions. For instance, when the analysis in VERSA returns a sample trace that illustrates a failure of a certain test, we had to manually trace the execution within the GCSR descriptions—a process that can be tedious. It is therefore fruitful to augment the GCSR tool set with a simulator that helps to visualize the execution paths in a GCSR description. We have laid out the ground for such a feature by defining the operational semantics in terms of the GCSR nodes and edges.
Chapter 7

Conclusion

We have presented the Graphical Communicating Shared Resources, GCSR, a specification language with formal semantics for the modeling, refinement and analysis of real-time systems.

GCSR is a first step towards incorporating software engineering practices into the CSR paradigm. This is achieved, on one hand, by the graphical syntax of GCSR which adopt the intuitive concepts of nodes and edges in state diagrams, a popular informal specification language within the software engineering community. On the other hand, the refinement theory for GCSR supports a top-down development methodology, also popular within the software engineering community.

GCSR is distinguished from other graphical, formal languages by its explicit support of resources and priorities, its structured modular and hierarchical thus scalable syntax, as well as its tight correspondence with a timed process algebra, ACSR [LBGG94]. The GCSR-ACSR correspondence allows GCSR to benefit from process algebraic analysis techniques such as equivalence checking, state space exploration, and testing. The operational semantics of GCSR, either directly through a mapping to labeled transition systems or indirectly through a mapping to ACSR, makes it possible to execute a GCSR specification to examine sample behaviors.

Refinement in ACSR and thus GCSR is supported through an ordering relation over process terms whose semantics is defined by extending an ordering relation over traces. During the design of the two refinement relations we kept in mind three main goals. One is that trace refinement must allow properties to be inherited between related traces. A second goal is that the extension of trace refinement must guarantee that ACSR refinement
is *compositional*, which allows the modular application of refinement. Our third goal, which has practical motivations, is that ACSR refinement can be represented through a set of operators; this makes the refinement theory provide both semantic support for a hierarchical design methodology, and tools to derive detailed specifications incrementally within a design environment.

Typically, properties about a real-time system behavior, e.g., safety property are expressed in terms of the timed occurrences of particular specification events. It was therefore natural to choose a trace refinement relation that preserves the timed occurrences of events from an abstract trace to a refining trace. After this step, we defined two extensions of the trace refinement that reflect two common notions of “implementation” and that are based on ordering relations in the powerdomain theory [Smy78, Gun92]: the Hoare ordering and Egli-Milner ordering. We then set to examine the compositionality and syntactic characterization of these two extensions.

In the unprioritized semantics, we found out that both extensions of the trace refinement are, under reasonable restrictions, compositional with respect to all ACSR operators except for the Parallel operator. The restriction for the Parallel operator is more stringent due to resource sharing. To characterize each of the two extended trace refinements, we defined a syntactic transformation scheme that rewrites an ACSR process to a more detailed process in a modular way and using event relabeling, event introduction, and action refinement as the basic rewrite operators. In addition, the transformation scheme uses the ACSR Close operator to loosen the restriction in the presence of a Parallel operator and thus widens the applicability of the syntactic refinement approach to all ACSR processes. Furthermore, the transformation scheme outlines a set of primitive operators that are easily represented as graphical refinement operations for GCSR. In the Hoare ordering, the transformation scheme augmented with the axioms of strong bisimulation [BGCL93] is complete in the absence of recursion. In the Egli-Milner ordering, the transformation scheme is complete modulo trace equivalence.

In the prioritized semantics, however, several operators do not behave well with trace refinement in either extension. This is essentially due to the fact that the trace refinement does not account for priorities nor preemption. Further work is therefore required to examine the possibility of building a notion of preemption within the trace refinement relation.

To facilitate the use of GCSR, we have developed a tool set for GCSR that allowed
us to evaluate the advantages and limitations of the GCSR language and its refinement theory. The case study, the production cell [LL95], confirmed that the formal treatment of resources and priorities allow a designer to examine the effects of the run-time resources and thus produce more reliable design solutions. In this case study, the integration of resources into the specification allowed us to verify several safety properties in a natural way; for example, to verify that the design does not allow two machines to collide, we verified that the design does not deadlock due to an attempt to use simultaneously the resources that represent the two involved machines.

We made use of the modular and hierarchical syntax of GCSR to structure the GCSR descriptions of the production cell in a manageable fashion. In addition, we used the refinement transformation scheme to produce a design solution in a hierarchical fashion. We used extensively the transformation scheme in the Hoare ordering to introduce design dependencies, i.e., communication events between the various components in the system; to add marker events to test several requirements; and to refine the resource usage to add incrementally the support of safety requirements in the design. Furthermore, an equivalence relation was essential in GCSR, for instance, to introduce a system structure to an abstract specification. The automated translation from GCSR to ACSR specifications was essential to the analysis of the safety and liveness requirements of the production cell. The process algebraic analysis techniques were adequate to verify all of the requirements.

**Future Work**

There are several future research directions that would expand the applicability of the work presented in this thesis. For the GCSR tool set, it is essential to develop a graphical simulator for GCSR and to implement a layout program for a “pretty” layout of GCSR descriptions and for the display of ACSR specifications once translated to GCSR.

For the GCSR language, there are several future research directions. One is to extend its syntax and semantics to support *template* definition along the lines of object-oriented languages. With a template construct in GCSR, a designer can describe one class of systems that differ only in their resource requirements, event names, or timing assumptions. They then can instantiate a class with a specific set of parameters. Such a construct increases specification reusability.

Another fruitful extension to GCSR is to connect it with an automatic code generator. This would bridge another gap in the development process, that between the design and
code levels. Such an extension may require the addition of data variables and value passing. It is however feasible as it can adapt the results of process algebras with value passing, e.g., ACSR-VP [CLX95a].

In addition, there is further work required for the refinement theory. One is to improve the compositionality results in the prioritized semantics. One avenue here is to examine the possibility of encapsulating a notion of preemption into the trace refinement. Another is to modify the semantics of ACSR.

Another future extension for the refinement theory is to support a stronger notion of compositionality. The compositionality results we have proven in this thesis basically show us when it is safe to substitute a refined version of the specification for the specification itself. These results allow us to apply our notion of refinement to several interesting cases. To generalize the applicability of our refinement, it would be fruitful to extend this notion of compositionality along the lines of the stronger notion of compositionality provided in most semantic approaches to action refinement. Since these approaches use semantic equivalence as the soundness of refinement, they ensure the following strong notion of compositionality:

If the implementation is "equivalent" to the specification, then performing equivalent action refinements on both the implementation and specification preserves equivalence.

Within the context of our notion of refinement, this stronger notion of compositionality translates into checking the following two properties:

1. Given a process $P$, a relation $\rho_2 \subseteq \text{actions}(P) \times \text{Proc}$, and a relation $\rho'_2 \subseteq \text{actions}(P) \times \text{Proc}$ such that:
   
   (a) for all $A \in \text{actions}(P)$, there exists $(A, R) \in \rho_2$ iff there exists $(A, R') \in \rho'_2$;

   (b) for all $A \in \text{actions}(P)$, if $(A, R) \in \rho_2$ and $(A, R') \in \rho'_2$, then $R \leq^o R'$;

   (c) $\text{ref}_{\rho_1 \times \rho_2}^\circ(P) \sim P'$; and

   (d) $\text{ref}_{\rho_1 \times \rho_2}^\circ(P) \sim P''$.

   Do we have $P' \leq^o P''$?

2. Given processes $P$ and $Q$ such that $P \leq^o Q$, and given processes $P'$ and $Q'$ such that

Do we have $P' \leq^o Q'$?
(a) \( \text{ref}_{\rho_1 \circ \rho_2}^\circ (P) \sim P' \); and

(b) \( \text{ref}_{\rho_1 \circ \rho_2}^\circ (Q) \sim Q' \).

Do we have \( P' \leq^\circ Q' \)?

Note that in the first property, possible action refinements are restricted to related, refining processes. This makes up for the freedom in our notion of relational action refinement without forcing it to be a function. While this property is a slightly weaker notion of compositionality than the one stated earlier, we conjecture that it holds for all ACSR processes. In the case of the second property, we also conjecture that it is feasible to restrict the relation \( \rho_2 \) to make up for the freedom in our notion of action refinement and prove that the property holds.

As the analysis of the production cell confirmed, it is often natural to express system requirements in terms of predicates in a logic, e.g., temporal linear logic \([MP91]\). Our preliminary research shows that safety properties expressed in the trace model can be preserved by our notions of refinement. The next step is to determine the largest logic preserved by these refinements.

Another fruitful area of extension for refinement in GCSR is to explore the possibility of supporting lower levels of abstraction that incorporate current software/hardware co-design methodologies. This would bridge gaps between hardware and software development and thus increase system reliability.
Appendix A

Selected Proofs for Chapter 3

A.1 Proof of Theorem 3.4.1

We prove that for any GCSR process $G$ we have $T_{GA}(G) \sim G$ by induction on the structure of $G$. The proof inductively constructs a bisimulation relation from the states in the LTS of $G$ to the states in the LTS of $T_{GA}(G)$. In this proof, the ACSR semantic rules refer to those in Appendix D; the node names and translation steps refer to those in Figure 3.7, and $G_N$ denotes the GCSR process that starts at the node $N$ in $G$.

Base case: Case $G$ consists of the NIL node is trivial.

Induction step: Assume the theorem holds for simple structures of GCSR processes and prove it also holds for structurally more complex GCSR processes. Since the LTS of a GCSR process is constructed from the initial node towards reachable nodes, we examine the possible types of initial nodes that $G$ can have. Let the start state of the LTS of $G$ be $s_0 = (q_0, E_0)$.

- Case $\text{initial}(G) = [q_0, \text{instantaneous}, \epsilon]$. The translation step used in this case is Step 3 of Figure 3.7. Using the operational semantics of GCSR (see Appendix D), we can conclude that $G$ has the following set of possible transitions:

$$
\begin{align*}
\text{s}_0 \xrightarrow{\epsilon_i} (q_{Ni}, E_{Ni}) & \text{ for } i = 1, \ldots, j \text{ and } (q_{Ni}, E_{Ni}) = \text{start state of LTS of } G_{Ni} \\
\text{s}_0 \xrightarrow{\alpha_i} (q'_{Mi}, E'_{Mi}) & \text{ for } i = 1, \ldots, k \text{ and } \\
(q_{Mi}, E_{Mi}) \xrightarrow{\alpha_i} (q'_{Mi}, E'_{Mi}) \text{ where } (q_{Mi}, E_{Mi}) = \text{start state of LTS of } G_{Mi}
\end{align*}
$$
Using the operational semantics of ACSR, we can also conclude that \( T_{GA}(G) \) has the same possible transitions as follows:

\[
\begin{align*}
\epsilon_1 T_{GA}(N1) + \cdots + \epsilon_j T_{GA}(Nj) + \epsilon_1 T_{GA}(M1) + \cdots + \epsilon_k T_{GA}(Mk) \\
\xrightarrow{-\epsilon_i} T_{GA}(N1) \text{ for } i = 1, \cdots, j \\
\xrightarrow{-\alpha_i} P_{Mi} \text{ for } T_{GA}(Mi) \xrightarrow{-\alpha_i} P_{Mi} \text{ and } i = 1, \cdots, k
\end{align*}
\]

By induction hypothesis, since \( G_{N1}, \cdots, G_{Mk} \) are structurally simpler than \( G \), then we have \( T_{GA}(G_n) \sim G_n \) for \( n = N1, \cdots, Nj, M1, \cdots, Mk \). Thus, map the start state of the LTS of \( G \) to the start state of the LTS of \( T_{GA}(G) \) and we get \( T_{GA}(G) \sim G \).

- **Case initial\((G) = [n_0, \text{time-consuming}, A] \)**. The translation step used in this case is Step 4. The only possible transition out of the start state of the LTS of \( G \) is through the operational rule \( \text{ParT} \). This rule will be applied for \( t \) time units, where \( t \) is the label of the time-labeled edge from \( n_0 \) to \( N1 \), until the start state of the LTS of \( G_{N1} \) is reached. All of these labeled transitions can be matched out of \( A^t : T_{GA}(N1) \) by the ACSR rule \( \text{ActT} \) with the last step rule \( \text{ActZ} \) to reach \( T_{GA}(N1) \).

The result then follows by induction hypothesis and by mapping all states from the start state of the LTS of \( G \) to the states in the LTS of \( T_{GA}(G) \), in the order they are generated.

- **Case initial\((G) \) is either a reference or compound node**. The translation step applied in this case is Step 5. The case of a reference node follows from the compound node, since a reference node is a short hand notation for a compound node with the referenced GCSR process inside.

Let initial\((G) = [n_0 : \{G_1, \cdots, G_n\}, F, C] \). Let us examine the possible transitions out of the start state of the LTS of \( G \):

<table>
<thead>
<tr>
<th>Rule</th>
<th>Transition</th>
<th>Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbf{ActI}</td>
<td>(1) ( (q_0, E_0) \xrightarrow{f} (q, E) )</td>
<td>( n_0 \in q )</td>
</tr>
<tr>
<td>\textbf{ActI}</td>
<td>(2) ( (q_0, E_0) \xrightarrow{i} (q_{Vi}, E_{Vi}) )</td>
<td>( U_i \in q_{Vi} \wedge (q_{Vi}, E_{Vi}) = \text{start state of LTS of } G_{Vi} ) for ( i = 1, \cdots, l )</td>
</tr>
<tr>
<td>\textbf{ActI}</td>
<td>(3) ( (q_0, E_0) \xrightarrow{i} (q_{Vi}', E_{Vi}') )</td>
<td>( (q_{Vi}', E_{Vi}') = \text{start state of LTS of } G_{Vi} ) for ( i = 1, \cdots, l )</td>
</tr>
<tr>
<td>\textbf{ParCom}</td>
<td>(4) ( A \xrightarrow{A} (q_{N2}, E_{N2}) )</td>
<td>( N2 \in q_{N2} \wedge ((n_0, l, N2), 1) \in E_0 )</td>
</tr>
</tbody>
</table>
The transitions (1) and (4) come from the components inside the compound node. Transition (1) can be repeated unlimited number of times from the state \((q, E)\) to other states. Transition (4) can be repeated from \((q, E)\) to other states at most \(t\) times, after which a transition similar to (5) can occur from the last state reached.

A first important note to make here is that after each application of the transition (1) or (2) we have: 1) \(n_0\) is in the reached configuration, and 2) by the definition of the function \(next\), the relevant edges in the reached state will contain the edges out of \(n_0\), updated to reflect time passage. In other words, after each repeated application of transition (1) or (5), we reach a state from which transitions (1) through (6) remain possible. Call the repeated application of these transitions as “derivative” transitions.

A second important note is that after taking any one of the transitions (2), (3), (4), or (6) the transitions out of the reached state come from the GCSR process starting, respectively, at the nodes \(U_i\) (for \(i = 1, \ldots, l\)), nodes \(V_j\) (for \(j = 1, \ldots, n\)), node \(N2\), or node \(N1\).

It is now easy to check that the above transitions can be matched in the process \(T_{GA}(G)\) using the ACSR operational rules \textbf{ScopeCI} for transition (1) and its derivatives; \textbf{ScopeIl} for transitions (2) and (3); \textbf{ScopeT} for transition (4) and its derivatives; \textbf{ScopeCT} for transition (5); and \textbf{ScopeE} for transition (6).

The result then follows by induction hypothesis and by mapping the start state of the LTS of \(G\) to the start state of the LTS of \(T_{GA}(G)\) and any derivative state through repeated application of transitions (1) and (5) to the corresponding derivative state. □

A.2 Proof of Theorem 3.4.5

We prove that for every GCSR process \(G\), there exists a GCSR process \(G' = T^{\ast}_{GG}(G)\) such that \(T_{AG}(T_{GA}(G)) \equiv G'\).
If $G$ is a basic GCSR process, then $T_{AG}(T_{GA}(G)) \equiv G$. If $G$ is not a basic GCSR process, we can apply the following ordered, transformation steps on $G$ to reach a basic GCSR process $G'$ such that $G' \equiv T_{AG}(T_{GA}(G))$.

1. Unfold the loops in $G$ by repeatedly applying transformation 1 in Figure 3.10. This step may result in defining new GCSR processes on the side, when the source node of the loop is not an initial node.

2. For each node $n$ with more than one incoming edge, use transformation 5 in Figure 3.12 to duplicate the GCSR component which starts at $n$. Note that to apply transformation 5, the component connected to $n$ must be defined. This is the case since loops have been unfolded in step 1.

The result of steps 1 and 2 is a GCSR process where each node has at most one incoming edge.

3. Use transformation 3 to eliminate unnecessary unlabeled edges and instantaneous nodes. This step is required in case $G$ contains unnecessary unlabeled edges and instantaneous nodes, other than what the translation produced.

4. Use transformation 3 again this time to add unlabeled edges out of instantaneous nodes so that each instantaneous node either has two unlabeled out-edges or has one event-labeled out-edge. This step can be applied since each node in the transformed $G$ now has at most one incoming edge.

5. For each compound node, repeatedly apply transformations 6, 7a, and 7b until each compound node $n$ is of the form $[n : (\{G\}, F, \emptyset)]$, $[n : (\{G\}, \emptyset, C)]$, or $[n : (\{G_1, G_2\}, \emptyset, \emptyset)]$.

6. For each compound node with out-edges, 1) apply transformation 6 so that it contains one nested GCSR process only, 2) apply transformations 2a and 2b to add a time-labeled edge and an exception edge if necessary, and 3) apply transformation 3 to add an instantaneous node that groups the other normal edges out of the compound node.

After applying the above transformation steps, it is easy to verify that the resulting GCSR process satisfies the properties of a basic GCSR process. \(\square\)
Appendix B

Selected Proofs for Chapter 4

B.1 Proof of Theorem 4.3.1

We prove that trace-set refinement in the Hoare ordering is compositional with the ACSR operators, except for the Parallel operator.

Assume that $P \leq^1_{\rho} Q$, and let the set of local events in $P$ under $\rho$ be $Local = Local(P, \rho)$.

(1a). We want to show that for every action $A^a \in Act$

$$A^a : P \leq^1_{\rho} A^a : Q$$

By hypothesis, we have for each $r \in traces(P)$ there is some $s \in traces(Q)$ such that $r \leq_{\rho} s$. Then, for each action $A^a$, $A^a r \leq_{\rho} A^a s$ using condition (3) of the definition of $\leq_{\rho}$ (Definition 4.2.2); the result follows.

(1b). We want to prove that for every event $e \in Evts$ such that $P$ and $Q$ are compatible with $e.NIL$ under $\rho$ we have

$$e.P \leq^1_{\rho \cup \{r, e\}} e.Q$$

Since $e \notin Local$, then using using condition (3) of the definition of $\leq_{\rho}$ (Definition 4.2.2) we have $er \leq_{\rho} es$ for all $r \in traces(Q)$ and $s \in traces(P)$ such that $r \leq_{\rho} s$.

(2a). We want to prove that for every process $E$ such that $P$ and $Q$ are compatible with $E$ under $\rho$ we have

$$(P + E) \leq^1_{\rho \cup Id_E} (Q + E)$$

By operational semantics of Choice, note that each $r \in traces(P+E)$ is either in $traces(P)$, $traces(E)$, or both. If $r \in traces(P)$ then, by induction hypothesis, there must be some
\[ s \in \text{traces}(Q) \text{ such that } r \leq_{\rho} s. \text{ Since } \text{events}(E) \cap \text{Local} = \emptyset, \text{ then } r \leq_{\rho \cup \text{id}_E} s. \text{ Also, if } r \in \text{traces}(E) \text{ then } r \leq_{\text{id}_E} r; \text{ hence } r \leq_{\rho \cup \text{id}_E} r. \]

(2b). The proof is the same as proof (2a).

(4). We want to prove that for every set of events \( F \subseteq \text{events}(Q) \), we have

\[ (P \setminus \rho(F)) \leq_{\rho} (Q \setminus F) \text{ where } \rho(F) = \{ \rho(e) : e \in F \}. \]

The assumption \( F \subseteq \text{events}(Q) \) is the simplest assumption to ensure that no local event of \( P \) is restricted inadvertently; that is, \( \rho(F) \cap \text{Local} = \emptyset \). This assumption makes use of the equivalence of \( P \setminus F \sim P \) when \( F \not\subseteq \text{events}(P) \) for any process \( P \). In general, without this assumption, we need to first augment \( \rho \) with the identity for the events in \( (F - \text{events}(Q)) \), and second, rename the local events of \( P \) so that they are distinct from the image of the events in \( F \). This method of renaming has been adopted in e.g. [Ace92] where \( \alpha \)-conversion is used.

In our proof, we will use the function \( \text{filter}(s, F) \) which reflects the restrict operator on traces:

\[ \text{filter}(s, F) = \begin{cases} 
\epsilon & \text{if } r = \epsilon \\
A^s \text{filter}(s', F) & \text{if } s = A^s s' \\
\text{filter}(s', F) & \text{if } s = es'^{\perp} \wedge (e \not\in F \wedge \tau \not\in F) \\
\epsilon & \text{if } s = es'^{\perp} \wedge (e \in F \vee \tau \in F)
\end{cases} \]

We now prove that given a trace \( r \in \text{traces}(P \setminus \rho(F)) \), there exists a trace \( s \in \text{traces}(Q \setminus F) \) such that \( r \leq_{\rho} s \).

\( r \in \text{traces}(P \setminus \rho(F)) \) implies (by operational semantics of \text{Restrict}) that there exists \( r' \in \text{traces}(P) \) such that \( r = \text{filter}(r', \rho(F)) \). Since \( P \leq_{\rho} Q \), then there exists \( s' \in \text{traces}(Q) \) such that \( r' \leq_{\rho} s' \). We prove that \( s = \text{filter}(s', F) \) gives \( r \leq_{\rho} s \). Note that the resource conditions of the trace refinement definition (conditions (4)-(6) of definition 4.2.2) are trivially satisfied because the restrict operation does not affect actions. The interesting part of the proof is checking that \( \text{dur}(r) = \text{dur}(s) \) and that all events of \( s \) are matched to events in \( r \).

\textbf{Case 1:} \( \text{dur}(r) = \text{dur}(r') \); that is \( r' \) was not truncated due to unsynchronized events in \( \rho(F) \). This can be due to a combination of the following two cases:

- \( r' \) contains no events in \( \rho(F) \), i.e. \( r' = r \). Since \( r' \leq_{\rho} s' \) by Property 4 we can infer that \( \text{events}(s') \cap F = \emptyset \); therefore \( s = \text{filter}(s', F) = s' \). Thus \( r \leq_{\rho} s \).
The last sequence of events in $r'$ was truncated. To simplify the readability of the proof, assume that there are no local events. This is a safe assumption, since by restricting $F \subseteq \text{events}(Q)$, we have $\rho(F) \cap \text{Local} = \emptyset$; thus no local event will be caught in the restriction. Now, let the last sequence of events in $r'$ be $e'_1, ..., e'_k$, where $e'_1$ is the first event that is in $\rho(F)$. Therefore, the last sequence of events in $r$ will be $e'_1, ..., e'_{k-1}$. Since $r' \leq_{\rho} s'$, by Property 4 we can infer that there is an event $e_k$ in the last sequence of events of $s'$ such that $e'_k \in \rho(e_k) \cap F$, and that the last sequence of events in $s = \text{filter}(s', F)$ is $e_1, ..., e_k$ with $e_k$ being the first event in the sequence and which is in $F$. Thus $s$ will also be truncated at $e_k$; that is, the last sequence of events in $s$ will be $e_1, ..., e_{k-1}$. Therefore $r \leq_{\rho} s$.

**Case 2:** $\text{dur}(r) \leq \text{dur}(r')$, that is synchronization was not successful and truncation occurred. This is similar to the second part of Case 1.

(5). We want to prove that for any set of resources $I$ we have

$$[P]_I \leq_{\rho} [Q]_I$$

First, note that the closure operation does not affect events or duration of traces; it only augments the actions in the traces so that they include the resources in the closing set $I$.

Note also that by operational semantics of Closure, for each $r \in \text{traces}(P[I])$ there exists $r' \in \text{traces}(P)$ such that, $r$ is obtained from $r'$ by augmenting all actions in $r'$ with the resources in $I$, noted as $r = [r']_I$.

Now, since $P \leq_{\rho} Q$, then there exists $s' \in \text{traces}(Q)$ such that $r' \leq_{\rho} s'$. We claim that the trace $s = [s']_I$ in $\text{traces}(Q[I])$ satisfies $r \leq_{\rho} s$. The interesting parts in proving $r \leq_{\rho} s$ are conditions (4)-(6) of the trace refinement definition (Definition 4.2.2). We show the case of condition (4); the others can be constructed in a similar way.

The proof is by contradiction: Assume $r = A^u r_1$ and $s = B^u s_1$ and $r \not\leq_{\rho} s$ because $A \not\subseteq B$. However, we have $r' = A'^u r'_1$ and $s' = B'^u s'_1$ are related; thus, $A' \subseteq B'$. From which we can infer that $[A']_I \subseteq [B']_I$. Since $A = [A']_I$ and $B = [B']_I$, this implies that $A \subseteq B$. A contradiction; thus $A \subseteq B$ and $r \leq_{\rho} s$.

(6). To prove the compositionality of trace-set refinement with respect to the scope operator in all its arguments, we note that any trace in a scope process is composed of at most two argument processes. In the next proofs, we focus on the case where the processes $P$ and $Q$ are involved. The other traces are matched through the identity relation over traces
which is a trace refinement. Let \( P \) and \( Q \) be compatible with processes \( E_i \) for \( i = 1, \ldots, 4 \) where \( E_4 = e, \text{NIL} \). Let the function \( \rho' = \rho \cup I d_{E_1 \ldots E_4} \).

(6a). We want to prove that if \( \rho'(\tau) = \overline{\rho'(e)} \) and \( \rho'/(L \cup \overline{L}) \) is one-to-one, complement preserving where \( L = \text{events}(Q) \cup \{e\} \), then we have

\[
P \triangle^e_{\rho}(E_1, E_2, E_3) \leq_{\rho'} Q \triangle^e_{\rho}(E_1, E_2, E_3)
\]

We focus on the traces that involve \( P \). In this case, a trace \( r = r's \) where \( r' \in \text{traces}(P) \), \( r' \) is finite, \( \text{dur}(r') \leq u \), and \( s \) is either in \( \text{traces}(E_1) \), \( \text{traces}(E_2) \), or \( \text{traces}(E_3) \). Since \( P \leq_{\rho} Q \), then there exists \( s' \in \text{traces}(Q) \) such that \( r' \leq_{\rho} s' \). By the compatibility condition, we can infer that \( r' \leq_{\rho'} s' \). By compositionality of \( \leq_{\rho'} \) with concatenation, this implies that \( r's \leq_{\rho'} s' \) for all \( s \in (\text{traces}(E_1) \cup \text{traces}(E_2) \cup \text{traces}(E_3)) \). The result follows by noting that \( s' \) is a trace of \( Q \triangle^e_{\rho}(E_1, E_2, E_3) \).

An additional detail must be noted when \( s \in \text{traces}(E_1) \). In this case, there exists \( s'' = r'\rho'(e) \in \text{traces}(P) \) with \( \rho'(e) \notin \text{events}(r') \). Since \( P \leq_{\rho} Q \), we can infer that there exists \( s'' \in \text{traces}(Q) \) and \( r'' \leq_{\rho} s'' \). Also, by compatibility of \( P \) and \( Q \) with \( e, \text{NIL} \), and by Property 4, we can infer that \( s'' = s'r \) with \( r \notin \text{events}(s') \). Thus, \( s' \) is a trace of \( Q \triangle^e_{\rho}(E_1, E_2, E_3) \).

(6b). We want to prove that

\[
E_1 \triangle^e_{\rho}(P, E_2, E_3) \leq_{\rho'} E_1 \triangle^e_{\rho}(Q, E_2, E_3)
\]

We focus on traces \( r \) of \( E_1 \triangle^e_{\rho}(P, E_2, E_3) \) that involve \( P \). In this case, \( r = ss' \) where \( s \in \text{traces}(E_1) \), \( s \) is finite, \( \text{dur}(s) < u \), and \( r' \in \text{traces}(P) \). Since \( P \leq_{\rho} Q \), then there exists \( s' \in \text{traces}(Q) \) such that \( r' \leq_{\rho} s' \). By definition of \( \rho' \), this implies that \( sr' \leq_{\rho'} ss' \) for all finite traces \( s \). The result follows by noting that \( ss' \) is a trace of \( E_1 \triangle^e_{\rho}(Q, E_2, E_3) \).

(6c). We want to prove that

\[
E_2 \triangle^e_{\rho}(E_1, P, E_3) \leq_{\rho'} E_2 \triangle^e_{\rho}(E_1, Q, E_3)
\]

We focus on the traces \( r \) of \( E_2 \triangle^e_{\rho}(E_1, P, E_3) \) that are defined over the actions of \( E_2 \) and \( P \). In this case, \( r = ss' \) where \( s \) is either: 1) \( e \), or 2) it ends with an action, \( \text{dur}(s) = u \), and \( r' \in \text{traces}(P) \). Since \( P \leq_{\rho} Q \), then there exists \( s' \in \text{traces}(Q) \) such that \( r' \leq_{\rho} s' \). By definition of \( \rho' \), this implies that \( sr' \leq_{\rho'} ss' \) for all finite \( s \) such that \( r = sr' \). The result follows by noting that \( ss' \) is a trace of \( E_2 \triangle^e_{\rho}(E_1, Q, E_3) \).
We want to prove that
\[ E_3 \triangle^*_u (E_1, E_2, P) \preceq_{\rho'} E_3 \triangle^*_u (E_1, E_2, Q) \]

We focus on the traces where \( P \) is involved. That is, traces \( r = sr' \) where \( s \in \text{traces}(E_3) \), \( s \) is finite, \( \text{dur}(s) \leq u \), and \( r' \in \text{traces}(P) \). Since \( P \preceq_{\rho} Q \), then there exists \( s' \in \text{traces}(Q) \) such that \( r' \preceq_{\rho} s' \). By definition of \( \rho \), this implies that for all finite \( s \in \text{traces}(E_3) \), \( sr' \preceq_{\rho'} ss' \). The result follows by noting that \( ss' \) is a trace of \( E_3 \triangle^*_u (E_1, E_2, Q) \).

We conjecture that, trace-set refinement is compositional with recursion for all ACSR processes. However, in constructing the proof, we restrict ourselves to finite-state processes, and assume \( FV(P) = FV(Q) \). In this case, we make use of the following lemma.

**Lemma B.1.1** For a finite-state process, \( P \), an infinite trace is in \( \text{traces}(P) \) whenever all its finite prefixes are.

To prove the above lemma, it suffices to note that since \( P \) generates a finite-state labeled transition system, then any infinite trace of \( P \) comes from a loop in the labeled transition system. The set of traces that satisfy the above property are called *limit-closed traces* in [IV91].

Note that by operational semantics, a trace in \( \text{traces}(\text{rec } X.P) \) is constructed from unfolding the recursion possibly infinitely many times. For finite-state processes, the states where recursion is unfolded correspond to the states where a loop in the process’s transition system is iterated.

In the presence of free variables, we need to augment our notion of trace and trace refinement to account for variables. From the normal form for finite-state ACSR processes that is developed in [BGCL93], we can conclude that, in the presence of free variables, a finite trace can end with \( [X \setminus F]_U \) where \( X \) is a process variable, \( F \) a set of event names and \( U \) a set of resource names. In addition, the definition of trace refinement \( r \preceq_{\rho} s \) (Definition 4.2.2) must be extended with the possibility
\[ r = [X \setminus E]_U \land s = [X \setminus F]_V \land \rho(F) \subseteq E \land U = V \]

Based on the above remarks, we next prove the theorem by induction on the number of times the recursion is unfolded.
The base step is when a trace $r \in traces(rec\ X.P)$ is obtained by zero unfolding. In this case, $r \in traces(P)$. Therefore by the theorem hypothesis ($P \mid\rho\mid Q$), there exists $s \in traces(Q)$ such that $r \leq_{\rho} s$. The result follows since $s \in traces(rec\ X.Q)$.

In the inductive step, we assume that the theorem holds for all traces obtained by at most $n$ recursion unfoldings, and prove it also holds for traces obtained by $(n + 1)$ recursion unfoldings. Let $r = r_1r_2 \in traces(rec\ X.P)$ such that $r_2$ is obtained by $n$ recursion unfoldings, and $r_1$ is obtained by one more unfolding. By induction hypothesis for $r_2$, we can infer that these exists $s_2 \in traces(rec X.Q)$ such that $r_2 \leq_{\rho} s_2$. We need to prove that there is $s_1$ such that $s_1s_2 \in traces(rec X.Q)$ and $r_1 \leq_{\rho} s_1$; the final result that $r_1r_2 \leq_{\rho} s_1s_2$ follows from compositionality of $\leq_{\rho}$ with respect to concatenation.

By operational semantics of recursion, $r_1$ is finite and must lead to a “state” of $P$ where the recursion can be unfolded to start producing $r_2$; that is $r_1$ ends with $[X\setminus E]_V$. By the theorem hypothesis ($P \leq_{\rho} Q$), we can infer that there exists a trace $s_1 \in traces(Q)$ such that $r_1 \leq_{\rho} s_1$. By the definition of trace refinement, this implies that $s_1$ is finite and ends with $[X\setminus F]_V$ where $\rho(F) \subseteq E$ and $U = V$. Thus, we can unfold the recursion in $s_1$ to produce more traces of $rec X.Q$. We claim that there is a trace $s_1$ such that $s_1s_2 \in traces(rec X.Q)$ and $r_1r_2 \leq_{\rho} s_1s_2$.

Assume for a contradiction that for all $s_1 \in traces(Q)$ such that $s_1s_2 \in traces(rec X.Q)$ and $r_1 \leq_{\rho} s_1$, but $r_1r_2 \not\leq_{\rho} s_1s_2$. By the definition of $\leq_{\rho}$ and since $r_2 \leq_{\rho} s_2$, we infer that this is the case if $r_1 \not\leq_{\rho} s_1$, which contradicts the hypothesis $P \leq_{\rho} Q$. Thus, there exists $s_1 \in traces(Q)$ such that $s_1s_2 \in traces(rec X.Q)$ and $r_1r_2 \not\leq_{\rho} s_1s_2$.  

\[\square\]

**B.2 Proof of Lemma 4.3.1**

We prove that trace-set refinement in the Hoare ordering is compositional with the Parallel operator under some restrictions. More specifically, we prove that, if $P \leq_{\rho} Q$ then for any process $E$ if $(P \cup Id_E) \setminus (L \cup \overline{L})$ where $L = events(Q) \cup events(E)$ is a one-to-one, complement-preserving function, and if $res(Q) \cap res(E) = \emptyset$, then we have

\[(P\|E) \leq_{\rho \cup Id_E} (Q\|E).\]

The intuition behind the resource restriction is the following. It ensures that $Q\|E$ will not deadlock due to a resource conflict. By the hypothesis that $P \leq_{\rho} Q$, then at any time the resources used by $P$ is a subset of the resources used by $Q$. We can then infer that $P\|E$
will not deadlock due to a resource conflict either.

Let \( r \in traces(P||E) \). Then by operational semantics, there exist \( r_1 \in traces(P) \) and \( r_2 \in traces(E) \), such that \( r \) is obtained by the parallel composition of \( r_1 \) and \( r_2 \). Since \( P \preceq Q \), then there exists \( s_1 \in traces(Q) \) such that \( r_1 \leq \rho s_1 \). Since \( Local \cap events(E) = \emptyset \), then we infer that \( r_1 \leq \rho \cup Id_E s_1 \). Also, we have \( r_2 \leq Id_E r_2 \) and hence \( r_2 \leq \rho \cup Id_E r_2 \). We claim that there exists a trace \( s \in traces(Q||E) \) obtained by the parallel composition of \( s_1 \) and \( r_2 \), and that \( r \leq \rho \cup Id_E s \).

Let us first review informally the parallel composition of two traces \( r_1 \) and \( r_2 \) \( (r_1 || r_2) \).

By operational semantics, \( r_1 || r_2 \) is a set of traces. Each trace in this set has the following three properties: 1) its duration is the minimum of the durations of \( r_1 \) and \( r_2 \)—since there is no resource contention; 2) its actions are composed by the union of the actions of \( r_1 \) and \( r_2 \) that happen at the same time; and 3) at any instant of time \( t \leq \text{dur}(r_1 || r_2) \), the sequence of events of \( r \in r_1 || r_2 \) at \( t \) is a shuffle or a “synchronized” shuffle of the sequences of events of \( r_1 \) and \( r_2 \) at \( t \). Informally, a synchronized shuffle is obtained from a (regular) shuffle by, starting from the left, deleting any two consecutive events that are complements; this reflects the effect of synchronization between events. Note that synchronization events are those events shared with \( events(E) \).

The first and second properties ensure condition (1) and resource inclusion condition of trace refinement for any \( s \in s_1 || r_2 \). We next show that there is an \( s \in s_1 || r_2 \) whose events can be corresponded through \( \rho \cup Id_E \) to events of \( r \) according to the definition of trace refinement. This is done by induction on the instant the events occur \( 0 \leq t \leq \text{dur}(r) \). We next just focus on an intermediate step of the proof where for a fixed \( t \) we check that we choose \( s \) such that \( event(s,t) \) can be matched with \( event(r,t) \).

Let \( 0 \leq t \leq \text{dur}(r) \). If \( event(r,t) \) is a shuffle of \( event(r_1,t) \) and \( event(r_2,t) \), then \( event(s_1,t)||event(r_1,t) \) also has a trace which is composed of a shuffle of \( event(s_1,t) \) and \( event(r_1,t) \). Choose a trace that has the same order of events as the corresponding (through \( \rho \cup Id_E \)) events in \( event(r,t) \).

If \( event(r,t) \) is a synchronized shuffle of \( event(r_1,t) \) and \( event(r_2,t) \), then let \( e \) be the event that \( event(r_1,t) \) and \( event(r_2,t) \) synchronized on; that is \( e \not\in event(r,t) \) and \( e \in event(r_1,t) \) and \( \overline{e} \in event(r_2,t) \). By assumption, we have \( Local \cap events(E) = \emptyset \). Therefore, \( e \not\in Local \). We can then infer by Property 4 that \( e \not\subseteq event(s_1,t) \). Thus, there is a trace \( s \subseteq event(s_1,t)||event(r_1,t) \) where \( e \not\subseteq event(s) \). Again, choose a trace that has the same order of events as the corresponding (through \( \rho \cup Id_E \)) events in \( event(r,t) \). □
B.3 Proof of Theorem 4.3.2

Soundness of the transformation rules of the Hoare ordering—We want to prove that

\[ \text{if } \text{ref}_{\{\rho_1, \rho_2\}}(Q) \rightsquigarrow P \text{ then } P \preceq_{\rho_1} Q . \]

The proof is by induction on the structure of \( Q \). The base step is when \( Q \) is NIL. In this case, there are two possible rules to apply (1) and (2) of Definition 4.3.5. Suppose rule (1) were applied; then we have trivially \( \text{NIL} \preceq_{\rho_1} \text{NIL} \). Suppose rule (2) were applied; then for any local event \( l \), we have \( l, \text{NIL} \preceq_{\rho_1} \text{NIL} \) by condition (2) of definition of \( \preceq_{\rho_1} \) (Definition 4.2.2).

In the inductive step, assume we have \( P \preceq_{\rho_1} Q \) with \( \text{ref}_{\{\rho_1, \rho_2\}}(Q) \rightsquigarrow P \), and prove the theorem holds for more complex processes.

- **Case the last step used is (2).** By induction hypothesis, for each trace \( r \in \text{traces}(P) \), there exists \( s \in \text{traces}(Q) \), such that \( r \preceq_{\rho_1} s \). Using condition (2) of definition of \( \preceq_{\rho_1} \) (Definition 4.2.2), we can infer that \( lr \preceq_{\rho_1} s \) for any local event \( l \). By operational semantics, since \( r \in \text{traces}(P) \), we have \( lr \in \text{traces}(l.P) \). Also, since

\[
\text{traces}(\bigcup_{l \in L} l.P) = \bigcup_{l \in L} \text{traces}(l.P)
\]

for a finite set of local events \( L \), we can infer that

\[
\bigcup_{l \in L} l.P \preceq_{\rho_1} Q
\]

(and we are done.)

- **Case the last step used is (4).** First note that \( P' \subseteq \text{res}(A) \). By definition of \( \rho_2 \) we have

\[
\forall R \in \rho_2(A^n) : (R \preceq_{\rho_1} A^n: \text{NIL}) ,
\]

from which we can infer (by operational semantics of the Close operator) that for any \( P' \subseteq \text{res}(A) \) we have

\[
[R]_{P'} \preceq_{\rho_1} A^n : \text{NIL} .
\]

Using the induction hypothesis and the fact that trace-set refinement is compositional with the scope operator, we can conclude that for any \( R \in \rho_2(A^n) \) and any \( P' \subseteq \text{res}(A) \), we have

\[
[R]_{P'} \triangle_a (\text{NIL}, P, \text{NIL}) \preceq_{\rho_1} A^n : \text{NIL} \triangle_a (\text{NIL}, Q, \text{NIL}) .
\]
Since
\[
\text{traces}(A^n : Q) = \text{traces}(A^n : \text{NIL} \triangle_u (\text{NIL}, Q, \text{NIL}))
\]
we can infer that
\[
[R]_{I'} \triangle_u (\text{NIL}, P, \text{NIL}) \leq_{\rho_1} A^n : Q
\]
for any \( R \in \rho_2(A^n) \) and any \( I' \subseteq \text{res}(A) \).

- **Case the last step used is (5).** The result follows by the operational semantics of the Close operator, induction hypothesis and compositionality of trace refinement with the parallel operator in the Hoare ordering (Lemma 4.3.1).

- **Case the last step used is (8).** By operational semantics of the Restrict operator, we have for every set of events \( F \),
\[
\text{traces}(\text{ref}_{(\rho_1, \rho_2)}^1(Q) \setminus F) \subseteq \text{traces}(\text{ref}_{(\rho_1, \rho_2)}^i(Q))
\]
the results is then immediate since \( \text{ref}_{(\rho_1, \rho_2)}^1(Q) \leq_{\rho_1} Q \)

- **Case the last step used is (3), (6), (7), (9), or (10).** The result follows by using induction hypothesis and the fact that trace-set refinement is compositional with respect to the event-prefix, Choice, Scope, Restrict, and Close operators respectively as proven in Theorem 4.3.1.

- **Case the last step used is (11).** The result follows by using induction hypothesis and the facts that: 1) \( \leq_{\rho_1} \) is compositional with respect to recursion, and 2) the transformation rules do not allow renaming of variables (rule number 12).

\[\square\]

### B.4 Proof of Theorem 4.3.3

**Completeness of the transformation rules of the Hoare ordering extension**— We want to prove that for all finite processes \( P \) and \( Q \), if \( P \leq_{\rho_1}^1 Q \) for some one-to-one, complement preserving function \( \rho_1 : \text{events}(Q) \rightarrow \text{Ents} \), then there exist processes \( Q' \sim Q \) and \( P' \sim P \) and a relation \( \rho_2 \subseteq \text{actions}(Q') \times \text{Proc} \) such that \( \text{ref}_{(\rho_1, \rho_2)}^1(Q') \sim P' \).

We use induction on the structure of the process \( Q \). In each step, we fix the structure of the process \( Q \), and construct \( \rho_2 \) and a process \( P' \sim P \) such that \( P' \) is obtained through the transformation rules either from \( Q \) or from a strongly bisimilar process \( Q' \sim Q \). The construction of \( \rho_2 \) and \( P' \) is guided by the hypothesis \( P \leq_{\rho_1}^1 Q \).

The base step is when \( Q = \text{NIL} \). Since \( \text{actions}(Q) = \emptyset \), then \( \rho_2 \) in this case is the null relation. By hypothesis, we have \( P \leq_{\rho_1}^1 \text{NIL} \); therefore, we can infer that \( P \) can be either
bisimilarity to the NIL process, or to a process that executes local events only. If $P \sim \text{NIL}$, then transformation rule (1) can be used to transform $Q$ to NIL and we are done. If $P$ is bisimilar to a process that executes local events only, then first note that since $P$ is finite its set of traces is finite and consists of finite traces. Then, by the completeness of the axioms for strong bisimulation, we can conclude that we can rewrite $P$ to an equivalent process

$$P' \overset{\text{def}}{=} \sum_{i \in \text{traces}(P)} i.P^i$$

where the processes $P^i$ are constructed as the sum of event-prefix processes that end with the NIL process. Thus, we can rewrite $Q$ to $P'$ using rule (2), i.e., to introduce new events and the last rewrite being rule (1), i.e., $\text{ref}_{\{p_1,p_2\}}^1(\text{NIL}) \sim \text{NIL}$.

In the inductive step, we assume that the theorem holds for processes that are structurally “simple” and prove that the theorem also holds for a process $Q$ that is constructed from simpler processes.

- **Case $Q = \epsilon.Q'$.** By hypothesis we have $P \leq_{p_1} Q$. We can infer (by the definition of trace refinement) that we have one of the following three cases:

  1. $\text{traces}(P) = \{\epsilon\}$. In which case we use the refinement rule (1), $\text{ref}_{\{p_1,p_2\}}^1(Q) \sim \text{NIL}$,
     with $\rho_2 \subseteq \text{actions}(Q) \times \{\text{NIL}\}$. In this case, the completeness of the axioms of strong bisimulation guarantees that we can rewrite $P$ to NIL;

  2. $\text{traces}(P)$ are composed of local events only. In this case, we use: $\rho_2 \subseteq \text{actions}(Q) \times \{\text{NIL}\}$; and the same reasoning as in the base step to construct $P' \sim P$.

  3. each trace $r \in \text{traces}(P)$ is either:
     1) $r = \alpha$, where $\alpha$ is a finite sequence of local events, or
     2) $r = \beta \rho_1(\epsilon)r'$ where $\beta$ is a finite sequence of local events and $r' \in \text{traces}(P_2)$ for a process $P_2$ such that $P_2 \leq_{p_1} Q'$. Figure B.1 illustrates the possible trace transitions of $P$. Now, since $Q'$ is structurally simpler than $Q$, then by induction hypothesis there is $\rho_2 \subseteq \text{actions}(Q') \times \text{Proc}$ such that $\text{ref}_{\rho_1 \cup \rho_2}^1(Q') \sim P_2$ (modulo strong bisimulation). Since $\text{actions}(Q) = \text{actions}(Q')$, then we can use $\rho_2$ also to rewrite $Q$ to $P$ (modulo strong bisimulation) as follows: 1) use strong bisimulation to rewrite $\epsilon.Q' \sim \epsilon.Q' + \text{NIL}$; 2) use the transformation rule (6) to rewrite the right hand side as follows:

$$\text{ref}_{\{p_1,p_2\}}^1(\epsilon.Q' + \text{NIL}) \sim \text{ref}_{\{p_1,p_2\}}^1(\epsilon.Q') + \text{ref}_{\{p_1,p_2\}}^1(\text{NIL})$$
3) use transformation rule (2) on ref$_{(\rho_1,\rho_2)}^\prime(e,Q')$ to add the local events in the sequences $\beta_0, \ldots, \beta_n$ while preserving the branching of $P$—this step uses the completeness of the axioms of strong bisimulation; 4) use transformation rule (2) on ref$_{(\rho_1,\rho_2)}^\prime$(NIL) to add the local events in the sequences $\alpha_0, \ldots, \alpha_k$ while preserving the branching of $P$—this step uses the completeness of the axioms of strong bisimulation; 5) use transformation rule (3) ref$_{(\rho_1,\rho_2)}^\prime(e,Q') \sim \rho(e).P_2$ for the first sum, and transformation rule (1) ref$_{(\rho_1,\rho_2)}^\prime$(NIL) $\sim$ NIL for the second sum; and we are done. That is, the resulting process is $P' \sim P$.

- Case $Q = A^u : Q'$. By hypothesis we have $P \leq^\prime_{\rho_1} Q$. We can infer (by the definition of trace refinement and trace semantics) that the set $\text{traces}(P)$ is composed as follows:

$$
\text{traces}(P) = \{rr' \mid P \xrightarrow{r}^* P_1 \xrightarrow{r'}^* \land 
\begin{align*}
& r \leq^\prime_{\rho_1} A^u \land \text{events}(r) \cap \text{range}(\rho_1) = \emptyset \land \\
& (\text{dur}(r) \leq u \land r' = e) \lor (\text{dur}(r) = u \land P_1 \leq^\prime_{\rho_1} Q')
\end{align*}
$$

Since $P$ is finite the set of processes $P_1$ as defined above is finite modulo strong bisimulation. Thus, let the process $R$ be obtained from $P$ by taking all the transitions leading to a process $P_1$, as defined in the set $\text{traces}(P)$ above, and replace the process $P_1$ by the NIL process. It is essential to note that $R$ preserves the branching in $P$ up to processes $P_1$. Note also that we have

$$
R \leq^\prime_{\rho_1} A^u : \text{NIL}
$$

and that the Condition (4.3) is satisfied. Also, let the process

$$
P_2 \overset{\text{def}}{=} \sum P_1
$$

where the processes $P_1$ are as defined in $\text{traces}(P)$. It is straightforward to see that
$P_2 \preceq_{\rho_1} Q'$. By induction hypothesis, we can find $P_2' \sim P_2$ such that

$$\text{ref}_{(\rho_1, \rho_2)}(Q') \sim P_2'.$$

Now, let $P' \overset{\text{def}}{=} R \triangle_n (\text{NIL}, P_2', \text{NIL})$. It is obvious that $P' \sim P$. Furthermore, let $\rho_2 = \{(A^n, R)\} \cup \rho_2'$; thus we can use the transformation rule (4) to rewrite $Q$ to $P'$.

- Case $Q = Q_1 \triangle_4 (Q_2, Q_3, Q_4)$. By hypothesis we have $P \preceq_{\rho_1} Q$. We can infer (by the definition of trace refinement) that the set $\text{traces}(P)$ is composed as follows:

$$\text{traces}(P) = \{rr' \mid P \xrightarrow{r} P_1 \xrightarrow{r'} \land \rho_1(\overline{r}) \not\in \text{events}(r) \land \exists s_1 \in \text{traces}(Q_1), r \preceq_{\rho_1} s_1 \land (\text{dur}(r) < t \land \exists s_2 \in \text{traces}(Q_2), r' \preceq_{\rho_1} s_2) \lor (\text{dur}(r) = t \land \text{event}(r, t) = \varepsilon \land \exists s_3 \in \text{traces}(Q_3), r' \preceq_{\rho_1} s_3) \lor (\text{dur}(r) < t \land \exists s_4 \in \text{traces}(Q_4), r' \preceq_{\rho_1} s_4) \}.$$

Since $P$ is finite the sets of processes $P_2, P_3$ and $P_4$ as defined above are finite—modulo strong bisimulation. Now, use these processes to define the following processes

$$P_2' \overset{\text{def}}{=} \sum P_2, \text{ and } P_4' \overset{\text{def}}{=} \sum P_4.$$

It is straightforward to prove that $P_i' \preceq_{\rho_i} Q_i$ for $i = 2, 3, 4$. Thus, by the induction hypothesis, there exist relations $\rho_2; i$ such that $\text{ref}_{(\rho_1, \rho_2, i)}(Q_i) \sim P_i'$ for $i = 2, 3, 4$—modulo strong bisimulation.

In addition, we can define the process $P_1$ as the sum of the prefix-processes obtained from the transitions on the traces $r_1$ as defined in the set $\text{traces}(P)$ with the last action leads to the process NIL and if the second subset of $\text{traces}(P)$ is not empty, we add $\rho_1(\overline{r})$ after the last action in $r_1$. Such a process exists due to the completeness of the strong
bismulation axioms and the assumption that $P$ is finite. Furthermore, such a process $P_1$ will preserve the branching in $P$ and will satisfy $P_1 \leq_{\rho_1} Q_1$. Thus, by induction hypothesis, there exists a relation $\rho_{2,1}$ such that we can use the transformation rules and the axioms of the strong bismulation to rewrite $Q_1$ to $P_1$.

Finally, let the process

$$P' \overset{\text{def}}{=} P_1 \triangleleft (P'_2, P'_3, P'_4)$$

and the relation $\rho_2 = \bigcup_{i=1}^4 \rho_{2,i}$. We have $P \sim P'$ and we can use the transformation rule (7) to rewrite $Q$ to $P'$.

- **Case $Q = Q_1 + Q_2$.** By operational semantics of the Choice and definition of trace refinement, we can infer that

$$\text{traces}(P) = \{ r \mid P \xrightarrow{r} P' \land (\exists s_1 \in \text{traces}(Q_1), r \leq_{\rho_1} s_1) \lor (\exists s_2 \in \text{traces}(Q_2) : r \leq_{\rho_1} s_2) \}$$

$$= \{ r \mid P \xrightarrow{r} P_1 \land \exists s_1 \in \text{traces}(Q_1), r \leq_{\rho_1} s_1 \}$$

$$\lor \{ r \mid P \xrightarrow{r} P_2 \land \exists s_2 \in \text{traces}(Q_2), r \leq_{\rho_1} s_2 \}$$

Thus, by the completeness of strong bismulation laws, we can infer that we can rewrite $P \sim P_1 + P_2$ where $P_1$ and $P_2$ are as defined in the set $\text{traces}(P)$. Furthermore, by definition we have $P_1 \leq_{\rho_1} Q_1$ and $P_2 \leq_{\rho_1} Q_2$. Using the induction hypothesis, we can find relations $\rho_{2,1}$ and $\rho_{2,2}$ and processes $P'_1 \sim P_1$ and $P'_2 \sim P_2$ and we use the transformation rules to rewrite $\text{ref}^i_{(\rho_1, \rho_{2,1})}(Q_1) \sim P'_1$ and $\text{ref}^i_{(\rho_1, \rho_{2,2})}(Q_2) \sim P'_2$. Finally, it is immediate that we can use the relation $\rho_2 = \rho_{2,1} \cup \rho_{2,2}$ and the transformation rule (6) to rewrite $Q$ to $P'_1 + P'_2$.

In the next three cases, we make use of the result in [BG94] which states that every finite-state process can be transformed using the axioms of strong bismulation to a process in the head normal form

$$\sum_{i \in I} e_i : P_i + \sum_{j \in J} A^i_j : Q_j.$$  

In the absence of recursion, the processes $P_i$ and $Q_j$ are also in head normal form. In other words, in the absence of recursion, we can rewrite any finite-state process to a strongly bisimilar process that does not contain a Parallel, Restriction, or Close operator.

- **Case $Q = Q_1 \| Q_2$.** The easiest way to prove this case is to first use the axioms of strong bismulation to rewrite $Q$ without the Parallel operator. The resulting process will be simpler, and thus we can apply the induction hypothesis.

- **Case $Q = Q\setminus F$ for $F \subseteq \text{events}(Q')$.** Again, the easiest way to prove this case is to first use the axioms of strong bismulation to rewrite $Q$ without the Restrict operator.
The resulting process is simpler and the theorem therefore follows from the induction hypothesis.

- Case $Q = [Q']_1$. In this case we can use the axioms of the strong bisimulation to rewrite $Q$ without the Close operator. The resulting process is simpler and thus we can apply the induction hypothesis. \hfill $\square$

### B.5 Compositionality in the Smyth Ordering

#### Theorem B.5.1

Trace refinement $\leq^1_\rho$ is compositional with all ACSR operators. That is, if $P \leq^1_\rho Q$, then

1. $A^a : P \leq^1_\rho A^a : Q$
2. if $P$ and $Q$ are compatible with $\epsilon.NIL$ under $\rho$:
   - $\epsilon.P \leq^1_{\rho \cup \{\epsilon, e\}} \epsilon.Q$
3. if $P$ and $Q$ are compatible with $E$ under $\rho$:
   - $(P + E) \leq^1_{\rho \cup Id_E} (Q + E)$
   - $(E + P) \leq^1_{\rho \cup Id_E} (E + Q)$
4. $(P \backslash \rho(F)) \leq^1_\rho (Q \backslash F)$ if $F \subseteq \text{events}(Q)$ where $\rho(F) = \{\rho(\epsilon) : \epsilon \in F\}$
5. $[P]_I \leq^1_\rho [Q]_I$
6. if $P$ and $Q$ are compatible with $E_i$ under $\rho$ for $i = 1, \ldots, 4$ and $E_4 = \epsilon.NIL$ and
   - $\rho' = \rho \cup Id_{E_1 \ldots E_4}$
   - $(\alpha) \text{ if } \rho'(\tau) = \rho'(e)$ and $\rho'(L \cup \overline{T})$ is one-one, where $L = \text{events}(Q) \cup \{e\}$:
     - $P \triangle^\epsilon_{\rho'} (E_1, E_2, E_3) \leq^1_{\rho'} Q \triangle^\epsilon_{\rho'} (E_1, E_2, E_3)$
     - $E_1 \triangle^\epsilon_{\rho'} (P, E_2, E_3) \leq^1_{\rho'} E_1 \triangle^\epsilon_{\rho'} (Q, E_2, E_3)$
     - $E_2 \triangle^\epsilon_{\rho'} (E_1, P, E_3) \leq^1_{\rho'} E_2 \triangle^\epsilon_{\rho'} (E_1, Q, E_3)$
     - $E_3 \triangle^\epsilon_{\rho'} (E_1, E_2, P) \leq^1_{\rho'} E_3 \triangle^\epsilon_{\rho'} (E_1, E_2, Q)$
   - $(\beta) \text{ if } recX.P \leq^1_\rho (recX.Q)\text{ if } recX.P \leq^1_\rho (recX.Q)$

A note should be made about the compositionality with the parallel operator; unlike in
the Hoare ordering, the restriction that the processes \( Q \) and \( E \) do not share resources is not needed.

**Proof:** We next prove the compositionality with the parallel operator. The proofs for the other operators are similar to those of trace refinement in the case of the Hoare ordering.

We prove that, if \( P \leq_{\rho} Q \) then for any process \( E \) we have

\[
(P \parallel E) \leq_{\rho \cup \text{Id}_E} (Q \parallel E)
\]

if \((\rho \cup \text{Id}_E) \setminus (L \cup \overline{T})\) is a one-to-one, complement-preserving function where \(L = \text{events}(Q) \cup \text{events}(E)\).

Let \( s \in \text{traces}(Q \parallel E) \). Then by operational semantics, there exist \( s_1 \in \text{traces}(Q) \) and \( s_2 \in \text{traces}(E) \), such that \( s \) is obtained by the parallel composition of \( s_1 \) and \( s_2 \). Since \( P \leq_{\rho} Q \), then there exists \( r_1 \in \text{traces}(P) \) such that \( r_1 \leq_{\rho} s_1 \). Also, we have \( s_2 \leq_{\text{Id}_E} s_2 \).

We claim that there exists a trace \( r \in r_1 \parallel s_2 \) such that \( r \in \text{traces}(P \parallel E) \) and \( r \leq_{\rho \cup \text{Id}_E} s \).

The proof proceeds in a similar fashion as the proof of Lemma 4.3.1. The resource condition and the event correspondence are met using the same arguments as in proof B.2. The difference is that for a trace \( s \in \text{traces}(Q \parallel E) \), we do not necessarily have \( \text{dur}(s) = \text{minimum}(\text{dur}(s_1), \text{dur}(r_2)) \), because \( s_1 \) and \( r_1 \) may share some resources at a time instance \( 0 \leq t \leq \text{minimum}(\text{dur}(s_1), \text{dur}(r_2)) \), in which case the parallel composition halts (returns \( \epsilon \)).

In this case, since \( \text{traces}(P \parallel E) \) is a prefix-closed set, then there is a trace \( r \in r_1 \parallel r_2 \) that has the same duration as \( s \). \( \square \)
Appendix C

Production Cell Details

C.1 Design Assumptions

Communication events. There are two types of communication events: 1) events that our design solution introduced to coordinate the activities of the distributed controller (Figure 6.2); and 2) events that represent the set of sensors and actuators (Table C.1).

<table>
<thead>
<tr>
<th>Machine</th>
<th>Actuator Values</th>
<th>Sensors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Press</td>
<td>prDownward, prUpward, prStop</td>
<td>prLow, prHigh, prMed</td>
</tr>
<tr>
<td>Robot</td>
<td>ra1Extend, ra1Retract, ra1Stop</td>
<td>* ra1Out, ra1In</td>
</tr>
<tr>
<td></td>
<td>ra2Extend, ra2Retract, ra2Stop</td>
<td>* ra2Out, ra2In</td>
</tr>
<tr>
<td></td>
<td>ra1MagOn, ra1MagOff</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ra2MagOn, ra2MagOff</td>
<td></td>
</tr>
<tr>
<td></td>
<td>rLeft, rRight, rStop</td>
<td>* rAtPr, rAtDb, rAtRt</td>
</tr>
<tr>
<td>Table</td>
<td>rtUpward, rtDownward, rtStopv</td>
<td>rtHigh, rtLow</td>
</tr>
<tr>
<td></td>
<td>rtLeft, rtRight, rtStop</td>
<td>* rtAtFb, rtAtR</td>
</tr>
<tr>
<td>Crane</td>
<td>crToFb, crToDb, crStop</td>
<td>crOnFb, crOnDb</td>
</tr>
<tr>
<td></td>
<td>crLower, crLift, crStopv</td>
<td>* crLow, crHigh</td>
</tr>
<tr>
<td></td>
<td>crMagOn, crMagOff</td>
<td></td>
</tr>
<tr>
<td>Feed Belt</td>
<td>fbStart, fbStop</td>
<td>bEndfb</td>
</tr>
<tr>
<td>Deposit Belt</td>
<td>dbStart, dbStop</td>
<td>bEndb</td>
</tr>
</tbody>
</table>
a straightforward way—e.g., event \texttt{pr\textit{Downward}} signals the press to start moving down and event \texttt{pr\textit{Stop}} signals the press to stop its motion.

The sensors report either digital values (i.e., from switches and photoelectric cells), or continuous values (i.e., from potentiometers.) Since our formalism does not support data values, we represent both types of sensory information with events whose occurrences describe the critical values in the state of the monitored machine. For example, to describe how far the first robot arm is extracted, we use the events \texttt{m1Out}, \texttt{m1In} to describe the facts that the first arm is completely extracted and completely retracted, respectively; intermediate positions are ignored. In Table C.1, the events which represent potentiometers are marked with an asterisk.

<table>
<thead>
<tr>
<th>Table C.2: Resources in the production cell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traveling Crane</strong></td>
</tr>
<tr>
<td>\texttt{crMotor} : motor to move the crane</td>
</tr>
<tr>
<td><strong>Feed Belt</strong></td>
</tr>
<tr>
<td>\texttt{fb} :* portion of the feed belt where crane deposits unforged plates</td>
</tr>
<tr>
<td>\texttt{fbMotor} : motor to move the feed belt</td>
</tr>
<tr>
<td>\texttt{fb}\ usage: detect metal plate pile up</td>
</tr>
<tr>
<td><strong>Elevating Rotary Table</strong></td>
</tr>
<tr>
<td>\texttt{erMotor1} : motor to rotate the table</td>
</tr>
<tr>
<td>\texttt{erMotor2} : motor to raise and lower the table</td>
</tr>
<tr>
<td>\texttt{ert} :* elevating rotary table</td>
</tr>
<tr>
<td>\texttt{ert}\ usage: synchronize with robot arm 1</td>
</tr>
<tr>
<td><strong>Press</strong></td>
</tr>
<tr>
<td>\texttt{press} :* the lower, mobile plate of the press</td>
</tr>
<tr>
<td>\texttt{press}\ usage: synchronize with the robot arms</td>
</tr>
<tr>
<td><strong>Deposit Belt</strong></td>
</tr>
<tr>
<td>\texttt{dbMotor} : motor to move the deposit belt</td>
</tr>
<tr>
<td>\texttt{db} :* portion of the deposit belt where robot deposit forged plates</td>
</tr>
</tbody>
</table>

**Resource requirements.** Table C.2 lists the set of resources in the production cell with the shared resources marked with an asterisk. Dedicated resources consist of the motors to move and rotate the system machines. Shared resources consist of machine parts that can be used by another machine; they are the end of the feed belt near the traveling
crane, the elevating rotary table, and the press. Our design represents both types of resources. In particular, we use the shared resources to prove that our design satisfies safety requirements.

Table C.3: Timing assumptions in the production cell

<table>
<thead>
<tr>
<th>Table C.3: Timing assumptions in the production cell</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Traveling Crane</strong></td>
</tr>
<tr>
<td>$T_{ch}$ = 4 move the crane horizontally between the deposit and the feed belts</td>
</tr>
<tr>
<td>$T_{cv}$ = 1 move the crane vertically to/away from the level of a belt</td>
</tr>
<tr>
<td>$T_{cf}$ =* 1 reserve the feed belt end before dropping a metal plate</td>
</tr>
<tr>
<td><strong>Feed Belt</strong></td>
</tr>
<tr>
<td>$T_{fb}$ =* 1 move a metal plate on the feed belt away from the crane</td>
</tr>
<tr>
<td><strong>Elevating Rotary Table</strong></td>
</tr>
<tr>
<td>$T_{th}$ = 4 rotate the table between the feed belt and the robot</td>
</tr>
<tr>
<td>$T_{tv}$ = 3 move the table vertically between the levels of the feed belt and robot</td>
</tr>
<tr>
<td><strong>Robot</strong></td>
</tr>
<tr>
<td>$T_{rp}$ =* 2 reserve the press before loading/unloading it</td>
</tr>
<tr>
<td>$T_{re}$ =* 1 reserve the table before unloading it</td>
</tr>
<tr>
<td>$T_{rh1}$ = 1 extend arm 1 horizontally toward/away from the press/rotary table</td>
</tr>
<tr>
<td>$T_{rh2}$ = 1 extend arm 2 horizontally toward/away from the deposit belt/press</td>
</tr>
<tr>
<td>$T_{rr1}$ = 2 rotate arm 1 horizontally to face the press</td>
</tr>
<tr>
<td>$T_{rr2}$ = 3 rotate arm 1 horizontally from the press to the deposit belt</td>
</tr>
<tr>
<td>$T_{rr3}$ = 2 rotate arm 2 horizontally to face the press</td>
</tr>
<tr>
<td><strong>Press</strong></td>
</tr>
<tr>
<td>$T_{pb}$ = 2 move the press to bottom position</td>
</tr>
<tr>
<td>$T_{pm}$ = 2 move the press to middle position</td>
</tr>
<tr>
<td>$T_{pt}$ = 2 move the press to top position</td>
</tr>
<tr>
<td>$T_{pr}$ = 3 press a metal plate</td>
</tr>
</tbody>
</table>

**Timing requirements.** Table C.3 summarizes our timing assumptions. The description of most of these timing assumptions is self-explanatory; we marked those that may require additional explanation with an asterisk. These timing assumptions model the time to reserve shared resources before carrying a task. They have been added in our design solution for two reasons. One reason is that we have tried to constrain our design solution to the set of sensors given in the original description [Lin95]. In particular, this set of sensors does not include a sensor to report the state of the feed belt end which is near
the crane. We use the timing assumption $T_{fb}$ to solve this problem. The second reason is that our formalism lacks data variables that can be used to model continuous factors. In particular, we can not model the position of a robot arm while it is extracting. This in turn may result in safety violations that temporarily occur during the occurrences of discrete events that mark the initial and final position of the robot arm. We use the timing assumptions $T_{cf}$, $T_{rp}$ and $T_{re}$ to solve this problem. Finally, the time constants $T_{rp}$ and $T_{re}$ represent respectively the times which the robot needs to reserve the press before loading or unloading it and to reserve the elevating rotary table before unloading it. These time constants are used while the robot arm extracts towards or retracts away from the press and the elevating rotary table.

**Initial states.** The machines in the production cells are initially in the following states: The deposit belt is running possibly conveying some metal plates; the crane is near the deposit belt and ready to pick up a metal plate; the feed belt is running possibly conveying some metal plates; the elevating rotary table has a metal plate on it and points at the robot; the robot has its first arm retracted at the elevating rotary table; and the press started forging a metal plate.

### C.2 Component Specification

**Elevating Rotary Table** The overall detailed specification of the elevating rotary table consists of the process

$$ElevRotTable_3 \overset{\text{def}}{=} (ERTController_3 \parallel ERTModel_3) \setminus F$$

where the set of synchronization events between the controller and model of the elevating rotary table is

$$F = \{ \text{rtLeft, rtAtFb, rtRight, rtAtRb, rtStoph}$$
$$\text{rtUpward, rtHigh, rtDownward, rtLow, rtStopv} \}.$$  

The controller process for the elevating rotary table is described in Figure C.1. The model process is described by the ACSR process

$$ERTModel_3 \overset{\text{def}}{=} ERTHoriMove \parallel EERTVertMove$$

with the subprocesses described in Figure C.2. In this design of the model of the elevating rotary table, the table can simultaneously rotate and move vertically.
The robot can be described by the following ACSR process:

\[
Robot \overset{\text{def}}{=} (RC\text{ontroller} \parallel RN\text{odel}) \setminus F
\]

where the set of restricted events is

\[
F = \{ r\text{Left}, r\text{Right}, r\text{Stop}, r\text{AtPr}, r\text{AtDb}, r\text{AtRt},
\quad r\text{a}1\text{Extend}, m\text{1Out}, m\text{1Retract}, m\text{1In}, m\text{1Stop},
\quad r\text{a}2\text{Extend}, m\text{2Out}, m\text{2Retract}, m\text{2In}, m\text{2Stop} \}
\]

Figures C.3 and C.4 show the various components in a detailed specification of the robot controller. Figures C.5 and C.6 show the detailed description of the robot model.

By inspecting the robot controller, it is easy to see that our design enforces the following safety requirements: The first arm of the robot extends only when the elevating rotary table points to the robot. This is guaranteed in part by the reservation step of the resource \emph{crt}
Figure C.2: Components of a model for the elevating rotary table
in the process GetNewBlank of the $RController_3$ of Figure C.4. In addition, when the first arm is extending or retracting, the elevating rotary table does not move or rotate. This is guaranteed by holding the resource $ert$ in the process PickUpBlank which executes between the events $raiExtend$ and $raiStop$. Similar restrictions are imposed on the first and second arms of the robot when they operate at the press.

Press. Figures C.7 and C.8 show the detailed description of the controller and model processes for the press, respectively.
Figure C.4: Robot controller (part 2 of 2)
Figure C.5: Robot model (part 1 of 2)
Figure C.6: Robot model (part 2 of 2)
Figure C.7: Press controller
Figure C.8: Press model
C.3  Analysis in VERSA

//.................................................................................................
// Safety Requirements: Part 1
// - no out of range motion
//.................................................................................................
// Crane doesn’t move out of range:
CTest = {} : CTest + (crCrash,1).NIL ;
CSafe = (Crane_3 || CTest) \ {crCrash} ;
// ---Result:
:-) CSafe !
CSafe <1> --(crCanLoad,1)--> no-name
   <2> --{}-- > no-name
[CSafe] :-0 show stats
State machine contains 32 reachable states (0 deadlocked), 37 edges.
Edges represent 17 timed transitions,
    14 internal actions, and
    6 external untimed actions.
LTS is non-Zeno.
3 non-deadlocked states are capable of stopping the clock.
Time to compute LTS: 0.06 seconds user time; 0 seconds system time.

//.................................................................................................
// Press and Robot don’t collide:
PR = [(Press_3 || Robot_3) \ {prLoaded, prUnloaded}]
   \ {press, rMotor, db, ert} ;
PRTTest = {} : PRTTest + (prCrash,1).NIL
         + (raiCrash,1).NIL + (ra2Crash,1).NIL + (rbCrash,1).NIL ;
PRSafe = (PR || PRTTest) \ {prCrash, raiCrash, ra2Crash, rbCrash} ;
Result:

PRSafe!

PRSafe <1> --{(press,2),(ert,1),(rMotor,0),(db,0)}--> no-name

[PRSafe] :-0 show stats

State machine contains 120 reachable states (0 deadlocked), 127 edges.

Edges represent 36 timed transitions,

80 internal actions, and

11 external untimed actions.

LTS is non-Zeno.

8 non-deadlocked states are capable of stopping the clock.

Time to compute LTS: 0.45 seconds user time; 0.01 seconds system time.

// Overall design is safe:
// no out of range motion and no machine collision

// SafetyTest receives a "crash" event and deadlocks.
SafetyTest = {} : SafetyTest + (prCrash,1).NIL + (ra1Crash,1).NIL
        + (ra2Crash,1).NIL + (rbCrash,1).NIL
        + (rtCrash,1).NIL + (crCrash,1).NIL;

// SafeDesign will be used for the analysis:
// If SafeDesign1 has a dealock state, then ProdCell violates the
// first part of the safety requirements; otherwise, ProdCell is safe.
SafeDesign = (ProdCell || SafetyTest)

\{prCrash, ra1Crash, ra2Crash, rbCrash, rtCrash, crCrash\};

// Continued ....
//----Result: SafeDesign is non-Zeno and deadlock free
:-) SafeDesign!

SafeDesign <1> --(blEnddb,1)-- no-name
   <2> --{(press,2),(ert,1),(fbMotor,1),(dbMotor,1),(fb,0),
           (db,0),(crMotor,0),(erMotor1,0),(erMotor2,0)}
   -- no-name

[SafeDesign] :-0 show stats
State machine contains 130554 reachable states (0 deadlocked),
   277477 edges.
Edges represent 8784 timed transitions,
   180333 internal actions, and
   88360 external untimed actions.

LTS is non-Zeno.
4347 non-deadlocked states are capable of stopping the clock.

Time to compute LTS: 634.71 seconds user time; 688.98 seconds system time.

//==================================
// Safety: Part 2
//==================================
// Travelling crane: doesn’t drop plates
// We use CSafe to get rid of the crCrash event in the crModel_3
CDesign = ([ CSafe] {crMotor, fb}) \ {*}
   % [ {tau / crUnloaded, tau / crMagOn} ];

CSafe2 = {} : CSafe2
         + (crCanLoad,1).scope(Wait, dummy, Tch+2*Tcv, NIL, CS1, NIL);
CS1    = (’crSafe,1).scope(Wait, dummy, Tcf, NIL, CS2, NIL);
CS2    = (’crMagOff,1).scope(Wait, dummy, Tch, NIL, CSafe2, NIL);

// Continued ....
// ---Result:
:-) CDesign == CSafe2 ?
   ufifailed--followingpaircouldnotbematched:
   <[CSafe]{crMotor,fb}\{*
   %[{Tau/crUnloaded,Tau/crMagOn},{}],{}:CSafe2
   +
   (crCanLoad,1).scope(Wait, dummy, (4*(2*1)),NIL,CS1,NIL)>
   --followingpairwasmatched:
   <CDesign,CSafe2>
   false(byprioritizedstrongequivalence)
   true(byprioritizedweakequivalence)
//--------------------------------------------------------------------------
// Travelling crane doesn’t pile up plates on feed belt
CNoPileUp = [ (CrController_3 || FBController_3) \ {crUnloaded}
   ] {fb, crMotor, fbMotor};
//----Result:
:-) CNoPileUp!
CNoPileUp <1> --(crCanLoad,1)-- no-name
   <2> --{(fb,0),(crMotor,0),(fbMotor,0)}-- no-name
[CNoPileUp] :-0 show stats
State machine contains 403 reachable states (0 deadlocked), 853 edges.
Edges represent 170 timed transitions,
   5 internal actions, and
   678 external untimed actions.
LTS is non-Zeno.
228 non-deadlockedstatesarecapableofstoppingtheclock.
Time to compute LTS: 0.4 seconds user time; 0.03 seconds system time.

// Continued ....
// Press does not close when a robot arm is inside
// use PSafe to eliminate the prCrash event.

RPSafe2 = [(RController_3 || PSafe) \{prUnloaded, prLoaded}] {press, rMotor, db};

//----Result:
:-) RPSafe2!

RPSafe2 <1> {press,2},(rMotor,0),(db,0)---> no-name
    <2> {ert,1},(press,2),(rMotor,0),(db,0)---> no-name

[RPSafe2] :-0 show stats
State machine contains 449 reachable states (0 deadlocked), 662 edges.
Edges represent 167 timed transitions,
    176 internal actions, and
    319 external untimed actions.
LTS is non-Zeno.
128 non-deadlocked states are capable of stopping the clock.

Time to compute LTS: 0.47 seconds user time; 0.02 seconds system time.

// Continued ....
// Liveness Requirements

// Deposit belt -- Liveness

DBLiveness = rec X. ({}:X
    + (blEnddb,1).(('crCanLoad,2).X
        + (tau,2).rec Y. ({}:Y + ('crCanLoad,2).X)));

AbstDB = DepositBelt_3 \ {*};

// --- Result:
:-) AbstDB == DBLiveness ?

ufi failed -- following pair could not be matched:
    <DepositBelt_3\{*},rec X.({}:X)
    +
        (blEnddb,1).
            (('crCanLoad,2).X
            +
                (tau,2).rec Y.({}:Y + ('crCanLoad,2).X)>)>

-- following pair was matched:
    <AbstDB,DBLiveness>

false (by prioritized strong equivalence)
    true (by prioritized weak equivalence)

// Robot and Press -- Bounded delay

#define Tx 2*(Trp + Trh1 + Trh2)   // time robot uses press exclusively

#define Tw max(Trh1 + Trr1, Tpr + Tpm + Tpb) \ 
    + max(Trr2+ 2*Trh2 + Trr3, Tpb) \ 
    + max(Trr3 + Trr2 + Trr1, Tpt)

// Continued ....
// Need to use PSafe to remove the crash events in Press_3
PTest = {}; PTest + (prCrash,1).NIL;
PSafe = (Press_3 || PTest) \ {prCrash};

// Abstract description of the design
RPDesign = [ (RSafe || PSafe) \ {prLoaded, prUnloaded} 
   ] {press, rMotor, db, ert} 
   % [ { tau / rtUnloaded, tau / rAtPr, tau / rAtDb } ];

// Tester that deadlocks if bounded delay requirement is violated:
RPTestDelay = scope(Wait, dummy, infty, NIL, NIL, (ra1MagOn,1).SDelay);
SDelay = scope(Wait, dummy, Tx+Tw, NIL, NIL, (rBackAtRt,1).RPTestDelay);

RPLive = ( RPDesign || RPTestDelay) \ {ra1MagOn,rBackAtRt};

// ----Result:
:-) RPLive!
RPLive <1> --{(ert,1),(press,2),(rMotor,0),(db,0)}--> no-name
[RPLive] :-0 show stats
State machine contains 184 reachable states (0 deadlocked), 188 edges.
Edges represent 58 timed transitions,
   130 internal actions, and
   0 external untimed actions.
LTS is non-Zeno.
Time to compute LTS: 0.49 seconds user time; 0.03 seconds system time.
Appendix D

The Semantics of ACSR

The semantics of an ACSR process can be defined as a labeled transition system where
the transition relation, \( \rightarrow \), is constructed according to the rules in Table D.1.

The labeled transition system \( \rightarrow \) can be extended to a trace transition system,
\( \rightarrow^* \), in a straightforward manner. This provides a trace semantics for ACSR. To define
\( \rightarrow^* \) from \( \rightarrow \), we use the function \( \hat{s} \) which removes the \( \tau \) events from the sequence \( s \).
That is, \( \hat{s} \) is a sequence of visible actions only, i.e., time and resource consumption and
non-\( \tau \) communication events.

**Definition D.0.1** Let \( s \) be an ACSR trace. We say that \( P \xrightarrow{s} P' \) (and simply write
\( P \xrightarrow{s} \)) if there exists a sequence of actions \( r = a_1 \ldots a_n \) and processes \( P_1, \ldots, P_n \) such that
\[
P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} P_n \equiv P'
\]
and \( s = \hat{r} \).

Similarly, \( P \xrightarrow{\hat{s}} P' \) if there exists a sequence of actions \( r = a_1 \ldots a_n \) and pro-
cesses \( P_1, \ldots, P_n \) such that
\[
P \xrightarrow{a_1} P_1 \xrightarrow{a_2} \ldots \xrightarrow{a_n} P_n \equiv P'
\]
and \( s = \hat{r} \).

Furthermore, \( P \xrightarrow{c} P \) and \( P \xrightarrow{c} \).

Note that if \( P \xrightarrow{s} \) then \( P \xrightarrow{s^*} \); similarly, if \( P \xrightarrow{ss'} \) then \( P \xrightarrow{s} \).

**Definition D.0.2** The set of traces of an ACSR process \( P \) is defined as follows:
\[
\text{traces}(P) \overset{\text{def}}{=} \{ s | P \rightarrow^* \} .
\]

188
Table D.1: ACSR transition relation

<table>
<thead>
<tr>
<th>ActT</th>
<th>ActI</th>
<th>ActTZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A^{t'} : P \xrightarrow{\alpha} A^{t'-1} : P$</td>
<td>$e, P \xrightarrow{\tau} P$</td>
<td>$P \xrightarrow{\alpha} P'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ChoiceL</th>
<th>ChoiceR</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
<td>$Q \xrightarrow{\alpha} Q'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ParT</th>
<th>ParIL</th>
<th>ParCom</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{A_t} P', Q \xrightarrow{A_t} Q'$</td>
<td>$P \parallel Q \xrightarrow{A_t \cup A_{t'}} P' \parallel Q'$</td>
<td>$P \parallel Q \xrightarrow{A_t} P' \parallel Q'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ScopeCT</th>
<th>ScopeCI</th>
<th>ScopeT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{A} P'$</td>
<td>$P \Delta_t(Q, R, S) \xrightarrow{\alpha} P' \Delta_{t-1}(Q, R, S)$</td>
<td>$R \xrightarrow{\alpha} R'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ScopeI</th>
<th>ScopeE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S \xrightarrow{\alpha} S'$</td>
<td>$P \Delta_t(Q, R, S) \xrightarrow{\tau} Q$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ScopeE</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \Delta_t(Q, R, S) \xrightarrow{\tau} Q$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ResI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P \xrightarrow{\alpha} P'$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CloseT</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[P]_I \xrightarrow{A_2 \cup A_2} [P']_I$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CloseI</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[P]_I \xrightarrow{\tau} [P']_I$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rec</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \xrightarrow{\alpha} C$</td>
</tr>
</tbody>
</table>
The set of maximal traces of an ACSR process $P$ is defined as follows:

$$\text{maxtraces}(P) \overset{\text{def}}{=} \{ s \mid P \xrightarrow{s} P' \land P' \not\xrightarrow{*} \}.$$ 

Note that the set of traces of a process is prefix-closed, while the set of maximal traces is not. The two definitions are often used interchangeably in the literature.
Bibliography


Index of Notation

GCSR Notation

action(n) 29 The action (or resource attribute) of a time-consuming node
resrc(n) 29 The set of resources used in the time-consuming node n
name(n) 29 The Name attribute of the reference node n
inside(n) 29 Set of GCSR processes inside the compound node n
close(n) 29 The Close attribute of the compound node n
restrict(n) 29 The Restrict attribute of the compound node n

\( \mathcal{T}_{GA}(G) \) 46 Translation a GCSR process G to an ACSR process
\( \mathcal{T}_{AG}(P) \) 49 Translation of an ACSR process P to a GCSR process
\( \mathcal{T}_{GG}(G) \) 51 Graphical transformation of a GCSR process G

\( G \equiv G' \) 54 Labeled graph isomorphism

\( (q, E) \xrightarrow{\alpha} (q', E') \) 37 GCSR labeled transition at the first level

\( (q, E) \xrightarrow{\alpha} (q', E') \) 39 GCSR labeled transition at the second level

ACSR Notation

\( \mathcal{R} \) 66 Domain of resource names
Act 68 Set of ACSR actions ranged over by \( A, B, C \)
res(A) 77 Set of resources in the action A

\( \mathcal{L} \) 66 Domain of event labels (or names)
Evts 68 Set of ACSR events ranged over by \( e, e', e_1, l, l' \)
\( \alpha, \beta, \gamma \)  ACSR events or actions
\( \alpha \prec \beta \) 43 \( \alpha \) is preempted by \( \beta \)

Proc 65 Set of ACSR processes ranged over by \( P, Q, E, R \)
res(P) 77 Set of resources in the ACSR process P
actions(P) 79 Set of actions in the ACSR process P
events(P) 74 Set of events in the ACSR process P
traces(P) 188 Unprioritized set of traces of the ACSR process P, ranged over by \( r, s \)
events(s) 68 Set of event labels in trace s

event(s, t) 71 Sequence of events in trace s at time t

α ⊆ event(s, t) 71 Event sequence α is a subsequence of event(s, t), possibly interspersed with other events

dur(s) 71 Time duration of trace s

time(s, e) 71 Set of times when event e occurs in trace s

≈ π 46 Prioritized strong equivalence

≈ π 46 Prioritized weak equivalence

P ∥ H 60 Hides resources in H from the behaviors of P

Refinement Notation

r ⪯ ρ s 69 Trace refinement where r uses fewer resources than s

r ⪰ ρ s 69 Trace refinement where r uses more resources than s

traces(P) ⪯ ρ traces(Q) 72 Hoare-ordering extension of trace refinement

P ⪯ ρ Q 75 Short hand notation for traces(P) ⪯ ρ traces(Q)

traces(P) ⪯ ρ traces(Q) 72 Smyth-ordering extension of trace refinement

P ⪯ ρ Q 75 Short hand notation for traces(P) ⪯ ρ traces(Q)

traces(P) ⪯ ρ traces(Q) 72 Egli-Milner ordering extension of trace refinement

P ⪯ ρ Q 75 Short hand notation for traces(P) ⪯ ρ traces(Q)

coref°_{ρ, ρ₂}(P, I) ≈ Q 79 Resource-closed transformation relation

ref°_{ρ, ρ₂}(P) ≈ Q 81 Hoare-ordering transformation scheme

ref°_{ρ, ρ₂}(P) ≈ Q 86 Egli-Milner ordering transformation scheme

Local(P, ρ) 74 Set of local events of P under ρ

Id_{E₁ \ldots Eₙ} 75 Identity defined over the set of events of E₁, ···, Eₙ