Contracts in Programming and in Enterprise Systems

– Progress Report –

Tom Hvitved
Department of Computer Science
University of Copenhagen
Denmark

2009

Preface

The document at hand is my *progress report*, which concludes the first two years (part A) of my Ph.D. program. I am enrolled in a four-year Ph.D. program (Danish: 4+4-*ordningen*), meaning that I began the Ph.D. program with one year of remaining master level studies – in particular before writing a masters thesis. The purpose of a progress report is to present the current status of research, and to present the open questions and ideas for research topics for the last two years (part B) of the Ph.D. program.

A progress report also counts as a masters thesis of 30 ECTS, but due to the open-ended nature of a progress report, a progress report is typically somewhat different from an ordinary masters thesis. In this report I have chosen to include two topics of research, conducted over the last year. The first part is a technical report titled "Foundations for Programming By Contract in a Concurrent and Distributed Environment", which is co-written with Anders Starcke Henriksen and supervised by Andrzej Filinski. The content of the technical report has resulted in a workshop paper, submitted and accepted for the "Games, Business Processes and Models of Interactions (SGI 2009)" workshop. The paper [25] is not included in this report.

The second part of the progress report is a survey titled "Contracts in Enterprise Systems", which serves as preliminary studies for future work on business contract formalization. This part is supervised by Andrzej Filinski and Fritz Henglein.

The context of my work is the collaborative research project "3rd generation Enterprise Resource Planning" (3gERP), which aims to develop a standardized, yet highly configurable and flexible, global (i.e., not restricted to a particular market or industry) ERP system for small and medium sized enterprises. The complete project description is available in [1]. Due to strategic changes in the aim of the 3gERP project, the second part of my progress report has only recently been considered a topic of interest – it therefore evidently bears the mark of "work in progress".

Tom Hvitved
Copenhagen, August 14, 2009

Abstract

In the first part of this report, we present an extension of the *programming-by-contract* (PBC) paradigm to a concurrent and distributed environment. Classical PBC is characterized by *absolute conformance* of code to its specification, *assigning blame* in case of failures, and a *hierarchical*, *cooperative* decomposition model – none of which extend naturally to a distributed environment with multiple administrative peers. We therefore propose a more nuanced contract model based on quantifiable *performance* of implementations; *assuming responsibility* for success, and a fundamentally *adversarial* model of system integration, where each component provider is optimizing its behavior locally, with respect to potentially conflicting demands. This model gives rise to a game-theoretic formulation of contract-governed process interactions.

We introduce an abstract model of communication, which assumes no common model of computation at the peers in the distributed system. An *implementation* consists of a set of processes and possible delegation, which defines a *strategy* for the set of games specified by the set of committed contracts. Verification of correctness is introduced via *contract portfolio conformance*, which generalizes the idea of Hoare triple validity (underlying classical PBC). Contract portfolio conformance corresponds to a mix between partial- and total correctness, called *timed total correctness*, and we show that portfolio conformance supports compositional reasoning.

In the second part of this report, we have conducted a survey on both theoretical and practical approaches to *contract formalization*. By contract formalization we mean representation of (business) contracts in computer systems, to encompass automatic validation, execution, and analysis of contracts. These activities are collectively referred to as *contract lifecycle management* (CLM), and the typical aspects of a CLM system include (1) contract creation, (2) contract negotiation, (3) contract approval, (4) contract execution, and (5) contract analysis. Our survey concludes with a summary of the pros and cons of the covered approaches, and a perspective towards future work.

The connection between the two topics of this report may at first seem absent; however, some aspects are in fact quite similar: In the programming setting, a contract specifies an interface, according to which the program is expected to behave. A program is said to be *correct*, if it can be verified (formally) that the program conforms with the specification/contract. In the business setting, a contract also acts as an interface; not between software components, but between businesses. Now the "program" is in general not a piece of software, rather it is a *business process* (workflow), and correctness is a matter of checking for *contract compliance*. We therefore propose that some of the analyses – even though they have a different flavor – are in fact not that different. We summarize some of the correspondences below:

Programming	**Business**
– Program	– Business process (workflow)
– Specification	– Contract
– Correctness	– Compliance
– Program extraction (from specification)	– Business process derivation (from contract)

Resumé

Denne rapport omhandler to separate emner: specifikation af samtidige og distribuerede programmer, samt repræsentation og håndtering af kontrakter i virksomhedssystemer.

I den første del af denne rapport præsenterer vi en udvidelse af *programming-by-contract* (PBC) paradigmet til et samtidigt og distribueret miljø. Det klassiske PBC paradigme er karakteriseret ved *absolut overensstemmelse* mellem et program og dets specifikation, *tildeling af skyld* i tilfælde af brud på specifikationen, samt en *kooperativ* tilgang til program-/systemudvikling. Disse karakteristika passer ikke umiddelbart ind i et distribueret miljø med flere administrative parter, hvorfor vi foreslår en udvidelse af PBC paradigmet baseret på *relativ opfyldelse* af programspecifikationer, samt at man påtager sig *ansvar* for eventuelle uddelegeringer. Endvidere argumenterer vi for, at et distribueret miljø med flere administrative parter ikke nødvendigvis lægger op til "ubetinget samarbejde", da parterne kan have modstridende interesser. Disse overvejelser udmønter sig i en spilteoretisk formulering af kontrakter (specifikationer), hvori relativ opfyldelse modellers via løbende betalinger.

For at modellere (distribueret) processinteraktion, præsenterer vi en abstrakt kommunikationsmodel, der ikke er bundet til en specifik beregningsmodel. En *implementering* består af en mængde af processer fra kommunikationsmodellen, samt en mulighed for at uddelegere forpligtelser. Implementeringen udgør dermed en *strategi* for mængden af spil, der er defineret i de indgåede kontrakter. Vi indfører begrebet *contract portfolio conformance* – der er en metode til at verificere, at en implementering er en *vindende strategi* – og vi viser at denne definition tillader kompositionelt ræsonnement.

I den anden del af denne rapport præsenterer vi en oversigt over såvel praktiske, som teoretiske tilgange til *kontraktformalisering*. Med kontraktformalisering menes der elektronisk repræsentation af (virksomheds)kontrakter, med det formål at kunne udføre automatisk validering, eksekvering og analyser af disse. Disse aktiviteter samles under betegnelsen *contract lifecycle*

management (CLM), og et typisk CLM system omfatter følgende aspekter: (1) kontraktoprettelse, (2) kontraktforhandling, (3) kontraktgodkendelse, (4) kontrakteksekvering, samt (5) kontraktanalyse. Vi afslutter gennemgangen med en opsummering af de forskellige tilganges kvaliteter, med henblik på fremtidig forskning.

Ved første øjesyn kan det være svært at se den umiddelbare sammenhæng mellem de to dele af rapporten, men vi påstår at der er flere lighedspunkter: I softwarekontekst specificerer en kontrakt et interface, som det pågældende program forventes at overholde. Et program siges at være *korrekt*, hvis det kan bevises formelt, at programmet overholder dets interface. I virksomhedskontekst specificerer en kontrakt ligeledes et "interface", men ikke imellem programmer, snarere imellem virksomheder. I dette kontekst kan programmet anskues som en *forretningsproces* (arbejdsproces), og korrekthed svarer til en garanti for, at kontrakten overholdes (såfremt arbejdsprocessen følges). Vi forudser derfor at mange af de teoretisk, såvel som praktiske, aspekter i formalisering- og analyse af forretningskontrakter/programspecifikationer vil have mange ligheder. Vi opsummerer nogle af disse ligheder i tabellen nedenfor:

Software	**Forretning**
– Program	– Forretningsproces (arbejdsproces)
– Specifikation	– Kontrakt
– Korrekthed	– Kontraktopfyldelse
– Ekstrahering af program (fra specifikation)	– Ekstrahering af arbejdsproces (fra kontrakt)

Acknowledgments

First and foremost I thank my supervisors Andrzej Filinski and Fritz Henglein, without whom this work would not have been possible. They have both provided invaluable ideas and feedback to my work, and they have been great sources of inspiration throughout my education.

I also thank my colleges and friends Anders Starcke Henriksen and Morten Ib Nielsen, for both providing feedback to preliminary versions of this report. Furthermore I acknowledge Anders Starcke Henriksen for his willingness to let me use our joint work in this report.

I wish to thank Kim Guldstrand Larsen for agreeing (on short notice) to be examiner on this project.

Last, but not least, I thank my girlfriend Annnemette Witt for her love and support.

x

Contents

Chapter 1

Foundations for Programming By Contract in a Concurrent and Distributed Environment

Joint work with Anders Starcke Henriksen

1.1 Introduction

We present in this chapter a new, foundational approach for extending *programming/design by contract* (PBC) [43] to a concurrent and distributed environment. PBC is a paradigm for specifying and verifying computer programs, typically by means of formal *pre-* and *postconditions* for code fragments (Hoare triples [26]). Given a program component c, a precondition A is a predicate over c's inputs (both explicit and implicit) specifying the *requirements* or *assumptions* made by c. If for instance c computes a function on numbers, A could be the requirement that input x satisfies $x \geq 0$. Conversely, a postcondition B is a predicate over both inputs and outputs of c, specifying c's *guarantees* about its outputs, for the given inputs. In the function example, B could specify that the output number r must satisfy $r^2 = x$. A piece of code that satisfies its specification, even in an inefficient or unexpected way (such as returning $r = -\sqrt{x}$), is then said to be *correct*.

The purpose of PBC is to enable modular design and implementation of programs, by establishing detailed specifications for all module interfaces, in such a way that correctness of the whole program (i.e., top-level module) follows from the correctness of all the component modules. In particular, any failure of the whole program to satisfy its specification can ultimately be attributed to a violation of a specific pre- or post-condition. The former occurs when a *caller* fails to satisfy the input requirements for invoking a submodule, while the latter indicates the failure of the *callee* to satisfy its output guarantee. In both cases, the implementor of the faulty module is the one who is *blamed*, and the module has to be corrected. This paradigm has also been called *the blame game* [67], due to the (somewhat degenerate) game-theoretic nature of each implementor's incentive being to avoid getting blamed.

The appeal of PBC derives from its *compositional* nature, meaning that the implementor of a module need not be aware of the entire context in which the module is used. Pre- and postconditions define exactly what the module and its context can *expect* from each other, hence when implementing the module nothing else can – or should – be assumed about the environment, and vice versa. For instance, in the numeric example above, it would be wrong of the environment to use the output of c directly as the input for a second invocation of c, even though this latent bug would go undetected if c simply returned $r = \sqrt{x}$.

The purpose of our work is to extend PBC from a classical one-machine setup to a setting where programs run concurrently on (potentially) different machines, owned by different administrative peers, which setting is becoming ever more relevant in the context of *cloud computing* and *software-as-a-service*. Compositionality – as described above – is a crucial feature in PBC,

and it becomes even more important for a distributed computation model, in which knowledge of the entire context is not realistic. Existing work on extending pre/postcondition-style specifications to a concurrent setting have been proposed [29, 52], but to our knowledge there exist no extensions of the PBC paradigm to a distributed environment.

1.1.1 Distributed PBC

In principle, extending PBC to a concurrent, message-passing setting is relatively straightforward. The evident difference from a sequential setting is that pre- and post-conditions must be generalized from one-shot input–output specifications to *communication-protocol* specifications. That is, input requirements now specify what may legitimately be sent *to* a concurrent module or process, while output guarantees capture what must in turn be sent *by* that process. Moreover, both input and output specifications may now in general refer to the entire communication history between the two processes, or some more compact abstraction thereof. Still, no fundamental conceptual changes to the PBC paradigm seem necessary. Session types [28] is an example of such communication-protocol specifications.

On the other hand, a proper account of realistic *distributed* systems does seem to require a complete reassessment of some basic assumptions of PBC. The problematic characteristics here are the notions of *absolute conformance*, *blame assignment*, and the ultimately *cooperative* model of system development.

By absolute conformance we mean that, once a module violates its specification, the "world stops" and no formal guarantees can be given about possible continued execution; the erroneous module must be repaired before the program can be reliably resumed or restarted. In a large-scale distributed environment, however, failures are a fact of life, and though we do need to assign blame for them, the model must also include a robust specification of the relevant recovery procedures, and how any further failures by both parties are to be accounted for.

Moreover, not all failures are equally serious, and some might even be expected to occur in the course of a typical interaction sequence. We thus prefer a contract conformance model based not on a binary outcome, but on a quantitative measure, making it possible to also uniformly express performance characteristics (such as responsiveness) and relative importance of potentially conflicting specifications. Using an economic metaphor, a module (or rather, the module's implementor) is rewarded by its environment for "good" behavior, and/or penalized for "bad" behavior.

We can recover the usual absolute notion of contract satisfaction as a requirement that a *correct* module's accumulated balance is always non-negative;

thus, in particular, such a module will never be the first to break a communication contract. But the key motivation for quantifiable performance contracts is that they support compositional reasoning about module correctness in systems with more than two module producers, as sketched next.

The second problem with traditional PBC is that simple blame assignment is not by itself a proper foundation for composing distributed modules, because the "environment" of a module cannot in general be considered as a monolithic entity that can be blamed as a whole. Consider an example where a company P ("service provider") chooses to implement a web service W_1 using a gateway service W_2 provided by company S ("subcontractor"). If P provides W_1 as a service to some other company C ("client"), then P cannot rely simply on propagating blame to S if W_1 fails as a result of W_2 "failing first". In other words, P is still *responsible* to C for the correct behavior of W_1 – and cannot be excused by W_2 failing. However, S is in turn responsible to P for W_2, which could imply that S has to pay some fine to P each time W_2 fails. And if P is properly organized, this fine will be sufficient to cover the fine that P has to pay to C.

Again, traditional PBC blame propagation can be seen as a degenerate instance of the responsibility model. The difference is that assuming responsibility for satisfying a contract in general involves more than merely being able to deflect all blame for failure. It is an inherently more robust notion, because it requires a module offering a service to explicitly plan for any or all of its subcontractors not fulfilling their nominal ("happy-path") contracts, and ensuring that any penalties imposed by the service's client can ultimately be recovered from the subservices. In particular, a correct module will never make a high-assurance service (i.e., with a high penalty for failure) rely on a low-assurance one.

The final problem with blame propagation is that it is not compositional: in blame propagation, all modules of the program must be known, since blame can be propagated to any of the modules. In the example above this would imply that C had to know S, as P might propagate blame onto S in case of failure. And S might in turn propagate blame onto its (sub)subcontractors, which means that the entire network must be known. This problem is avoided by requiring that P must assume responsibility for S when servicing C.

Finally, by a cooperative decomposition model in traditional PBC we mean that all the module implementors are – to some degree – ultimately working towards a single goal of building a correct system. Thus, if a module specification is ambiguous or incomplete, the implementor will typically still

opt to implement the specification in the "intended" way, even if he doesn't explicitly stand to gain anything from it. In particular, he would normally not choose a deliberately suboptimal algorithm for solving a problem, nor intentionally cause minor failures – even if such failures are nominally allowed by the performance contract. (For instance, the specification might say that providing a wrong answer to a question is unacceptable, but explicitly declining to answer is a legitimate response – yet simply uniformly refusing to answer would be considered a "bad-faith" implementation.)

In a distributed setting, the implicit assumption of cooperation is not necessarily justified. In the web-service example above it might be *locally optimal* for the subcontractor S to sometimes have W_2 failing (and taking the penalty), if that for instance would make it possible for S to respond to a request from some other (higher-payoff) company O that neither the main contractor P nor the ultimate client C know or care about. Thus, all parties in a distributed system need to recognize that their PBC communication peers may occasionally – or even consistently – defect on contracts, not due to coding errors or legitimate misinterpretation of the specification, but as a deliberate design choice. The adversarial nature also prompts the need for absolute guarantees in contracts (e.g., "must respond after at most 10 seconds") instead of unenforceable promises ("must respond eventually").

In summary of the observations above, we propose an explicitly *game-theoretic* [42] extension of the PBC paradigm. Companies P, S and C above are modeled as *players* (which we also call *principals*). Principals are the responsible actors in the distributed environment, and responsibility is codified in *contracts*, which generalize the pre/postcondition-style specifications of PBC. Each contract is a game between two principals, which specifies *what* should be communicated between them, and *when*. This notion respects compositionality, as bilateral contracts only mention the exact part of the context to whom the principal has commitments (cf. the discussion on blame propagation above). The *moves* of principals in the game are what they communicate to each other in each round. Guarantees are quantified by assigning to each transition of the game state a *payoff*, which can be thought of as the incremental payment to the first player from the second, resulting from the transition. (Since communication games may go on indefinitely, we assign payoffs also to non-terminal states of the games.) Such a payoff may represent either a payment for services properly rendered, or a fine for unsatisfactory performance. The precise game-theoretic formulation of a contract is therefore an *infinite, simultaneous, zero-sum, two-person game*. In general, each player participates in multiple, concurrent games, and aims to maximize his total payoff, rather than to do well in any particular game.

Even though each contract specifies a zero-sum game, and all principals may be assumed to be rational and enter contracts in expectation of a positive

payoff, it doesn't necessarily mean that at least one principal has to lose. The reason is that the system is considered *open*, hence the "losing" principal may actually have an overall positive payoff, as a result of some unknown contracts with an unspecified environment, or ultimately with "nature".

A contract describes a *logical* commitment between two principals, but not how communication is enacted *physically*. In order to fulfill its contractual obligations, each principal implements an overall *strategy* for playing all of its communication games. An implementation consists of a set of *processes* for performing actual computation and communication, and a means of *delegation* to other principals. The latter makes it possible to satisfy a logical commitment without doing the actual communication oneself. (In some cases it may in fact not even be possible to be in charge of the communication oneself.)

Having described how contracts are extended from one-shot input–output specifications (pre/postconditions), we need to generalize sequential programs to concurrent processes. The aim is to consider a simple model of communication, which assumes no common computational model at the peers in the network. Our model of communication is inspired by the Input Output Timed Automaton (IOTA) model [9], in which messages are sent asynchronously between automata. Unlike process calculi such as CSP [27], CCS [44], and the π-calculus [45], we assume no advanced synchronization primitives, and processes are described *extensionally* (*black box*), rather than an *intentionally*, to reflect that the internal structure of a process may not be known. Furthermore, there is no possibility of *refusing* input as in for instance CSP, which means that contracts can be defined on *traces* of *actual* communication, rather than traces of input/output requests.

The link between implementations and contracts is established via a notion of *contract portfolio conformance*, which generalizes the definition of Hoare triple validity. Contract conformance is defined as a *safety* property, meaning that all violations happen in finite time (contracts are falsifiable). This is partly due to the fact that implementations are black boxes (their internal organization cannot be inspected), and monitoring of contracts should be possible only by inspecting the observational behavior of implementations. However, this does not imply that contract conformance corresponds to partial correctness of Hoare logic, in which a program stuck in an infinite loop satisfies any specification. Rather it can be seen as a cross between partial correctness and total correctness, which we call *timed total correctness*. By this terminology we mean that is is possible to specify *absolute timed guarantees* (as identified earlier), rather than the less useful "eventually guarantees". The desire for absolute guarantees means that we cannot

use for instance session types, as mentioned earlier, but we show how a simplified – but timed – version of session types can be expressed in the model.

As mentioned earlier, PBC permits modularized reasoning, based on the rule of composition from Hoare logic:

$$\text{If } \models \{A\}\, c_1\, \{B\} \text{ and } \models \{B\}\, c_2\, \{C\} \text{ then } \models \{A\}\, c_1; c_2\, \{C\}$$

We have already elaborated on how bilateral contracts and assuming responsibility for success make for a compositional contract model. But we also show that contract portfolio conformance permits compositional reasoning (generalizing the rule above), which means that the model also enjoys *internal* compositionality (principals can reason modularized).

To summarize the discussion of this subsection, we include a table comparing the differences between classical PBC and our extended model.

Classical PBC	**Distributed PBC**
Absolute conformance	Ongoing performance
Blame propagation	Assuming responsibility
Cooperative decomposition	Adversarial composition
Sequential program	Concurrent process

1.1.2 Chapter Outline

The rest of this chapter is structured as follows:

Section 1.2: We introduce the abstract model of communication. *Processes* are described only by their observational behavior, and we require just one construct: parallel composition. In this section we argue why a dense model of time is problematic, and henceforth base the theory on a discrete model of time.

Section 1.3: We introduce the model of *I/O automata*. Like processes, automata are an abstract model of communication, however this model allows for easier reasoning due to its simplicity. We show that the two models are *equivalent*, which relies on a general notation of *compsets* (Appendix A). The process model can be thought of as a denotational (trace) semantics, and the automaton model as a small-step semantics. The rest of the chapter is based on automata rather than processes.

Section 1.4: We introduce the model of *principals* and *contracts*. Contracts are always bilateral (between two principals), describing the logical commitments for communication.

Section 1.5: We introduce *implementations*, *contract conformance*, and *automaton contracts*. We show that contract conformance permits compositional reasoning.

Section 1.6: In this section we illustrate by example the expressiveness of the model. Examples include *sequential Hoare logic*, *timed session types* and *quality of service agreements*.

All (larger) proofs presented throughout the chapter are enclosed in Appendix B.

1.2 Process Model

In this section we define an abstract model of communication, in which the communicating peers are called *processes*. As mentioned in the introduction, the overall goal is to extend the programming-by-contract paradigm from a centralized setting to a peer-to-peer setting. Before being able to do so, we hence need to describe exactly *what kind of peers* we are interested in specifying. This section (and the subsequent) target that question. In Section 1.4 we return to the question of what a specification is, and who the responsible actors are.

The reader may skip directly to Section 1.3, which defines a model of communication that is equivalent with the process model. However, Definition 1.2.1 and Definition 1.2.2 (below) are also used in that model. The process model is included to illustrate some of the choices that were made in the design of the communication model.

Our model is based on the fact that we cannot assume a global, common model of computation, hence processes are only described by their observational behavior. This approach is different from various process calculi, such as CSP, CCS, and the π-calculus, in which processes are *intensional*. Our *extensional* model is more related to the Input Output Timed Automaton (IOTA) model, and like this model, only actual actions are tracked – not offers (i.e., processes cannot refuse input). As is the case for IOTA, CSP, CCS, and the π-calculus, we use *channels* as an abstraction for an ideal communication medium.

Definition 1.2.1 (Channel)**.** *The set of all channels is denoted $\mathcal{C}$ (countable set). We usually write $\alpha, \beta, \gamma, \ldots$ for concrete channels and $c_1, c_2, \ldots$ for channel variables.*

Unlike CCS and the π-calculus, channels are always directed, and have exactly one sender and one receiver, which is similar to the IOTA model. The reason for this low-level approach is to have as few assumptions about the communication medium as possible. This means for instance that it is not possible for a process to broadcast to several other processes on a single channel – instead the process has to explicitly implement the fan out to the receiving processes via individual channels.

In order to specify processes by their observational behavior, we first define *moves* (the name is inspired by the game-theoretic approach which we return to in Section 1.4).

Definition 1.2.2 (Move over C)**.** *Given a finite set of channels $C \subseteq_{\text{fin}} \mathcal{C}$ and an alphabet $\mathcal{A}_c$ for each $c \in C$, a move m over C is a function such that $m(c) \in \mathcal{A}_c$ for all $c \in C$. The set of all moves over C is denoted $\mathcal{M}_C$ (i.e., $\mathcal{M}_C = \Pi_{c \in C} \mathcal{A}_c$). We usually write $m_1, m_2, \ldots$ for moves.*

A move m over C is a snapshot of what gets communicated on the channels in C, at a particular point in time. Each $\mathcal{A}_c$ contains a special action ε meaning "no action" (the *silent action*). The condition that C be finite is imposed since a move describes an *observation*, which should always be finite.

A *log* (or trace) on a set of channels can now be described as a list of moves with time stamps, describing *what* has happened on the channels and *when*. This intuition is formalized in the following definition.

Definition 1.2.3 (Log over C)**.** *The set of logs over $C \subseteq_{\text{fin}} \mathcal{C}$ with end time $t \in \mathbb{N}$ is defined by*

$$\mathcal{L}_C^t = [0; t) \to \mathcal{M}_C$$

The set of all logs over $C \subseteq \mathcal{C}$ is defined by

$$\mathcal{L}_C = \biguplus_{t \in \mathbb{N}} \mathcal{L}_C^t$$

where $\uplus$ denotes disjoint union. We usually write $l_1, l_2, \ldots$ for logs.

A log l over C with end time t is thus a description of all that has happened on the channels of C *before* time t. We note that all time stamps are in $\mathbb{N}$, which means that we consider a *discrete* model of time – we return later to why a dense model of time is problematic. One problem with the definition above is that logs may often be sparse due to silent actions. When representing a log, one would typically prefer a simpler list notation, for instance

$$[t_1 : (\alpha \mapsto 10), t_2 : (\beta \mapsto 3), t_3 : (\alpha \mapsto 2, \beta \mapsto 5); t_4] \tag{1.1}$$

where $t_1 < t_2 < t_3 < t_4$, and t_4 represents the end of the log. The list then denotes the log $l \in \mathcal{L}_{\{\alpha,\beta\}}^{t_4}$ defined by

$$l(t)(c) = \begin{cases} 10, & t = t_1 \wedge c = \alpha \\ 3, & t = t_2 \wedge c = \beta \\ 2, & t = t_3 \wedge c = \alpha \\ 5, & t = t_3 \wedge c = \beta \\ \varepsilon, & \text{otherwise} \end{cases}$$

The reason for choosing Definition 1.2.3 rather than the compact list representation, is simply to make reasoning about logs easier (in the list representation one needs for instance to make sure that the list is monotonic in the time stamps, which is not necessary when defined as a function on time stamps). However, the intuition of a log should be as of (1.1). Below follows a series of definitions related to logs.

Definition 1.2.4 (End of log)**.** *Given a log $l \in \mathcal{L}_C$ we define the end of log by*

$$\mathrm{eol}(l) = t, \text{ when } l \in \mathcal{L}_C^t$$

Definition 1.2.5 (Log merge)**.** *Given two logs, $l_1 \in \mathcal{L}_{C_1}^t$ and $l_2 \in \mathcal{L}_{C_2}^t$ with $C_1 \cap C_2 = \emptyset$, define the merged log $l_1 \bowtie l_2 \in \mathcal{L}_{C_1 \cup C_2}^t$ by*

$$(l_1 \bowtie l_2)(t')(c) = \begin{cases} l_1(t')(c), & c \in C_1 \\ l_2(t')(c), & c \in C_2 \end{cases}$$

Definition 1.2.6 (Log restriction)**.** *Given a log $l \in \mathcal{L}_C$ it can be restricted to channels $C' \subseteq C$, written $l_{|C'} \in \mathcal{L}_{C'}$, and restricted to time $t \leq \mathrm{eol}(l)$, written $l_{|t} \in \mathcal{L}_C^t$, where*

$$l_{|C'}(t) = l(t)_{|C'}$$
$$l_{|t} = l_{|[0;t)}$$

(We have used ordinary function restriction on the right hand side of the equations.)

Definition 1.2.7 (Log prefix)**.** *For a given log set $\mathcal{L}_C$ define $(\cdot \sqsubseteq \cdot) \subseteq \mathcal{L}_C \times \mathcal{L}_C$ by*

$$l_1 \sqsubseteq l_2 \overset{\text{def}}{\iff} \mathrm{eol}(l_1) \leq \mathrm{eol}(l_2) \wedge l_1 = l_{2|\mathrm{eol}(l_1)}$$

$\sqsubseteq$ defines a partial order on $\mathcal{L}_C$ (reflexive, transitive and symmetric).

The motivation for defining logs is to describe processes only by their observational behavior. A process should hence be described by what happens on the channels it uses, and when – which is exactly what logs do. However, a process should not be any arbitrary relation on input/output logs, the exact requirements are formulated in the following definition.

Definition 1.2.8 (Process)**.** *A process $\mathsf{p} \in \mathfrak{P}$ is a triple*

$$\mathsf{p} = (C_I, C_O, f)$$

where C_I (input channels) and C_O (output channels) are disjoint and finite, and

$$f : \mathcal{L}_{C_I} \to \mathcal{L}_{C_O}$$

is a log transformer, satisfying

$$\forall l \in \mathcal{L}_{C_I}.\mathrm{eol}(l) = \mathrm{eol}(f(l)) \tag{1.2}$$

$$\forall l_1, l_2 \in \mathcal{L}_{C_I}.\forall t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2)). \\ l_{1|t} = l_{2|t} \Rightarrow f(l_1)_{|t+1} = f(l_2)_{|t+1} \tag{1.3}$$

There are several things to note about the definition of a process. First of all we have defined a process to have a set of *fixed*, *finite* and *disjoint* input- and output channels. Input channels are the source of *stimuli* to a process, and output channels are the *reactions* (as mentioned, inspired by the the IOTA model). The reason why channels are fixed is for simplicity: unlike the π-calculus, in which channels can be sent as values, we only consider a static network topology. Future work should investigate the possibility of extending our model with dynamically created channels, so we leave the discussion for now. The disjointness of input- and output channels stems from that fact that all internal communication should be non-observable – hence it makes no sense for a process to communicate (externally) with itself. Finally processes can only communicate on a finite set of channels, as a process should not be able to cope with (and output) infinite information in finite time.

The next thing we observe is that the relation between stimuli and reaction (the log transformer) is *deterministic*. The reason for this approach is that we have distinguished between *internal* nondeterminism and *external* nondeterminism. The latter can be modeled by having an input channel to some external environment, however, the former cannot. It is possible to extend the definition to powersets to model internal nondeterminism as well, but we omit it here for simplicity. The two additional requirements on log transformers are motivated below:

(1.2) This condition states that a log transformer must preserve the end time for logs. Intuitively this means that we always end the observation of input and output at the same time.

(1.3) This condition is called *strict monotonicity*, and guarantees two properties: (a) Processes cannot "change the past", i.e., if output is known for a log l and l' is an extension of l, then the output for l' is an extension of the output for l (i.e., monotonicity with respect to $\sqsubseteq$, as we shall see soon). (b) Processes cannot respond instantly, i.e., if two input logs are equal *before* time t (and possibly differing *at* time t) then output is equal before *and* at time t. This restriction is imposed to reflect the intuition that "reaction takes time", i.e., we will not allow instant (infinitely fast) response.

Observation 1.2.9. It follows from strict monotonicity that the initial output for a process $\mathsf{p} = (C_I, C_O, f)$ is always predetermined, as $\forall l_1, l_2 \in \mathcal{L}_{C_I}.l_{1|0} = l_{2|0}$ holds vacuously. Hence hence $f(l_1)(0) = f(l_2)(0)$.

Lemma 1.2.10 (Strict monotonicity $\Rightarrow$ monotonicity)**.** *Let* $\mathsf{p} = (C_I, C_O, f)$ *be a process. Then* $f : \mathcal{L}_{C_I} \to \mathcal{L}_{C_O}$ *is monotone with respect to* $\sqsubseteq$.

Proof. Assume $l_1 \sqsubseteq l_2$. Then $\mathrm{eol}(f(l_1)) = \mathrm{eol}(l_1) \leq \mathrm{eol}(l_2) = \mathrm{eol}(f(l_2))$ so if $f(l_1) \not\sqsubseteq f(l_2)$, then there exists some $t < \mathrm{eol}(f(l_1))$ such that $f(l_1)(t) \neq f(l_2)(t)$. But since $l_{1|t} = l_{2|t}$ it follows by strict monotonicity that $f(l_1)(t) = f(l_2)(t)$ which is a contradiction. Hence we must have that $f(l_1) \sqsubseteq f(l_2)$ as required. □

Another thing to notice in the definition of a process is that observations always end at *finite* time. The reason for this approach is that we think of logs as observations, and we argued earlier that observations should always be finite. However, if we were to extend logs to have also infinite end time, then by the strict monotonicity condition (which can be compared with the continuity condition of denotational semantics [68]), all behavior on infinite logs would be uniquely determined by the behavior on finite logs. Hence infinite logs will not make processes more expressive.

We now introduce the concept of *process composition* which is – in fact – the only operator in the model of processes.

Definition 1.2.11 (Parallel composition)**.** *Given* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *such that* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$*, parallel composition is defined by* $\mathsf{p}_1 \parallel \mathsf{p}_2 = (C_I, C_O, f)$*, where*

$$C_{\text{int}} = (C_I^1 \cap C_O^2) \cup (C_I^2 \cap C_O^1) \qquad \text{(Internal channels)}$$
$$C_I = (C_I^1 \cup C_I^2) \setminus C_{\text{int}} \qquad \text{(Input channels)}$$
$$C_O = (C_O^1 \cup C_O^2) \setminus C_{\text{int}} \qquad \text{(Output channels)}$$

The composed log transformer is defined iteratively. For $l \in \mathcal{L}^t_{C_I}$ *define* $\mathcal{I}^1_n \in \mathcal{L}^t_{C_O^1}$ *and* $\mathcal{I}^2_n \in \mathcal{L}^t_{C_O^2}$ *for all* $n \in \mathbb{N}$ *by:*

$$\mathcal{I}^1_0 = \emptyset_{\mathcal{L}^t_{C_O^1}}$$
$$\mathcal{I}^2_0 = \emptyset_{\mathcal{L}^t_{C_O^2}}$$
$$\mathcal{I}^1_{n+1} = f^1((\mathcal{I}^2_n \bowtie l)_{|C_I^1})$$
$$\mathcal{I}^2_{n+1} = f^2((\mathcal{I}^1_n \bowtie l)_{|C_I^2})$$

Then $f(l) = (\mathcal{I}^1_N \bowtie \mathcal{I}^2_N)_{|C_O}$*, where* N *is least such that* $\mathcal{I}^1_N = \mathcal{I}^1_{N+1}$ *and* $\mathcal{I}^2_N = \mathcal{I}^2_{N+1}$.

Observation 1.2.12 (Fixed point)**.** Let $\mathcal{I}^1_n$ and $\mathcal{I}^2_n$ be as of Definition 1.2.11. If there exists an $N \in \mathbb{N}$ such that $\mathcal{I}^1_N = \mathcal{I}^1_{N+1}$ and $\mathcal{I}^2_N = \mathcal{I}^2_{N+1}$ then $\mathcal{I}^1_N = \mathcal{I}^1_{N+k}$ and $\mathcal{I}^2_N = \mathcal{I}^2_{N+k}$ for all $k \in \mathbb{N}$.

The motivation for parallel composition is to be able to construct processes from subprocesses. Composing two processes consists of defining a new

common channel interface (C_I and C_O), and making internal communication (C_{int}) invisible from the outside. We require that two processes cannot share input- or output channels, as channels are always point-to-point.

The definition of parallel composition is a bit involved: the reason for this is internal communication, which needs to be "fed back as input". In the first iteration we calculate the output for both processes (with respect to the common input). Some of the output then needs to be routed internally, which is what happens in the second iteration. But this may in turn result in new internal messages which are included in the next iteration. This iteration is continued until a fixed point is reached. The example below illustrates parallel composition with two concrete processes.

Example 1.2.13. Consider two processes

$$\mathsf{p}_1 = (\{\alpha\}, \{\beta\}, f_1) \quad \mathsf{p}_2 = (\{\beta\}, \{\alpha, \gamma\}, f_2)$$

defined by

$$f_1(l)(t)(\beta) = \begin{cases} 0, & t = 0 \\ 2x, & l(t-1)(\alpha) = x \\ \varepsilon, & \text{otherwise} \end{cases}$$

$$f_2(l)(t)(c) = \begin{cases} x+1, & l(t-1)(\beta) = x \\ \varepsilon, & \text{otherwise} \end{cases}$$

p_1 initially outputs 0 on β and continuously doubles input from α on β. p_2 continuously increments its input from β on both α and γ. The parallel composition $\mathsf{p}_1 \parallel \mathsf{p}_2$ has no input channels and γ as output channel. In order to calculate the output after, say 6 time units, we must apply the composed log transformer to the log $[\cdot; 6]$. The iterative procedure of Definition 1.2.11 produces (using syntax from 1.1):

$\mathcal{I}_0^1 = [\cdot; 6]$
$\mathcal{I}_1^1 = [0 : (\beta \mapsto 0); 6]$
$\mathcal{I}_2^1 = [0 : (\beta \mapsto 0); 6]$
$\mathcal{I}_3^1 = [0 : (\beta \mapsto 0), 2 : (\beta \mapsto 2); 6]$
$\mathcal{I}_4^1 = [0 : (\beta \mapsto 0), 2 : (\beta \mapsto 2); 6]$
$\mathcal{I}_5^1 = [0 : (\beta \mapsto 0), 2 : (\beta \mapsto 2), 4 : (\beta \mapsto 6); 6]$
$\mathcal{I}_6^1 = [0 : (\beta \mapsto 0), 2 : (\beta \mapsto 2), 4 : (\beta \mapsto 6); 6]$
$\mathcal{I}_7^1 = \mathcal{I}_6^1$

$\mathcal{I}_0^2 = [\cdot; 6]$
$\mathcal{I}_1^2 = [\cdot; 6]$
$\mathcal{I}_2^2 = [1 : (\alpha, \gamma \mapsto 1); 6]$
$\mathcal{I}_3^2 = [1 : (\alpha, \gamma \mapsto 1); 6]$
$\mathcal{I}_4^2 = [1 : (\alpha, \gamma \mapsto 1), 3 : (\alpha, \gamma \mapsto 3); 6]$
$\mathcal{I}_5^2 = [1 : (\alpha, \gamma \mapsto 1), 3 : (\alpha, \gamma \mapsto 3); 6]$
$\mathcal{I}_6^2 = [1 : (\alpha, \gamma \mapsto 1), 3 : (\alpha, \gamma \mapsto 3), 5 : (\alpha, \gamma \mapsto 7); 6]$
$\mathcal{I}_7^2 = \mathcal{I}_6^2$

Hence the output is $(\mathcal{I}_6^1 \bowtie \mathcal{I}_6^2)_{|\{\gamma\}} = [1 : (\gamma \mapsto 1), 3 : (\gamma \mapsto 3), 5 : (\gamma \mapsto 7); 6]$.

It is not *a priori* evident that the fixed point of Definition 1.2.11 always exists. However, we will show soon that it does indeed always exist. We will see later, that if we extend the model to using dense time, then the

fixed point will not always exist, meaning that some processes cannot be composed in parallel (which should not be the case).

We first state a lemma saying parallel composition is well-defined, i.e., that the composition of two processes is itself a process.

Lemma 1.2.14. *Let* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *be given processes, and assume that* $\mathsf{p}_1 \parallel \mathsf{p}_2 = (C_I, C_O, f)$ *exists (Definition 1.2.11). Then* $\mathsf{p}_1 \parallel \mathsf{p}_2$ *is a process.*

Proof. In Appendix B. □

The next lemma shows that the fixed point does in fact always exist, i.e., parallel composition is always defined (whenever p_1 and p_2 have compatible channels).

Theorem 1.2.15. *Let* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *be given processes, such that* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$*. Then parallel composition* $\mathsf{p}_1 \parallel \mathsf{p}_2$ *is always defined.*

Proof. In Appendix B. □

We now have a model of processes in which parallel composition is always defined (given that the two processes are compatible). The strength of this model is that it is "intuitive" and deals only with processes as black box entities. The weakness, however, is that it is hard to reason about processes in the model due to the iterative nature of parallel composition. We would now, for instance, be interested in showing that parallel composition is commutative and associative, but the latter is not so straightforward to show. We therefore introduce a new (but equivalent) model of *automata* in the next section, which is easier to reason about than the model of this section.

Theorem 1.2.16 (Parallel composition is commutative)**.** *Given two processes* $\mathsf{p}_1, \mathsf{p}_2 \in \mathfrak{P}$ *then*

$$\mathsf{p}_1 \parallel \mathsf{p}_2 = \mathsf{p}_2 \parallel \mathsf{p}_1$$

(Meaning either both sides are undefined or they are equal.)

Proof. It is obvious from Theorem 1.2.15 that both compositions are either undefined or defined. If both compositions are defined, then it follows directly from commutativity of $\cup$, $\cap$ and $\bowtie$ that the two processes are equal. □

1.2.1 A Digression on Time

The reader may skip the following section and go straight to the next section on automata, as the contents of this section is not a prerequisite for the rest of this chapter. As mentioned in the beginning of this section, we consider a discrete model of time (c.f. Definition 1.2.3 in which $\mathbb{N}$ is the domain of time). An alternative approach is to consider a dense model of time, for instance $\mathbb{Q}_+$, as proposed by Alur and Dill [3]. Now a log with end time q will have type

$$[0; q) \to \mathcal{M}_C \tag{1.4}$$

This definition allows for the possibility of having infinitely many moves in finite time, which we will not allow. Alur and Dill avoid this problem by defining a notion of *progress* (for *timed words* which are much like our notion of logs):

Definition 1.2.17 (Progress)**.** *A log $l \in \mathcal{L}_C$ is said to have* progress *whenever the set $\{q \in \mathbb{Q}_+ \mid \exists \alpha \in C.l(q)(\alpha) \neq \varepsilon\}$ is finite.*

We now need to reconsider the strict monotonicity property (1.3) of log transformers (Definition 1.2.8). We still want "ordinary monotonicity" to follow from strict monotonicity (Lemma 1.2.10), so the question is how much time a process should take to react. One approach is to have some minimal $\delta \in \mathbb{Q}_+$ and require

$$\forall l_1, l_2 \in \mathcal{L}_{C_I}.\forall t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2)).l_{1|t} = l_{2|t} \Rightarrow f(l_1)_{|t+\delta} = f(l_2)_{|t+\delta}$$

However, this is equivalent to a discrete model of time, where $\mathbb{Q}_+$ is partitioned into δ-intervals. Hence if the density of $\mathbb{Q}_+$ is to be fully utilized, the definition of strict monotonicity should be

$$\begin{array}{l} \forall l_1, l_2 \in \mathcal{L}_{C_I}.\forall t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2)). \\ l_{1|t} = l_{2|t} \Rightarrow \forall t' \leq t.f(l_1)(t') = f(l_2)(t') \end{array} \tag{1.5}$$

meaning that a process can be *arbitrarily* fast (but still not infinitely fast). But as the following example shows, introducing a dense model of time in this manner will break the existence of parallel composition.

Example 1.2.18 (Infinite speedup)**.** Consider a dense model of time (1.4) and two processes $\mathsf{p}_1 = (\{\alpha\}, \{\beta\}, f_1)$ and $\mathsf{p}_2 = (\{\beta\}, \{\alpha, \gamma\}, f_2)$ defined by

$$f_1(l)(t)(\beta) = \begin{cases} a, & t = 0 \\ x, & l(1 - \frac{1}{n})(\alpha) = x \wedge t = 1 - \frac{1}{n+1} \\ \varepsilon, & \text{otherwise} \end{cases}$$

$$f_2(l)(t)(c) = \begin{cases} x, & l(1 - \frac{1}{n})(\beta) = x \wedge t = 1 - \frac{1}{n+1} \\ \varepsilon, & \text{otherwise} \end{cases}$$

Both p_1 and p_2 satisfy the requirements for being a process (Definition 1.2.8) and both preserve progress (they both duplicate some of the input with decreasing delay, and p_1 initiates by sending a on β). For instance p_1 transforms the log

$$\left[\frac{2}{3} : (\alpha \mapsto c), \frac{9}{10} : (\alpha \mapsto d), 33 : (\alpha \mapsto e); 40\right]$$

to

$$\left[0 : (\beta \mapsto a), \frac{3}{4} : (\beta \mapsto c), \frac{10}{11} : (\beta \mapsto d); 40\right]$$

However, when composed in parallel we get (intuitively) a process that outputs the following infinite sequence of a's

$$\left[\frac{1}{2} : (\gamma \mapsto a), \frac{3}{4} : (\gamma \mapsto a), \frac{5}{6} : (\gamma \mapsto a), \ldots, \frac{2i-1}{2i} : (\gamma \mapsto a), \ldots\right]$$

Hence when applied to the empty log with end time 1 (which has progress), p_1 and p_2 will produce an output log without progress (the sequence $\{1 - \frac{1}{2i}\}_{i \in \mathbb{N}}$ converges towards 1). In other words, parallel composition is undefined, as the iterative procedure of Definition 1.2.11 never terminates.

The example above shows that even though two processes in separation seem natural (strictly monotone, progress preserving) they constitute in parallel a process that does not preserve progress. What we want, is a model in which "natural" processes (with composable channels) always define in parallel a new process, hence the dense model of time does not work.

So one may wonder why a dense model of time is not problematic in Alur and Dill's model. The reason is that they consider input (accepting) automata only – not input/output automata. Hence it is not possible to exploit density of time to compose two automata with progress to one without, in their model.

1.3 Automaton Model

In the last section we introduced an abstract, extensional model of communication. This model is characterized by the use of a discrete model of time, and processes are described in an asynchronous manner via input/output logs. In this section we introduce the equivalent model of *I/O automata* which takes a *synchronous* approach to communication, as automata react in each time unit. Automata are introduced to avoid the somewhat complicated definition of parallel composition for processes, and the theory and results in the remainder of this chapter are hence based on automata rather than processes.

Definition 1.3.1 (Automaton)**.** *An automaton* $\mathsf{a} \in \mathfrak{A}$ *is a 6-tuple:*

$$\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$

where C_I *(input links) and* C_O *(output links) are disjoint and finite,* S *is the (potentially infinite) set of automaton states, and* $s_0 \in S$ *is the start state.* $\delta_{\mathrm{o}} : S \to \mathcal{M}_{C_O}$ *and* $\delta_{\mathrm{t}} : S \times \mathcal{M}_{C_I} \to S$ *are the output and transition function, respectively: the I/O automaton's output in the current time unit is determined by its internal state, while its next state depends on the current one, and the input received.*

As noted above, the output m_o as a reaction to input m_i is only observed in the *next* time unit. Intuitively this means that reaction "takes time", i.e., it is not possible to provide a response instantly. Even though automata have a notion of *state*, they are still considered extensional rather than intensional; we do not impose any structure on the set of states S, and the automaton does not specify how the transition functions are computed, only what they compute. The definition of parallel composition is now more simple than that for processes (Definition 1.2.11).

Definition 1.3.2 (Parallel composition)**.** *Given* $\mathsf{a}_1 = (C_I^1, C_O^1, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ *and* $\mathsf{a}_2 = (C_I^2.C_O^2, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ *such that* $C_I^1 \cap C_I^2 = C_O^1 \cap C_O^2 = \emptyset$*, parallel composition of the two automata is defined by*

$$\mathsf{a}_1 \parallel \mathsf{a}_2 = (C_I, C_O, S^1 \times S^2, \langle s_0^1, s_0^2 \rangle, \delta_{\mathrm{o}}, \delta_{\mathrm{t}}) \,,$$

where the channels of the composite automaton are given by:

$$C_{\mathrm{int}} = (C_I^1 \cap C_O^2) \cup (C_I^2 \cap C_O^1) \qquad \text{(Internal channels)}$$

$$C_I = (C_I^1 \cup C_I^2) \setminus C_{\mathrm{int}} \qquad \text{(Input channels)}$$

$$C_O = (C_O^1 \cup C_O^2) \setminus C_{\mathrm{int}} \qquad \text{(Output channels)}$$

and its output and transition functions:

$$\delta_{\mathrm{o}}(\langle s_1, s_2\rangle) = (\delta^1_{\mathrm{o}}(s_1) \cup \delta^2_{\mathrm{o}}(s_2))_{|C_O}$$
$$\delta_{\mathrm{t}}(\langle s_1, s_2\rangle, m) = \langle \delta^1_{\mathrm{t}}(s_1, (m \cup \delta^2_{\mathrm{o}}(s_2))_{|C^1_I}), \delta^2_{\mathrm{t}}(s_2, (m \cup \delta^1_{\mathrm{o}}(s_1))_{|C^2_I})\rangle$$

(In the above, for a move $m \in \mathcal{M}_C$, $m_{|C'}$ denotes the domain restriction of m to C'; and moves $m_1 \in \mathcal{M}_{C_1}$ and $m_2 \in \mathcal{M}_{C_2}$ over disjoint channel sets C_1 and C_2 are combined as $m_1 \cup m_2 \in \mathcal{M}_{C_1 \cup C_2}$.)

Unlike processes, automata have a notion of internal state. This means that two automata that are observationally equivalent, need not be the same. However, we do not wish to distinguish between automata that are observationally equivalent, which motivates the following definitions.

Definition 1.3.3 (Observational equivalence relation)**.** *Consider two automata $\mathsf{a}_1 = (C_I, C_O, S^1, s^1_0, \delta^1_{\mathrm{o}}, \delta^1_{\mathrm{t}})$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s^2_0, \delta^2_{\mathrm{o}}, \delta^2_{\mathrm{t}})$. A relation $R \subseteq S^1 \times S^2$ is said to be an observational equivalence relation (OER) for a_1 and a_2 if and only if, whenever $s_1\ R\ s_2$ the following holds:*

$$\delta^1_{\mathrm{o}}(s_1) = \delta^2_{\mathrm{o}}(s_2) \wedge \forall m \in \mathcal{M}_{C_I}.\delta^1_{\mathrm{t}}(s_1, m)\ R\ \delta^2_{\mathrm{t}}(s_2, m)$$

Definition 1.3.4 (Observational equivalence)**.** *Consider two automata $\mathsf{a}_1 = (C_I, C_O, S^1, s^1_0, \delta^1_{\mathrm{o}}, \delta^1_{\mathrm{t}})$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s^2_0, \delta^2_{\mathrm{o}}, \delta^2_{\mathrm{t}})$. We define the observational equivalence relation*

$$(\cdot \equiv \cdot) \subseteq S^1 \times S^2$$

as the union of all OERs for a_1 and a_2. Observational equivalence is extended to automata, $(\cdot \equiv \cdot) \subseteq \mathfrak{A} \times \mathfrak{A}$, where $\mathsf{a}_1 \equiv \mathsf{a}_2$, whenever $s^1_0 \equiv s^2_0$.

Observational equivalence is defined in a coinductive manner – similar to Milner's notion of *bisimulation* [46]– due to the infinite nature of automata.

Lemma 1.3.5. *Below follows a series of results about OERs.*

(1) The identity relation $R_{\mathrm{id}} \subseteq S \times S$ is an OER for a and a:

$$R_{\mathrm{id}} = \{(s, s) \mid s \in S\}$$

(2) Let $R \subseteq S_1 \times S_2$ be an OER for a_1 and a_2. Then the inverse relation $R^{-1} \subseteq S_2 \times S_1$ is an OER for a_2 and a_1:

$$R^{-1} = \{(s_2, s_1) \mid (s_1, s_2) \in R\}$$

(3) Let $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ be OERs for a_1,a_2 and a_2,a_3. Then the composed relation $R_1 \circ R_2 \subseteq S_1 \times S_3$ is an OER for a_1 and a_3:

$$R_1 \circ R_2 = \{(s_1, s_3) \mid \exists s_2.\ (s_1, s_2) \in R_1 \wedge (s_2, s_3) \in R_2\}$$

Proof. In Appendix B. □

Lemma 1.3.6. *Observational equivalence is a congruence relation on automata, i.e., an equivalence relation (reflexive, transitive and symmetric) and*

$$\forall \mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4 \in \mathfrak{A}. \mathsf{a}_1 \equiv \mathsf{a}_2 \wedge \mathsf{a}_3 \equiv \mathsf{a}_4 \Rightarrow \mathsf{a}_1 \parallel \mathsf{a}_3 \equiv \mathsf{a}_2 \parallel \mathsf{a}_4$$

Proof. In Appendix B. □

We can now show that parallel composition of automata is associative (modulo observational equivalence). Note that we need to require pairwise disjointness of input- and output channels for all three automata, as otherwise parallel composition is in fact not associative – even though it may be well-defined!

Theorem 1.3.7 (Parallel composition is associative)**.** *Consider three automata,* $\mathsf{a}_i = (C_I^i, C_O^i, S^i, s_0^i, \delta_{\mathrm{o}}^i, \delta_{\mathrm{t}}^i) \in \mathfrak{A}$*, for* $i = 1, 2, 3$*, where*

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset$$

then

$$\mathsf{a}_1 \parallel (\mathsf{a}_2 \parallel \mathsf{a}_3) \equiv (\mathsf{a}_1 \parallel \mathsf{a}_2) \parallel \mathsf{a}_3$$

Proof. In Appendix B. □

Commutativity of $\cdot \parallel \cdot$ will be shown in the next section.

1.3.1 Equivalence of Models

We show in this section that the two models of communication – processes and automata – are equivalent. The rest of this chapter does not depend on the material in this section, and the reader may therefore skip this section (and go to Section 1.4).

In order to show equivalence, we show that the models are *isomorphic* when considered *compsets* (Appendix A). The elements of the compsets are processes and automata respectively, and composition is in both cases parallel composition (the partiality is in both cases due to not all processes/automata having compatible channels).

Lemma 1.3.8. $(\mathfrak{P}, \parallel)$ *defines a compset.*

Proof. The fact that $(\cdot \parallel \cdot) : \mathfrak{P} \times \mathfrak{P} \rightharpoonup \mathfrak{P}$ follows from Lemma 1.2.14, the partiality of $\parallel$ is only due to incompatible channels, c.f. Theorem 1.2.15. We note that if $\mathsf{p}_1 \parallel \mathsf{p}_2$ is defined then $\mathsf{p}_2 \parallel \mathsf{p}_1$ is defined, as the condition is the same. Finally assume that $\mathsf{p}_1 \parallel \mathsf{p}_2$ and $\mathsf{p}_1 \parallel \mathsf{p}_3$ and $\mathsf{p}_2 \parallel \mathsf{p}_3$ are defined, which means that:

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset$$

We therefore have that:

$$((C_I^1 \cup C_I^2) \setminus C_{\text{int}}^{1\|2}) \cap C_I^3 = \emptyset$$
$$((C_O^1 \cup C_O^2) \setminus C_{\text{int}}^{1\|2}) \cap C_O^3 = \emptyset$$

and therefore $(\mathsf{p}_1 \parallel \mathsf{p}_2) \parallel \mathsf{p}_3$ is defined. □

Lemma 1.3.9. $(\mathfrak{A}, \|)$ *defines a compset.*

Proof. The fact that $(\cdot \parallel \cdot) : \mathfrak{A} \times \mathfrak{A} \rightharpoonup \mathfrak{A}$ follows by construction (and the partiality of $\parallel$ is only due to incompatible channels). The conditions on the composition follows as for processes (Lemma 1.3.8). □

Automata as Processes

Definition 1.3.10 (Big step relation)**.** *Given an automaton*

$$\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\text{o}}, \delta_{\text{t}})$$

define the big step relation $\boxed{t \vdash s, l_i \Downarrow l_o} \subseteq \mathbb{N} \times S \times \mathcal{L}_{C_I} \times \mathcal{L}_{C_O}$ *by:*

$$\textsf{e-end}\ \frac{\forall t' < \text{eol}(l_i).l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\text{eol}(l_i)}}{t \vdash s, l_i \Downarrow l_\varepsilon}\ (t \geq \text{eol}(l_i))$$

$$\textsf{e-step}\ \frac{\delta_{\text{o}}(s) = m \qquad \delta_{\text{t}}(s, l_i(t)) = s' \qquad t+1 \vdash s', l_i \Downarrow l_o}{t \vdash s, l_i \Downarrow l_o[t \mapsto m]}\ (t < \text{eol}(l_i))$$

(In the above, for a log $l \in \mathcal{L}_C^t$*,* $t' < t$*, and* $m \in \mathcal{M}_C$*;* $l[t' \mapsto m]$ *denotes the log which is identical with* l*, except* $l[t' \mapsto m](t') = m$*.)*

Lemma 1.3.11. *Big step evaluation is total and deterministic, and whenever* $t \vdash s, l_i \Downarrow l_o$*,* $\text{eol}(l_i) = \text{eol}(l_o)$*.*

Proof. Follows immediately from the definition (formally a proof by induction on the derivation). □

Definition 1.3.12 (Automaton translation)**.** *We define the translation*

$$\ulcorner . \urcorner : \mathfrak{A} \to \mathfrak{P}$$

by

$$\ulcorner (C_I, C_O, S, s_0, \delta_o, \delta_t) \urcorner = (C_I, C_O, f)$$

where $f(l_i) = l_o$*, whenever* $0 \vdash s_0, l_i, \Downarrow l_o$*.*

The previous definition gives a process semantics to automata. We must, however, show that the translation is in fact well-defined, i.e., that $\ulcorner \mathsf{a} \urcorner$ defines a process for all automata a.

Theorem 1.3.13 ($\ulcorner . \urcorner$ is well-defined)**.** *The denotation of an automaton* $\mathsf{a} = (C_I, C_O, S, s_0, \delta_o, \delta_t)$ *,* $\ulcorner \mathsf{a} \urcorner = (C_I, C_O, f)$*, is a process.*

Proof. In Appendix B. □

The next two lemmas show that observational equivalence in the automaton model coincides with equality in the process model under the translation $\ulcorner . \urcorner$.

Lemma 1.3.14. *If* $\mathsf{a}_1 \equiv \mathsf{a}_2$ *then* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$*.*

Proof. In Appendix B. □

Lemma 1.3.15. *If* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$ *then* $\mathsf{a}_1 \equiv \mathsf{a}_2$*.*

Proof. In Appendix B. □

The final lemma shows that the translation of automata to processes is compositional, meaning that whenever $\mathsf{a}_1 \parallel \mathsf{a}_2$ is defined, so is $\ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner$, and $\ulcorner \mathsf{a}_1 \parallel \mathsf{a}_2 \urcorner = \ulcorner \mathsf{a}_1 \urcorner \parallel \ulcorner \mathsf{a}_2 \urcorner$.

Lemma 1.3.16 ($\ulcorner . \urcorner$ is homomorphic)**.** $\ulcorner . \urcorner : (\mathfrak{A}, \parallel) \to (\mathfrak{P}, \parallel)$ *is a (compset) homomorphism.*

Proof. In Appendix B. □

Processes as Automata

Having provided a process semantics for automata in the last section, we now show how processes can be given an automaton semantics.

Definition 1.3.17 (Process translation)**.** *We define the translation*

$$\llcorner\cdot\lrcorner : \mathfrak{P} \to \mathfrak{A}$$

by

$$\llcorner(C_I, C_O, f)\lrcorner = (C_I, C_O, \mathcal{L}_{C_I}, \emptyset \in \mathcal{L}^0_{C_I}, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$

where

$$\begin{aligned} \delta_{\mathrm{o}}(l) &= f(l \mathbin{@} m_d)(\mathrm{eol}(l)) \\ \delta_{\mathrm{t}}(l, m) &= l \mathbin{@} m \end{aligned}$$

($\emptyset$ denotes the empty log in $\mathcal{L}^0_{C_I}$; m_d is any "dummy" move of $\mathcal{M}_{C_I}$; and for a log $l \in \mathcal{L}^t_C$ and a move $m \in \mathcal{M}_C$, $l \mathbin{@} m \in \mathcal{L}^{t+1}_C$ behaves like l except $(l \mathbin{@} m)(t) = m$.)

The intuition behind this definition is that the automaton keeps a trace of all that has happened so far (the input log). The output is then determined by applying the log transformer to the input log (extended with some move). The reason why we need to extend the input log is due to the condition (1.2) of log transformers (Definition 1.2.8). Condition (1.3) will then guarantee that *any* extension will produce the same output – and it is exactly the condition that "reaction takes time" that makes it possible to model processes as automata! Note also that we utilize the fact that set of states for an automaton may be infinite, as $\mathcal{L}_{C_I}$ is infinite.

It would now be natural to show compositionality of the translation $\llcorner\cdot\lrcorner$, similar to Lemma 1.3.16. However, we need not show this directly, as the result will follow automatically from the theory of compsets and the results about the two translations following in the next section.

Equivalence

Lemma 1.3.18 ($\ulcorner\cdot\urcorner$ is the left inverse of $\llcorner\cdot\lrcorner$)**.** $\forall \mathsf{p} \in \mathfrak{P}.\ulcorner\llcorner\mathsf{p}\lrcorner\urcorner = \mathsf{p}$

Proof. In Appendix B. □

It does not hold that $\llcorner\cdot\lrcorner$ is the left inverse of $\ulcorner\cdot\urcorner$, simply because $\ulcorner\cdot\urcorner$ is not injective. However, when we consider automata modulo observational equivalence then $\ulcorner\cdot\urcorner$ becomes injective.

Theorem 1.3.19. $\ulcorner\cdot\urcorner : (\mathfrak{A}/\equiv, \|) \to (\mathfrak{P}, \|)$ *defined by*

$$\ulcorner[a]\urcorner = \ulcorner a \urcorner$$

is an isomorphism.

Proof. We know from Lemma 1.3.16 that $\ulcorner\cdot\urcorner : (\mathfrak{A}, \|) \to (\mathfrak{P}, \|)$ is a homomorphism. Furthermore this homomorphism is surjective by Lemma 1.3.18 since

$$\forall \mathsf{p} \in \mathfrak{P}.\ulcorner\llcorner \mathsf{p} \lrcorner\urcorner = \mathsf{p}$$

By Lemma 1.3.14 and Lemma 1.3.15 it follows that

$$\mathsf{a}_1 \equiv \mathsf{a}_2 \Leftrightarrow \ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$$

hence by Lemma A.0.6 ($\equiv$ is a congruence relation on $\mathfrak{A}$) the result follows. □

We now get "for free" that $\llcorner\cdot\lrcorner$ is compositional, and the inverse of $\ulcorner\cdot\urcorner$.

Corollary 1.3.20 (Equivalence of models)**.** $\llcorner\cdot\lrcorner : (\mathfrak{P}, \|) \to (\mathfrak{A}/\equiv, \|)$ *is an isomorphism with inverse isomorphism* $\ulcorner\cdot\urcorner$*, i.e.,*

$$\llcorner\cdot\lrcorner : (\mathfrak{P}, \|) \simeq (\mathfrak{A}/\equiv, \|) : \ulcorner\cdot\urcorner$$

Proof. By Theorem 1.3.19 we know that $\ulcorner\cdot\urcorner : (\mathfrak{A}/\equiv, \|) \to (\mathfrak{P}, \|)$ is an isomorphism. We also know from Lemma 1.3.18 that $\forall \mathsf{p} \in \mathfrak{P}.\ulcorner\llcorner \mathsf{p} \lrcorner\urcorner = \mathsf{p}$ and hence by Lemma A.0.4 (b) that $\llcorner\cdot\lrcorner$ is the inverse of $\ulcorner\cdot\urcorner$. Finally Lemma A.0.4 (a) implies that $\llcorner\cdot\lrcorner$ is itself an isomorphism as required. □

Having the equivalence result of Corollary 1.3.20 means that we can automatically transfer the results from the process model to the automaton model and vice versa.

Corollary 1.3.21 (Parallel composition of processes is associative)**.** *Let* $\mathsf{p}_i = (C_I^i, C_O^i, f^i) \in \mathfrak{P}$ *be processes for* $i = 1, 2, 3$*, such that*

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset$$

then

$$\mathsf{p}_1 \parallel (\mathsf{p}_2 \parallel \mathsf{p}_3) = (\mathsf{p}_1 \parallel \mathsf{p}_2) \parallel \mathsf{p}_3$$

Proof. Follows from Lemma A.0.4 (c) and Theorem 1.3.7. □

Corollary 1.3.22 (Parallel composition of automata is commutative)**.** *For all* $\mathsf{a}_1, \mathsf{a}_2 \in \mathfrak{A}.\mathsf{a}_1 \parallel \mathsf{a}_2 \equiv \mathsf{a}_2 \parallel \mathsf{a}_1$.

Proof. Follows from Lemma A.0.4 (d) and Theorem 1.2.16. We note that this proposition would be fairly easy to prove directly - but this proof illustrates that the equivalence works in both directions. □

We now have a simple, extensional model of communication which is equivalent to the original model proposed in Section 1.2. The next sections will use this model to (finally) introduce *contracts*.

1.4 Principals & Contracts

In the previous sections we spent quite some time on defining the model of communication. In this section we can therefore proceed with the original goal, namely the extension of PBC to distributed and concurrent systems. We introduce the formal notions of *principals* and *contracts*. Principals can be thought of as administrative parties, for instance a person or an organization, who acts as an administrative peer in the distributed environment.

Definition 1.4.1 (Principal)**.** *The set of all principals is denoted $\mathcal{P}$. We write $\mathsf{P}_1, \mathsf{P}_2, \ldots$ for single principals.*

Two principals can negotiate a contract, which is an abstraction for *communication obligations*. In order to capture different kinds of communication, we define *logical communication links*. A logical communication link specifies the type of messages to be communicated (*actions*), and is hence almost identical with channels (Definition 1.2.1).

Definition 1.4.2 (Logical communication link)**.** *A logical communication link is written λ and sets of links are denoted Λ. A logical communication link is directed and always between two principals. For each logical communication link λ, there is a corresponding set of actions communicated on that link, denoted $\mathcal{A}_\lambda$. Each action set has a distinguished silent action ε.*

In the context of distributed computing, a logical link will typically have as action set the set of all IP packets (between two pre-specified IP addresses). But logical links can also be used for modeling *real world* events. For instance in a commercial setting, a logical communication link could be used for modeling shipment of goods, payments from one principal to another, etc. In all cases, a logical link from principal P_1 to principal P_2 means that P_1 has some "communication" obligations to P_2.

The reason why we have emphasized links as being logical is that the principals at each end of a link need not be the actual physical sender or receiver for that link. The sender (and receiver respectively) of a logical link has the opportunity of being the physical sender (receiver), but it may pass on this opportunity to another principal. The term logical hence means that the principals have committed to some actions on the links in the contract, but they may not be in control of the underlying physical communication (which is what differentiates logical links from channels, c.f. Section 1.2). In some cases it may not even be possible for a principal to perform the communication itself; if for instance the action set specifies shipment of goods to a client in Australia, and the principal is located in Denmark, then the principal will – most likely – have to delegate the "communication" to a shipper.

Logical links are always associated with exactly one contract. In order to specify contracts we first (re)introduce *moves*. The definition is identical to Definition 1.2.2, except channels are replaced by logical links.

Definition 1.4.3 (move over Λ)**.** *For a finite set of logical communication links, Λ, we define a move over Λ to be a function m, such that $m(\lambda) \in \mathcal{A}_\lambda$ for all $\lambda \in \Lambda$. The set of all moves over Λ is denoted $\mathcal{M}_\Lambda$ (i.e., $\mathcal{M}_\Lambda = \Pi_{\lambda\in\Lambda}\mathcal{A}_\lambda$).*

We are now ready to present contracts:

Definition 1.4.4 (Contract)**.** *A contract c between principals P and A (player and adversary) is a 5-tuple:*

$$c = (\Lambda_{\mathsf{PA}}, \Lambda_{\mathsf{AP}}, G, g_0, \rho)$$

where Λ_{PA} and Λ_{AP} are the finite sets of logical links from principal P to principal A and vice versa, G is the (potentially infinite) set of contract states, and $g_0 \in G$ is the start state. $\rho : G \times \mathcal{M}_{\Lambda_{\mathsf{PA}}} \times \mathcal{M}_{\Lambda_{\mathsf{AP}}} \to G \times \mathbb{Q}$ is the rule function for the game: when the contract is in a state g, and the moves on Λ_{PA} and Λ_{AP} are m_{P} and m_{A}, let $(g', k) = \rho(g, m_{\mathsf{P}}, m_{\mathsf{A}})$; then g' is the new contract state, and k is the – possibly negative – incremental payoff to P from A, resulting from the transition.

A contract, as defined above, evolves in each time unit, based on the chosen moves (m_{P} and m_{A}) of the two players. A move m may consist of simply "doing nothing", in which case $m(\lambda) = \varepsilon$ for all λ. Time units are assumed to be small and fixed; general timing constraints are expressed by means of explicit counters in the game state (which we will see an example of soon). Note that there are no "illegal" moves per se: moves in violation of the nominal game rules will typically be assigned a large negative payoff, but the contract remains in a well-defined state, to guide an orderly recovery.

Observation 1.4.5. A contract $(\Lambda_{\mathsf{PA}}, \Lambda_{\mathsf{AP}}, G, g_0, \rho)$ describes an *infinite, simultaneous, zero-sum, two-person game* between principals P and A. Finite contracts can be modeled by introducing a terminal state g_t and extend ρ such that

$$\rho(g_t, m_{\mathsf{P}}, m_{\mathsf{A}}) = (g_t, 0)$$

for all $(m_{\mathsf{P}}, m_{\mathsf{A}}) \in \mathcal{M}_{\Lambda_{\mathsf{PA}}} \times \mathcal{M}_{\Lambda_{\mathsf{AP}}}$.

A contract is always defined from the viewpoint of one principal (in the defining case P). But it is also possible to define the *dual* contract from the viewpoint of A (by swapping the links and negating payoff), but we will

not use dual contracts in this chapter, and do therefore not give a formal definition.

In order to model situations with more than two principals, principals can in general negotiate a (finite) set of contracts, called a *contract portfolio*.

Definition 1.4.6 (Contract portfolio)**.** *A contract portfolio for a principal* P *is a finite set of contracts* $C = \{c_1, \ldots, c_n\}$ *where* $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$ *for* $i = 1, \ldots, n$.

Contract portfolios make it possible to have multi-party contracts, by means of bilateral contracts only. This approach is radically different from the commonly used global approach to multi-party contracts, represented typically as sequence diagrams [63]. For instance in the context of web services, the global approach is manifested in the Web Services Choreography Description Language (WSCDL) [66]. Our approach is illustrated in the following example.

Example 1.4.7 (MMS greeting service)**.** In the following example we describe how a real-world contract can be represented in the model introduced so far. The example serves two purposes: first, we get a graphical representation of principals and contracts for easier overview; second, we show how this model distinguishes itself from other models with multiple parties in that we require all contracts be bilateral.

The example we consider includes three principals (Figure 1.1): a service provider (P, from the viewpoint of whom we are considering), a subcontractor (S), and a client (C).

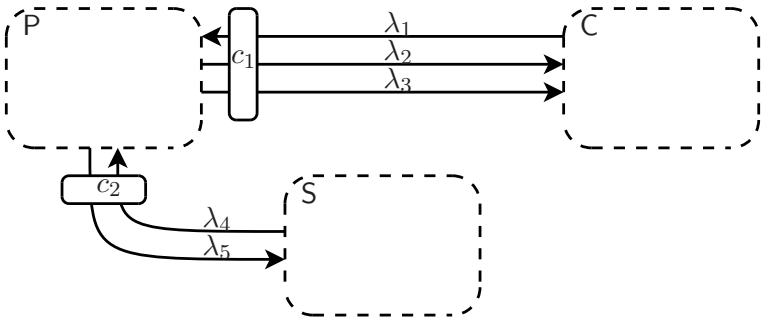


Figure 1.1: Graphical representation of three principals with bilateral contracts.

The overall idea is that P offers a mobile-phone greeting service, which enables client C to send one (amongst a set of) predefined greeting card MMS to a specified phone number. To send the actual MMS, the service provider has subcontracted with an MMS gateway provider (S), who provides the

service of sending MMSs with arbitrary content. The traditional way to describe such a situation is by means of a global choreography, Figure 1.2, in which all principals are present (multi-party).

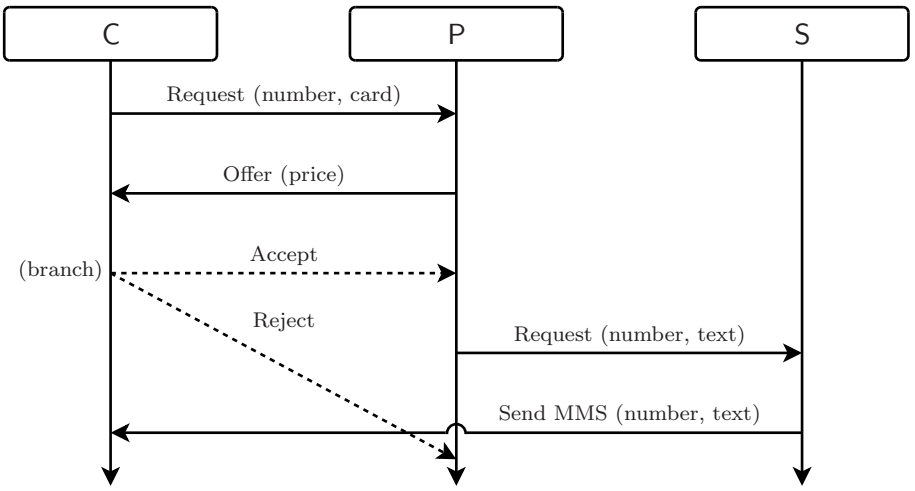


Figure 1.2: A global choreography for principals P, C, and S.

The global choreography gives a good intuition of what is going on, however, in our opinion there are several shortcomings with this approach (if used as a contract): as presented in the figure, all principals need be aware of each other when signing the contract. This means that both P and C must know S, which in most cases is not desirable; often C wants to leave it to P to decide which MMS gateway to use (in fact C will probably think that P *is* an MMS gateway). Hence at the principal-level, the model above is not compositional.

Another – and perhaps more severe – problem with the global approach is *blame ambiguity*. It is not evident from a global workflow description who is to blame if the choreography is not followed. In some sense the choreography represents the "happy-path" only, i.e., a scenario in which the principals are expected to cooperate towards fulfilling the workflow. But in a setting with different principals with individual goals, cooperation is almost self-contradictory.

So instead we show how P negotiates separate contracts with both C (c_1) and S (c_2). Informally, P's contract with C says:

> C *can request the price of sending greeting card* g *to phone number* n*, and* P *can reply with a price* p*. Subsequently* C *can accept or reject. If* C *accepts,* P *has to send the MMS before at most* t *time units. If* P *fails to do so,* P *is assigned a penalty of* 1.

As specified in the contract, P is the one responsible for sending the MMS, and the contract has no mention of S. Payoffs model what *should* be paid from C to P, not what has been paid. Actions are used to model real world events, for instance delivery of the MMS, communication between P and C, etc. The contract with C contains three logical links:

$$c_1 = (\{\lambda_2, \lambda_3\}, \{\lambda_1\}, G_1, g_1, \rho_1)$$

λ_1 and λ_2 are used for communication between P and C and λ_3 is used for the special "communication" of sending an MMS. The fact that λ_3 is directed from P to C should not be interpreted that an MMS has to be sent from P to C; it means that P is responsible to C for sending an MMS.

Formally we therefore have (omitting the silent actions, letting $\mathcal{G}$ denote the set of all greeting cards, and letting $\mathcal{N}$ denote the set of phone numbers):

$$\begin{aligned}
\mathcal{A}_{\lambda_1} &= \{\text{accept}, \text{reject}\} \cup \{\text{req}(n, g) \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\} \\
\mathcal{A}_{\lambda_2} &= \{\text{price}(p) \mid p \in \mathbb{Q}\} \\
\mathcal{A}_{\lambda_3} &= \{\text{mms}(n, g) \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\}
\end{aligned}$$

The formalized contract is presented graphically in Figure 1.3 (left). Contract states are depicted as circles, and the double-circled state is a terminal state, c.f. Observation 1.4.5. An arrow from g_1 to g_2 with label $\lambda : a$ and a boxed $\boxed{k}$ means that $\rho(g_1, m_1, m_2) = (g_2, k)$, whenever $m_1(\lambda) = a$ or $m_2(\lambda) = a$ (depending on which of the principals is the sender on λ). When no box is present on a transition, it means an implicit $\boxed{0}$ (i.e., no payoff). An arrow from g_1 to g_2 with no label and (implicit) $\boxed{k}$ means that $\rho(g_1, m_1, m_2) = (g_2, k)$, whenever no other label matches m_1 and m_2. If there is no explicit unlabeled arrow from a state g, there is an implicit unlabeled arrow from g to itself.

In the diagram some of the states have *internal state*, e.g., $\langle n, g \rangle$. This means that the node actually represents a *class* of states – potentially one for each combination of n and g. We therefore have

$$\begin{aligned}
G_1 = \{\star, \dagger\} &\cup \{\langle n, g \rangle \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\} \cup \\
&\{\langle n, g, p \rangle \mid n \in \mathcal{N} \wedge g \in \mathcal{G} \wedge p \in \mathbb{Q}\} \cup \\
&\{\langle n, g, p, \tau \rangle \mid n \in \mathcal{N} \wedge g \in \mathcal{G} \wedge p \in \mathbb{Q} \wedge 1 \leq \tau \leq t\} \\
g_1 = \star &
\end{aligned}$$

The rule function can be read from the diagram (according to the description above) and be presented in following *tableau form*:

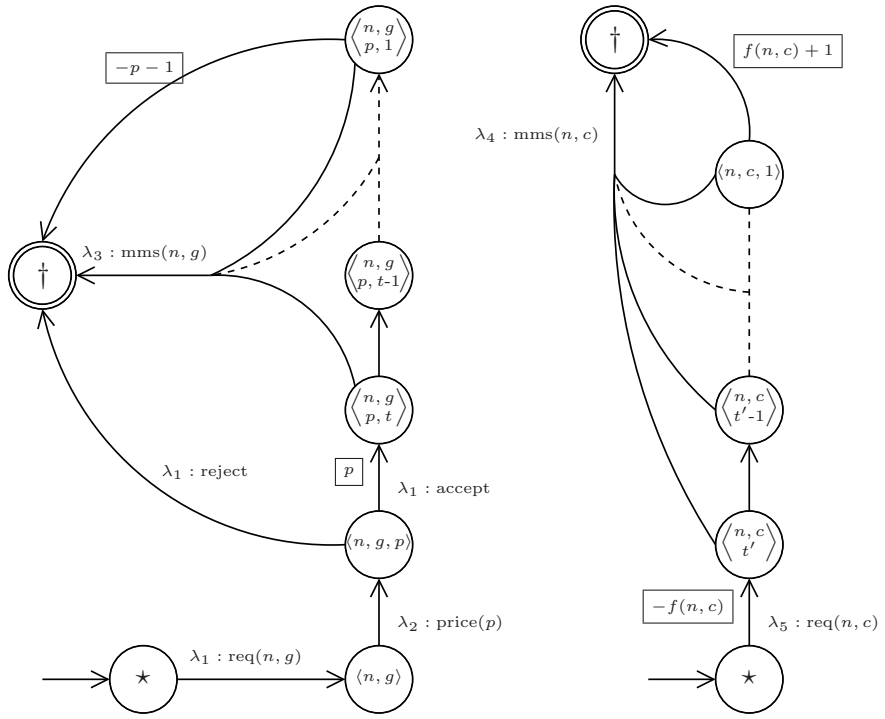


Figure 1.3: The contracts negotiated by P with C (left) and S (right).

g	m_{P}	m_{C}	$\rho_1(g, m_{\mathsf{P}}, m_{\mathsf{C}})$
$\star$	–	$\lambda_1 \mapsto \text{req}(n, g)$	$(\langle n, g \rangle, 0)$
$\star$	–	–	$(\star, 0)$
$\langle n, g \rangle$	$\lambda_2 \mapsto \text{price}(p)$	–	$(\langle n, g, p \rangle, 0)$
$\langle n, g \rangle$	–	–	$(\langle n, g \rangle, 0)$
$\langle n, g, p \rangle$	–	$\lambda_1 \mapsto \text{reject}$	$(\dagger, 0)$
$\langle n, g, p \rangle$	–	$\lambda_1 \mapsto \text{accept}$	$(\langle n, g, p, t \rangle, p)$
$\langle n, g, p, \tau \rangle$	$\lambda_3 \mapsto \text{mms}(n, g)$	–	$(\dagger, 0)$
$\langle n, g, p, 1 \rangle$	–	–	$(\dagger, -p-1)$
$\langle n, g, p, \tau + 1 \rangle$	–	–	$(\langle n, g, p, \tau \rangle, 0)$
$\dagger$	–	–	$(\dagger, 0)$

The tableau form should be read from top to bottom, i.e., given a state g and moves m_{P} and m_{C}, then $\rho^1(g, m_{\mathsf{P}}, m_{\mathsf{C}})$ is determined by the first line matching g, m_{P}, and m_{C} (i.e., pattern matching as known from functional programming languages, where "–" matches any move).

Principal P's contract with the MMS gateway (S) says:

> P *can request an MMS to phone number* n *with content* c *at a price* $f(n, c)$ *(for some predefined rate function* f*). Subsequently*

> S *must send the MMS before* t' *time units. If* S *fails to do so,* S *is assigned a penalty of* 1.

The contract c_2 is presented in Figure 1.3 (right), where

$$\begin{aligned} c_2 &= (\{\lambda_5\}, \{\lambda_4\}, G_2, g_2, \rho_2) \\ G_2 &= \{\star, \dagger\} \cup \{\langle n, c, \tau\rangle \mid n \in \mathcal{N} \wedge c \in \mathcal{C} \wedge 1 \leq \tau \leq t'\} \\ g_2 &= \star \\ \mathcal{A}_{\lambda_4} &= \{\mathrm{mms}(n, c) \mid n \in \mathcal{N} \wedge c \in \mathcal{C}\} \\ \mathcal{A}_{\lambda_5} &= \{\mathrm{req}(n, c) \mid n \in \mathcal{N} \wedge c \in \mathcal{C}\} \end{aligned}$$

(again omitting the implicit silent actions). $\mathcal{C}$ represents the set of all possible MMS content (which in particular contains the predefined greeting cards provided by P, i.e., $\mathcal{G} \subseteq \mathcal{C}$). The rule function can again be read from the diagram:

g	m_{P}	m_{S}	$\rho_2(g, m_{\mathsf{P}}, m_{\mathsf{S}})$
$\star$	$\lambda_5 \mapsto \mathrm{req}(n, c)$	$-$	$(\langle n, c, t'\rangle, -f(n, c))$
$\star$	$-$	$-$	$(\star, 0)$
$\langle n, c, \tau\rangle$	$-$	$\lambda_4 \mapsto \mathrm{mms}(n, c)$	$(\dagger, 0)$
$\langle n, c, 1\rangle$	$-$	$-$	$(\dagger, f(n, c) + 1)$
$\langle n, c, \tau + 1\rangle$	$-$	$-$	$(\langle n, c, \tau\rangle, 0)$
$\dagger$	$-$	$-$	$(\dagger, 0)$

Now P has negotiated contracts with principals C and S. But the contracts are not "active" yet, since a physical realization of the logical links has to be established. Hence P has to construct a physical implementation for fulfilling its contract portfolio $\{c_1, c_2\}$. The contract with S bears no obligations, hence this contract is easy to fulfill, but in the contract c_1 with C, P has the obligation to deliver the greeting card MMS (at least if P intends to make money – otherwise denying to answer with a price will do). Implementations is the subject of the next section, where we return to this example.

In Section 1.6 we show how the comprehensive contract diagrams in Figure 1.3 can be represented more compactly by means of a small contract language. This language includes *timeouts* as a primitive, hence we will not have to be bothered with manual counters as in Example 1.4.7. The example with manual counters has been included, however, to show how the core contract model works; but it is not the intention that contracts should be written directly in the abstract model.

1.5 Implementations & Conformance

The last section introduced principals and contracts between them. In this section we describe how contracts are realized physically by means of *implementations*. An implementation defines a strategy for playing the games specified by a contract portfolio. A strategy will consist of a set of automata (or equivalently processes, c.f. Section 1.3.1) together with a mapping of logical links in the contract portfolio to channels in the automata. However, a strategy will also consist of an ability to *delegate* a logical obligation to another principal of the contract portfolio. In Example 1.4.7 from Section 1.4, such a delegation will be used for fulfilling the commitment to send an MMS, as the service provider (P) is not able implement an automaton with alphabet $\mathcal{A}_{\lambda_3}$ (i.e., P does not have the hardware for sending the MMS).

Rather than defining an implementation as a set of automata (as mentioned above), we consider only the case in which an implementation consists of a single automaton; this simplification is justified by the fact that multiple, parallel automata (which may interact by unobservable communications on internal links) can be described by a single automaton (cf. Section 1.3).

Hence the first part of an implementation is an automaton. We now define *routings*, which constitute the second part of an implementation:

Definition 1.5.1 (Routing)**.** *Let $\{c_1, \ldots, c_n\}$ be a contract portfolio for a principal P, where $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$. A routing $r = (r_1, r_2, r_3)$ for $\{c_1, \ldots, c_n\}$ and input/output channels C_I/C_O consists of three maps:*

$$
\begin{aligned}
&r_1 : C_I \to \Lambda_I \\
&r_2 : C_O \to \Lambda_O \\
&r_3 : \Lambda_I \setminus r_1(C_I) \to \Lambda_O \setminus r_2(C_O)
\end{aligned}
$$

where $\Lambda_I = \bigcup_{i=1}^n \Lambda_{\mathsf{A}_i\mathsf{P}}$ and $\Lambda_O = \bigcup_{i=1}^n \Lambda_{\mathsf{PA}_i}$. r_1 and r_2 realize physical communication by using the input/output channels C_I/C_O, while r_3 represents delegation to other principals. The routing must satisfy the following conditions:

(1) r_1, r_2 are injective.

(2) r_3 is bijective.

(3) $\forall i \in \{1, 2, 3\}.\ \forall \chi.\ \mathcal{A}_\chi = \mathcal{A}_{r_i(\chi)}$.

(1) and (2) state that all input/output channels represent (exactly) one logical link of the portfolio, and (3) guarantees that only compatible channels/links are connected, i.e., the link alphabet must match the channel alphabet.

Intuitively, a routing specifies which obligations of the contract portfolio are handled by the principal itself, and which parts are delegated to other principals in the portfolio. The idea is that the principal can choose the routing first, and then construct an automaton once the non-delegated obligations are known:

Definition 1.5.2 (Implementation)**.** *Let* $C = \{c_1, \ldots, c_n\}$ *be a contract portfolio for a principal* P*. An implementation,* $i = (\mathsf{a}, r)$*, consists of an automaton* $\mathsf{a} = (C_I, C_O, S, s_o, \delta_\mathrm{o}, \delta_\mathrm{t})$*, and a routing,* r*, for* C *and* C_I/C_O*.*

In order to make the abstract definitions above more clear, we return to Example 1.4.7 from Section 1.4:

Example 1.5.3 (MMS greeting service, cont'd. from Example 1.4.7)**.** An example of a legal implementation for the service provider (P) is $i = (\mathsf{a}, r)$, where

$$\mathsf{a} = (\{\alpha\}, \{\beta, \gamma\}, S, s_0, \delta_\mathrm{o}, \delta_\mathrm{t})$$

and $r = (r_1, r_2, r_3)$ is defined by

$$\begin{aligned} r_1(\alpha) &= \lambda_1 \\ r_2(\beta) &= \lambda_5 \\ r_2(\gamma) &= \lambda_2 \\ r_3(\lambda_4) &= \lambda_3 \end{aligned}$$

The implementation is depicted in Figure 1.4, in which the implementation part is drawn inside P. We will shortly return to what a "good" choice of a is.

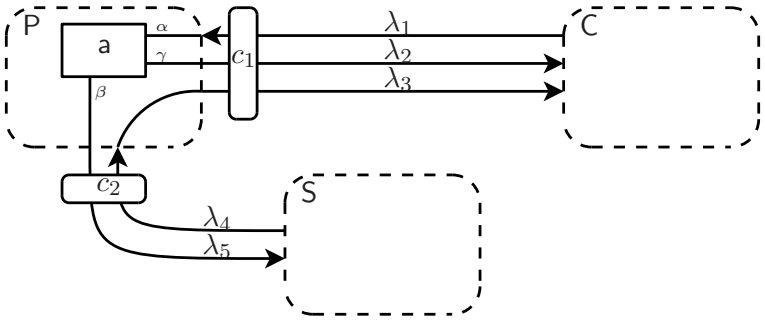


Figure 1.4: An implementation at principal P.

1.5.1 Contract Conformance

We now wish to define what it means for an implementation to "satisfy" a contract portfolio (contract conformance). Since contracts do not have binary outcome (win/lose), contract conformance should be an economic guarantee related to the payoffs in the contract portfolio. We therefore define contract conformance as a guarantee of an all-time non-negative accumulated payoff, meaning that the implementation may give profit – but definitely no loss.

If for instance we want to know whether an implementation will provide a certain profit, p, after a period of time, t, then this can be achieved by extending the contract portfolio with a *pseudo contract*, which yields a payoff of $-p$ after t time units. If the implementation conforms with the extended portfolio, then a profit of at least p will be guaranteed after t time units. However, we cannot provide guarantees such as "the implementation will provide a profit of p *eventually*", since contract conformance defines a safety property; but from an economic point of view, such a guarantee is not very useful anyway.

Before we define contract conformance, we introduce the following auxiliary definition:

Definition 1.5.4. *Consider a routing r for input/output channels C_I/C_O and contract portfolio $C = \{c_1, \ldots, c_n\}$, where*

$$r = (r_1, r_2, r_3)$$
$$c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$$

and let $\Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$ be the set of all incoming links from the contracts. We then define for each $i = 1, \ldots, n$ the function $\overline{r}_i : \mathcal{M}_{C_O} \times \mathcal{M}_{\Lambda_I} \to \mathcal{M}_{\Lambda_{\mathsf{PA}_i}}$ by

$$\overline{r}_i(m_{C_O}, m_{\Lambda_I})(\lambda) = \begin{cases} m_{C_O}(\alpha) & , \text{ if } r_2(\alpha) = \lambda \\ m_{\Lambda_I}(\lambda') & , \text{ if } r_3(\lambda') = \lambda \end{cases}$$

The definition above captures the idea that some of the output obligations of the contract portfolio may be handled by the automaton (the first case) while others may be delegated (the second case). Hence if opponent A_i has moved m_i and the automaton has produced output m, then P's move in contract c_i is $\overline{r}_i(\bigcup_{j=1}^{n} m_j, m)$.

Contract conformance can now be defined in a coinductive manner – as observational equivalence for automata – due to the infinite nature of both contracts and processes:

Definition 1.5.5 (Contract conformance)**.** *Consider an implementation* $i = (\mathsf{a}, r)$ *for the contract portfolio* $C = \{c_1, \ldots, c_n\}$*, where*

$$\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$
$$r = (r_1, r_2, r_3)$$
$$c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$$

and let $\Lambda_I = \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}}$ *be the set of all incoming links from the contracts. A relation* $R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$ *is said to be a conformance relation for* i *and* C*, if and only if, for all* $(k, s, g_1, \ldots, g_n) \in R$ *the following holds:*

$$\begin{aligned}
&\forall m \in \mathcal{M}_{\Lambda_I}.\\
&\quad (\forall i \in \{1, \ldots, n\}.\, (g_i', k_i) = \rho_i(g_i, \overline{r}_i(\delta_{\mathrm{o}}(s), m), m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}})) \Rightarrow\\
&\qquad \sum_{i=1}^{n} k_i \geq k \wedge (k - \sum_{i=1}^{n} k_i, \delta_{\mathrm{t}}(s, m \circ r_1), g_1', \ldots, g_n') \in R
\end{aligned}$$

The implementation i *is said to conform with contract portfolio* C*, written* $\models i : C$*, if there exists a conformance relation* R *for* i *and* C*, such that* $(0, s_0, g_1, \ldots, g_n) \in R$*.*

This definition of $\models i : C$ formalizes the guarantee of a consistently non-negative accumulated payoff, when using the strategy i for playing the games induced by the portfolio $\{c_1, \ldots, c_n\}$. More generally, whenever a is in state s, contract c_i is in state g_i, and $(k, s, g_1, \ldots, g_n) \in R$, where R is a conformance relation for i and C, then the accumulated payoff will remain at least k throughout the remainder of the game. Note also how delegation is handled by the auxiliary functions $\overline{r}_i$ from Definition 1.5.4.

1.5.2 Automaton Contracts

The definition of contract conformance from last subsection allows us to reason about implementations, when we know all the (logical) contracts that have been negotiated with other principals, and the delegation that has been chosen. However, we will like to reason about automata without having to worry about delegation and principals directly. For this purpose we now introduce *automaton contracts*:

Definition 1.5.6 (Automaton contract)**.** *An automaton contract* c *is a 4-tuple:*

$$c = (C, G, g_0, \rho)$$

where C *is a finite set of channels,* G *is the (potentially infinite) set of contract states, and* $g_0 \in G$ *is the start state.* $\rho : G \times \mathcal{M}_C \to G \times \mathbb{Q}$ *is the rule function for the game: when the contract is in a state* g*, and the move on* C *is* m*, let* $(g', k) = \rho(g, m)$*; then* g' *is the new contract state, and* k *is the – possibly negative – incremental payoff.*

The definition of automaton contracts is quite similar to the original definition of (logical) contracts. The similarity is due to the fact that the automaton contracts can be seen as instances of the original contracts with the logical links renamed to physical channels. This also means that the physical channels of the game need not be contained in the channels of the automaton, which is the reason why automaton contracts are not defined with respect to a particular automaton.

Automaton contract conformance can now be defined in a similar manner as for contracts:

Definition 1.5.7 (Automaton contract conformance)**.** *Consider an automaton* $\mathsf{a} = (C_I, C_O, S, s_0, \delta_\mathrm{o}, \delta_\mathrm{t})$ *and let* $\mathsf{C} = \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ *a set of automaton contracts, where*

$$\mathsf{c}_i = (C_i, G_i, g_i, \rho_i)$$

and let $C = (C_I \cup \bigcup_{i=1}^{n} C_i) \setminus C_O$ *be the set of all incoming/external channels. A relation* $R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$ *is said to be a conformance relation for* a *and* C*, if and only if, for all* $(k, s, g_1, \ldots, g_n) \in R$ *the following holds:*

$$\begin{aligned}
&\forall m \in \mathcal{M}_C.\\
&\quad (\forall i \in \{1, \ldots, n\}.\, (g_i', k_i) = \rho_i(g_i, (m \cup \delta_\mathrm{o}(s))_{|C_i})) \Rightarrow\\
&\qquad \sum_{i=1}^{n} k_i \geq k \wedge (k - \sum_{i=1}^{n} k_i, \delta_\mathrm{t}(s, m_{|C_I}), g_1', \ldots, g_n') \in R
\end{aligned}$$

The automaton a *is said to conform with contract portfolio* C*, written* $\models \mathsf{a} : \mathsf{C}$*, if there exists an automaton conformance relation* R *for* a *and* C*, such that* $(0, s_0, g_1, \ldots, g_n) \in R$*.*

Before we show how logical contracts er transformed to automaton contracts, we provide an example illustrating automaton contracts, and automaton contract conformance:

Example 1.5.8 (MMS greeting service, cont'd. from Example 1.5.3)**.** Consider the two automaton contracts $\mathsf{c}_1 = (\{\alpha, \gamma, \delta\}, G_1, g_1, \rho_1')$ and $\mathsf{c}_2 = (\{\beta, \delta\}, G_2, g_2, \rho_2')$ obtained by replacing the logical links of the logical contracts (p. 28) with the following channels

$$\begin{array}{ll} \lambda_1 \mapsto \alpha & \lambda_3, \lambda_4 \mapsto \delta \\ \lambda_2 \mapsto \gamma & \lambda_5 \mapsto \beta \end{array} \tag{1.6}$$

The definitions of G_1, g_1, G_2, and g_2 are identical with those of Example 1.4.7, and ρ_1' is obtained by replacing the links of ρ_1 with the channels above, i.e., $\rho_1'(g, m) = \rho_1(g, m \circ \theta_{|\{\lambda_1\}}, m \circ \theta_{|\{\lambda_2, \lambda_3\}})$, where θ is the substitution (1.6) above (and similar for ρ_2').

The automaton $\mathsf{a} = (\{\alpha\}, \{\beta, \gamma\}, S, s_o, \delta_\mathrm{o}, \delta_\mathrm{t})$ is defined by:

$$\begin{aligned} S = {} & \{\textbf{start,end}\} \cup \\ & \{\textbf{req\_received}(n, g) \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\} \cup \\ & \{\textbf{price\_offered}(n, g) \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\} \\ & \{\textbf{accepted}(n, g) \mid n \in \mathcal{N} \wedge g \in \mathcal{G}\} \\ s_0 = {} & \textbf{start} \end{aligned}$$

The transition functions are represented in the following tableau forms

s	$\delta_\mathrm{o}(s)$
start	
req_received(n, g)	$\gamma \mapsto \mathrm{price}(1.5 * f(n, g))$
price_offered(n, g)	
accepted(n, g)	$\beta \mapsto \mathrm{req}(n, g)$
end	

s	m	$\delta_\mathrm{t}(s, m)$
start	$\alpha \mapsto \mathrm{req}(n, g)$	**req_received**(n, g)
start	–	**start**
req_received(n, g)	–	**price_offered**(n, g)
price_offered(n, g)	$\alpha \mapsto$ reject	**end**
price_offered(n, g)	$\alpha \mapsto$ accept	**accepted**(n, g)
price_offered(n, g)	–	**price_offered**(n, g)
accepted(n, g)	–	**end**
end	–	**end**

Note that a has nothing to do with actual sending of the MMS, it only contacts S on channel β to request the MMS, and directly thereafter stops (enters the **end** state). This is witnessed by the channel δ, which is not connected to the automaton. δ is an abstraction of the fact that S and C "communicate" directly without P's automaton being able to react to their communication, hence a should be able to conform with the contract no matter what happens on δ (which is exactly what the definition of contract conformance guarantees). Figure 1.5 gives an informal graphical representation of the automaton.

So can we show that this automaton conforms with c_1 and c_2? In order to do so we need to find a conformance relation $R \subseteq \mathbb{Q} \times S \times G_1 \times G_2$ containing $(0, \textbf{start}, \star, \star)$. And in fact such a conformance relation does not always exist, but if we assume that $t' < t - 1$, i.e., that S guarantees to send the MMS *before* $t-1$ time units (t being the guarantee negotiated with the client C), then a conformance relation does exist (the extra one unit is needed for the automaton to propagate the request to the MMS gateway). One such relation is given below, where we introduce the following shorthand notation: $p_{n,g} = 1.5 * f(n, g)$ (the price suggested by P), and $r_{n,g} = 0.5 * f(n, g)$ (P's revenue if there is no timeout). In each set below, whenever n and g are mentioned, there is an implicit condition that $n \in \mathcal{N}$ and $g \in \mathcal{G}$ – we leave these out for readability:

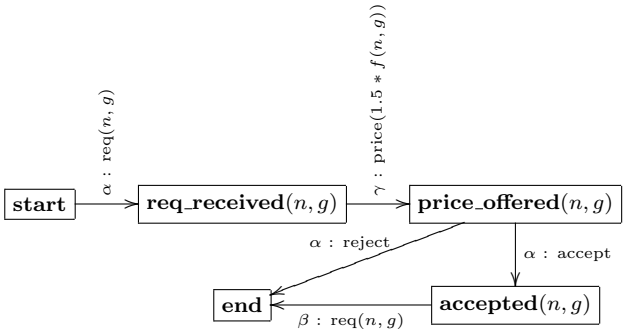
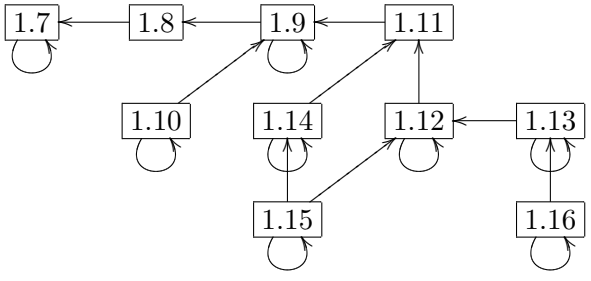


Figure 1.5: A graphical representation of the automaton a deployed by P.

$$
\begin{aligned}
R = {} & \{(0, \mathbf{start}, \star, \star)\} && (1.7)\\
& \cup \{(0, \mathbf{req\_received}(n,g), \langle n, g\rangle, \star)\} && (1.8)\\
& \cup \{(0, \mathbf{price\_offered}(n,g), \langle n, g, p_{n,g}\rangle, \star)\} && (1.9)\\
& \cup \{(0, \mathbf{end}, \dagger, \star)\} && (1.10)\\
& \cup \{(-p_{n,g}, \mathbf{accepted}(n,g), \langle n, g, p_{n,g}, t\rangle, \star)\} && (1.11)\\
& \cup \{(-r_{n,g}, \mathbf{end}, \langle n, g, p_{n,g}, t-k-1\rangle, \langle n, g, t'-k\rangle) \mid 0 \le k < t'\} && (1.12)\\
& \cup \{(-p_{n,g}-1, \mathbf{end}, \langle n, g, p_{n,g}, t-t'-k-1\rangle, \dagger) \mid 0 \le k < t-t'-1\} && (1.13)\\
& \cup \{(-r_{n,g}, \mathbf{end}, \dagger, \langle n, g, t'-k\rangle) \mid 0 \le k < t'\} && (1.14)\\
& \cup \{(-r_{n,g}, \mathbf{end}, \dagger, \dagger)\} && (1.15)\\
& \cup \{(0, \mathbf{end}, \dagger, \dagger)\} && (1.16)
\end{aligned}
$$

We will not show in detail that R does indeed define a conformance relation, rather we show the dependencies that make R a conformance relation below. An arrow $\boxed{A} \longrightarrow \boxed{B}$ means that $A \in R$ is a prerequisite for $B \in R$.



When constructing a conformance relation such as the one above, situations which may normally be overseen in contract analysis, are identified. For

instance the (very unlikely) event that S sends the MMS requested by C right after C has accepted the offer: in this case S sends the MMS before before P has requested it, and even though this is very unlikely to happen, it still needs to be handled by the conformance relation (1.14). Notice also how the set (1.10) represents the case where C rejects P's offer, (1.15) represents the case of successful delivery, and (1.16) represents the case of failure to deliver the MMS.

In the next section we will see that R being a conformance relation for a, c_1, and c_2 means that a provides a strategy for P which will never result in a loss of money.

1.5.3 From Contracts to Automaton Contracts

This section defines the projection from (logical) contracts to automaton contracts, and formally proves that this transformation is sound. The transformation is defined with respect to a particular routing, which makes it possible to model delegation as external channels.

The transformation from logical to automaton contracts is straightforward; for each logical contract we define one automaton contract, the links that are routed to automaton channels are transformed directly to the corresponding channels, and logical delegations are transformed into "fresh" channels (as δ in Example 1.5.8). The essence of the transformation is the *renaming map*.

Definition 1.5.9 (Renaming map)**.** *Let $C = \{c_1, \ldots, c_n\}$ be a contract portfolio for a principal* P*, where $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$, and let $\Lambda = \bigcup_{i=1}^{n} \Lambda_{\mathsf{PA}_i} \cup \bigcup \Lambda_{\mathsf{A}_i\mathsf{P}}$. Given a routing $r = (r_1, r_2, r_3)$ for C and input/output channels C_I/C_O, we define a renaming map, θ, to be a map from links to channels:*

$$\theta : \Lambda \to \mathcal{C}$$

satisfying:

- $\forall \alpha \in C_I.\ \theta(r_1(\alpha)) = \alpha.$
- $\forall \alpha \in C_O.\ \theta(r_2(\alpha)) = \alpha.$
- $\forall \lambda_1, \lambda_2.\ r_3(\lambda_1) = \lambda_2 \Rightarrow \theta(\lambda_1) = \theta(\lambda_2).$
- $\forall \lambda_1, \lambda_2.\ \theta(\lambda_1) = \theta(\lambda_2) \Rightarrow (\lambda_1 = \lambda_2 \vee r_3(\lambda_1) = \lambda_2 \vee r_3(\lambda_2) = \lambda_1).$

The first three conditions state that renaming must agree with the routing map, and the last condition states that the renaming map must be injective (modulo delegation).

When we define the transformation from logical contracts to automaton contracts, we assume that a renaming map is available in order to avoid specifying channel names for the delegated links. But to give an explicit renaming is not hard; the renaming map will be uniquely determined on links that are not delegated, and delegations amount to choosing a fresh channel.

Definition 1.5.10 (Contract projection)**.** *Let $C = \{c_1, \ldots, c_n\}$ be a contract portfolio for a principal P, where $c_i = (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i)$. Given a routing $r = (r_1, r_2, r_3)$ for C and input/output channels C_I/C_O, and a renaming map θ respecting C and r, we define the contract projection to be $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$, where*

$$\mathsf{c}_i = (\theta(\Lambda_{\mathsf{PA}_i} \cup \Lambda_{\mathsf{A}_i\mathsf{P}}), G_i, g_i, \rho_i')$$

with

$$\rho_i'(g, m) = \rho_i(g, m \circ \theta_{|\Lambda_{\mathsf{PA}_i}}, m \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}})$$

We write $\{c_1, \ldots, c_n\} \rightsquigarrow^{\theta} \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ for this projection.

Observation 1.5.11. Whenever $C \rightsquigarrow^{\theta} \mathsf{C}$ and θ is a renaming map for r (r being a routing for C_I/C_O), then each channel $\alpha \in C_I \cup C_O$ is mentioned in at least one contract in C.

Example 1.5.12 (MMS greeting service, cont'd. from Example 1.5.8)**.** We have in fact already seen an example of a logical contract portfolio being mapped to a set of automaton contracts. In Example 1.5.8 the rename map θ is defined by

$$\theta(l) = \begin{cases} \alpha, & l = \lambda_1 \\ \gamma, & l = \lambda_2 \\ \delta, & l = \lambda_3 \vee l = \lambda_4 \\ \beta, & l = \lambda_5 \end{cases}$$

The reader may check that θ does indeed define a legal renaming map with respect to the routing of Example 1.5.3. The channel δ is an example of a logical route that is mapped to a "fresh" channel, which does not occur in the automaton of the implementation. This corresponds to labeling the "wire" of Figure 1.4 that is not connected to the automaton with δ. With the notion of Definition 1.5.10 we therefore have $\{c_1, c_2\} \rightsquigarrow^{\theta} \{\mathsf{c}_1, \mathsf{c}_2\}$.

With the definition of contract projection we are able to state and prove the soundness result.

Theorem 1.5.13 (Soundness of projection)**.** *Consider a contract portfolio $C = \{c_1, \ldots, c_n\}$ for P and routing $r = (r_1, r_2, r_3)$ for C and input/output channels C_I/C_O. If*

$$C \rightsquigarrow^{\theta} \mathsf{C}$$

where θ is a renaming map for r, and

$$\models \mathsf{a} : \mathsf{C}$$

for an automaton $\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$, then

$$\models (\mathsf{a}, r) : C$$

Proof. In Appendix B. □

Observation 1.5.14. The soundness result above together with the conformance relation built in Example 1.5.8, and the observation in Example 1.5.12, gives us that the implementation of the MMS greeting card service is indeed "good", i.e., P is guaranteed never to lose money on the service.

1.5.4 Compositionality

We conclude this section with the main result about contract conformance, namely that it is compositional. The theorem allows us to prove conformance for a parallel composition of automata by reasoning about the components in isolation. Compositionality here is an internal form of compositionality, expressing how a single principal can create a full implementation from subautomata. This compositionality is different from the external compositionality described earlier, which was realized via bilateral contracts between principals.

Definition 1.5.15 (Dual automaton contract)**.** *For an automaton contract $\mathsf{c} = (C, G, g_o, \rho)$, the dual contract $\overline{\mathsf{c}} = (C, G, g_0, \overline{\rho})$ is defined by $\overline{\rho}(g, m) = (g', -k)$, where $\rho(g, m) = (g', k)$.*

Definition 1.5.16 (Channel set)**.** *Let $\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$ be a set of automaton contracts, where $\mathsf{c}_i = (C_i, G_i, g_i, \rho_i)$. We then define*

$$\mathsf{chan}(\{\mathsf{c}_1, \ldots, \mathsf{c}_n\}) = \bigcup_{i=1}^{n} C_i$$

Theorem 1.5.17 (Contract conformance is compositional)**.** *Consider two automata $\mathsf{a}_1 = (C_I^1, C_O^1, S_1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I^2, C_O^2, S_2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$, where parallel composition*

$$\mathsf{a}_1 \parallel \mathsf{a}_2 = (C_I, C_O, S_1 \times S_2, \langle s_1, s_2 \rangle, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$

is defined (Definition 1.3.2). If

$$\begin{gathered} \models \mathsf{a}_1 : \mathsf{c}_1, \ldots, \mathsf{c}_n, \mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1} \\ \models \mathsf{a}_2 : \overline{\mathsf{c}_1}, \ldots, \overline{\mathsf{c}_n}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2} \\ \mathsf{chan}(\{\mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2}\}) \cap C_{\mathrm{int}} = \emptyset \end{gathered}$$

(where C_{int} *is the internal channels of* $\mathsf{a}_1 \parallel \mathsf{a}_2$*), then*

$$\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}'_1, \ldots, \mathsf{c}'_{n_1}, \mathsf{c}''_1, \ldots, \mathsf{c}''_{n_2}$$

Proof. In Appendix B. □

This theorem expresses how to fulfill a set of contracts by splitting the obligations between two *contractually compatible* automata. If the internal set of contracts (in the theorem they are written $\mathsf{c}_1, \ldots, \mathsf{c}_n$) is empty, then the theorem simply says that two disjoint automata can fulfill a set of contracts by partitioning the set between them. If the internal set of contracts is non-empty, then the contracts express how the automata can communicate internally to fulfill the external obligations. The duality expresses that they must play opposite roles in the internal contracts.

A special and important case is the one where we seek to fulfill one contract, c, but we subdivide it into smaller contracts, $\mathsf{c}_1, \ldots, \mathsf{c}_n$. We can then write automata for each of the smaller contracts and combine these via one "orchestrator" automaton, which communicates with all the other automata (the orchestrator must then conform with $\mathsf{c}, \overline{\mathsf{c}_1}, \ldots, \overline{\mathsf{c}_n}$). Parallel composition of the subautomata and the orchestrator is then guaranteed to conform with the original contract.

The theorem can be seen as a generalization of two rules of composition for sequential Hoare triples:

$$\text{If } \models \{A\}\, c_1\, \{C\} \text{ and } \models \{C\}\, c_2\, \{B\} \text{ then } \models \{A\}\, c_1; c_2\, \{B\} \tag{1.17}$$

$$\begin{array}{l} \text{If } \models \{A_1\}\, c_1\, \{B_1\} \text{ and } \models \{A_2\}\, c_2\, \{B_2\} \text{ and} \\ \quad \models \{A_2\}\, c_1\, \{A_2\} \text{ and } \models \{B_1\}\, c_2\, \{B_1\} \\ \text{then } \models \{A_1 \wedge A_2\}\, c_1; c_2\, \{B_1 \wedge B_2\} \end{array} \tag{1.18}$$

The first rule (1.17) corresponds to the latter case explained above, where internal contracts are utilized to fulfill a contract (the assertion C can be interpreted as the internal contract). The second rule (1.18) corresponds to the former case explained above, where the two automata have no internal contracts, which in the rule above means that the commands do not interfere. In Section 1.6.3 we illustrate the first correspondence in more detail.

We conclude this subsection with an example illustrating the compositionality theorem:

Example 1.5.18. This example uses the two processes from Example 1.2.13 as automata (cf. the translation in Definition 1.3.17). We use a doubler automaton, a_1, and an incrementer automaton, a_2. We want to show that

when we put them in parallel as in Example 1.2.13, they produce an ever growing list of integers, on the form $f^0(0), f^1(0), f^2(0), \ldots$, where $f(x) = 2x+1$ (we do not provide the actual formal proof here, only a proof sketch). We require that the integers must be sent with a delay of at most 10 time units (after the previous result). The automaton contract therefore says:

$$
\begin{aligned}
\mathsf{c} &= (\{\gamma\}, G, g_0, \rho) \\
\mathcal{A}_\gamma &= \mathbb{N} \cup \{\varepsilon\} \\
G &= \{\langle n, t\rangle \mid n \in \mathbb{N} \wedge t \in \mathbb{N}\} \cup \{\texttt{stop}\} \\
g_0 &= \langle 0, 10\rangle \\
\rho(\langle n, 1\rangle, m) &= \begin{cases} (\langle 2n+1, 10\rangle, 0) & \text{if } m(\gamma) = n \\ (\texttt{stop}, -1) & \text{otherwise} \end{cases} \\
\rho(\langle n, t+1\rangle, m) &= \begin{cases} (\langle n, t\rangle, 0) & \text{if } m(\gamma) = \varepsilon \\ (\langle 2n+1, 10\rangle, 0) & \text{if } m(\gamma) = n \\ (\texttt{stop}, -1) & \text{if } m(\gamma) \neq n \end{cases} \\
\rho(\texttt{stop}, m) &= (\texttt{stop}, 0)
\end{aligned}
$$

To show that the parallel composition fulfills this contract, we first write an "incrementer" contract for a_2, where a_2 is given 5 time units to produce its output after receiving an input. This means that a_1 will have time to calculate the "doubling" function as well:

$$
\begin{aligned}
\mathsf{c}_{\text{inc}} &= (\{\alpha, \beta, \gamma\}, G, g_0, \rho) \\
\mathcal{A}_c &= \mathbb{N} \cup \{\varepsilon\} \\
G &= \{\texttt{wait}\} \cup \{\langle n, t\rangle \mid n \in \mathbb{N} \wedge t \in \mathbb{N}\} \cup \{\texttt{stop}\} \\
g_0 &= \texttt{wait} \\
\rho(\texttt{wait}, m) &= \begin{cases} (\texttt{stop}, -1) & \text{if } m(\alpha) \neq \varepsilon \vee m(\gamma) \neq \varepsilon \\ (\texttt{wait}, 0) & \text{if } m(\beta) = \varepsilon \\ (\langle n+1, 5\rangle, 0) & \text{if } m(\beta) = n \end{cases} \\
\rho(\langle n, 1\rangle, m) &= \begin{cases} (\texttt{stop}, 1) & \text{if } m(\beta) \neq \varepsilon \\ (\texttt{wait}, 0) & \text{if } m(\alpha) = m(\gamma) = n \\ (\texttt{stop}, -1) & \text{otherwise} \end{cases} \\
\rho(\langle n, t+1\rangle, m) &= \begin{cases} (\texttt{stop}, 1) & m(\beta) \neq \varepsilon \\ (\langle n, t\rangle, 0) & \text{if } m(\alpha) = m(\gamma) = \varepsilon \\ (\texttt{wait}, 0) & \text{if } m(\alpha) = m(\gamma) = n \\ (\texttt{stop}, -1) & \text{otherwise} \end{cases} \\
\rho(\texttt{stop}) &= (\texttt{stop}, 0)
\end{aligned}
$$

It is fairly straightforward to show that $\models \mathsf{a}_2 : \mathsf{c}_{\text{inc}}$. The remaining obligation is to show that $\models \mathsf{a}_1 : \overline{\mathsf{c}_{\text{inc}}}, \mathsf{c}$ which we also omit here. By Theorem 1.5.17 with $C_{\text{int}} = \{\alpha, \beta\}$ it then follows that $\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}$, which was the goal.

1.6 Examples

In this section we present various examples of contracts expressed in our model.

1.6.1 Contract Language

To make the examples in following sections easier to formulate, we introduce a concrete programming-language-like syntax for expressing a subset of automaton contracts. The language should not been seen as a restriction on the model, only as a shorthand for a large class of contracts (in other words, the contract language is not as expressive as the contract model). Even though the syntax is formalized for automaton contracts, we can just as well use it for (principal) contracts (by using links instead of channels).

Syntax

The abstract syntax of the contract language is presented below.

$$
\begin{aligned}
\text{Num} \ni\ & \mathbf{n} \\
\text{Var} \ni\ & Cn, Sn, x \\
\text{Chvar} \ni\ & Ch \\
\text{Contract} \ni c ::=\ & \underline{\texttt{contract}}\ Cn\ \texttt{\{}\ Cb\ \texttt{\}} \\
\text{ContractBody} \ni Cb ::=\ & \underline{\texttt{in}}\ d_1 \ldots d_{n_1}\ \underline{\texttt{out}}\ d'_1 \ldots d'_{n_2}\ \underline{\texttt{start}}\ Se\texttt{;}\ s_1 \ldots s_k \\
\text{Chdecl} \ni d ::=\ & Ch\ \texttt{:}\ t\texttt{;} \\
\text{Type} \ni t ::=\ & \underline{\texttt{nat}} \mid \underline{\texttt{int}} \mid \underline{\texttt{bool}} \\
\text{State} \ni s ::=\ & \underline{\texttt{state}}\ Sn\texttt{(}p_1, \ldots p_n\texttt{)\{}\ Sb\ \texttt{\}} \\
\text{Param} \ni p ::=\ & x\ \texttt{:}\ t \\
\text{StateBody} \ni Sb ::=\ & h_1 \ldots h_n\ \tau \\
\text{Handle} \ni h ::=\ & \underline{\texttt{when}}\ x\ \underline{\texttt{on}}\ Ch\ g\ \underline{\texttt{do}}\ Tr\texttt{;} \\
\text{Timeout} \ni \tau ::=\ & \cdot \mid \underline{\texttt{timeout}}\ e\texttt{:}\ Tr\texttt{;} \\
\text{Guard} \ni g ::=\ & \cdot \mid \underline{\texttt{where}}\ e \\
\text{Trans} \ni Tr ::=\ & Se\ \underline{\texttt{with}}\ e \mid \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ Tr_1\ \underline{\texttt{else}}\ Tr_2 \\
\text{Sexp} \ni Se ::=\ & Sn\texttt{(}e_1, \ldots, e_n\texttt{)} \\
\text{Exp} \ni e ::=\ & \mathbf{n} \mid x \mid \underline{\texttt{true}} \mid \underline{\texttt{false}} \mid e_1\ o\ e_2 \mid \\
& \underline{\texttt{if}}\ e_1\ \underline{\texttt{then}}\ e_2\ \underline{\texttt{else}}\ e_3 \\
\text{Opr} \ni o ::=\ & \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/} \mid \texttt{=}
\end{aligned}
$$

The syntax comes with the following comments:

- Numbers (Num) are non-empty sequences of digits possibly prefixed by a sign:
$$[\texttt{-}]?[\texttt{0}-\texttt{9}]^+$$
- Variables (Var) are ordinary strings starting with a letter:
$$[\texttt{a}-\texttt{zA}-\texttt{Z}][\texttt{a}-\texttt{zA}-\texttt{Z0}-\texttt{9}]^*$$
- In declaration of channels, states and parameters all names are distinct.
- We assume to have a predefined end state (`stop`) in all contracts defined by:
```
state stop() {}
```
- We write `-`e as syntactic sugar for 0 `-` e.
- We write multiplication with a rational as a shorthand for integer multiplication and division, e.g., 1.2 `*` e instead of (12 `*` e) `/` 10.

Example 1.6.1. The contract:

> *Wait for a number, n, on channel a, and then return $n+1$ on channel b, after at most 10 steps.*

can be written as:

```
contract c
{
  in a : int;
  out b : int;
  start init();

  state init()
  {
    when x on a do snd(x);
  }

  state snd(n : int)
  {
    when x on b do
      if x = n+1 then stop() else stop() with -1;
    timeout 10 : stop() with -1;
  }
}
```

The keyword `with` is used to indicate the payoff associated with the state transition (absence of `with` means an implicit "`with` 0"); in the above, payoff 0 denotes success and payoff -1 denotes failure.

Informal Semantics

In this section we give a short informal semantics of the contract language (in the next sections we show the formal semantics, via a mapping into the contract model, but the reader may skip that section, since it presents nothing conceptually new). A contract consists of a name which has no semantic meaning and a contract body. The body consists of three different parts:

- Input- and output channels with type annotations. The types specify the action set of the channels;
- The start state; and
- The state *classes* of the contract.

A state class consists of a finite set of variables, which – when instantiated – comprises a single game state. The body of a state class consists of three parts:

- The channel handlers, which are used to specify which state to transition to after certain actions are communicated on the channels (and with possible payoff). Handlers are evaluated from top to bottom.
- A timeout handler, which can trigger if none of the channel handlers are triggered, and the specified timeout is reached (also with possible payoff).

Channel handlers are associated with exactly one channel, and may be *guarded*. The body of the handler specifies the next state class and payoff. If no handler (channel or timeout) is triggered, then the contract will remain in the same state class.

State expressions evaluate to states, and expressions evaluate to simple values ($\underline{\texttt{int}}, \underline{\texttt{nat}}, \underline{\texttt{bool}}$), where $\underline{\texttt{int}}$ is a standard integer type, $\underline{\texttt{nat}}$ is used to specify timeouts and $\underline{\texttt{bool}}$ is used in conditionals. Division is integer division rounded towards $-\infty$.

Typing

Only a subset of the syntactically well-formed contracts make sense, so we define well-typed contracts. First we define 3 types of contexts: variable contexts (Γ), channel contexts (Ψ) and state contexts (Φ):

$$\begin{aligned}\Gamma &= \cdot \mid x : t, \Gamma \\ \Psi &= \cdot \mid \mathit{Ch} : t, \Psi \\ \Phi &= \cdot \mid \mathit{Sn} : (t_1, \ldots, t_n), \Phi\end{aligned}$$

We then define typing rules for the different syntactical categories of the language:

$\boxed{\vdash c}$

$$\textsf{t-contract}\ \frac{\vdash Cb}{\vdash \underline{\texttt{contract}}\ Cn\ \texttt{\{}\ Cb\ \texttt{\}}}$$

$\boxed{\vdash Cb}$

$$\textsf{t-contract-body}\ \frac{\vdash Ds_1, Ds_2; Ss \Rightarrow \Psi; \Phi \qquad \Phi; \cdot \vdash Se \qquad \Psi; \Phi \vdash Ss}{\vdash \underline{\texttt{in}}\ Ds_1\ \underline{\texttt{out}}\ Ds_2\ \underline{\texttt{start}}\ Se\texttt{;}\ Ss}$$

$\boxed{\vdash Ds; Ss \Rightarrow \Psi; \Phi}$

$$\textsf{t-env}\ \frac{\vdash Ds \Rightarrow \Psi \qquad \vdash Ss \Rightarrow \Phi}{\vdash Ds; Ss \Rightarrow \Psi; \Phi}$$

$\boxed{\vdash Ds \Rightarrow \Psi}$

$$\textsf{t-decl-nil}\ \frac{}{\vdash \cdot \Rightarrow \cdot} \qquad \textsf{t-decl-cons}\ \frac{\vdash Ds \Rightarrow \Psi'}{\vdash Ch\texttt{:}t\ Ds \Rightarrow Ch : t, \Psi'}$$

$\boxed{\vdash Ss \Rightarrow \Phi}$

$$\textsf{t-state-nil}\ \frac{}{\vdash \cdot \Rightarrow \cdot}$$

$$\textsf{t-state-cons}\ \frac{\vdash Ss \Rightarrow \Phi}{\vdash \underline{\texttt{state}}\ Sn\texttt{(}x_1\texttt{:}t_1, \ldots, x_n\texttt{:}t_n\texttt{)\{}\ Sb\ \texttt{\}}\ Ss \Rightarrow Sn : (t_1, \ldots, t_n), \Phi}$$

$\boxed{\Psi; \Phi \vdash Ss}$

$$\textsf{t-states}\ \frac{\Psi; \Phi \vdash s_1\ \cdots\ \Psi; \Phi \vdash s_n}{\Psi; \Phi \vdash s_1 \ldots s_n}$$

$\boxed{\Psi; \Phi \vdash s}$

$$\textsf{t-state}\ \frac{\vdash Ps \Rightarrow \Gamma \qquad \Psi; \Phi; \Gamma \vdash Sb}{\Psi; \Phi \vdash \underline{\texttt{state}}\ Sn\texttt{(}Ps\texttt{)\{}\ Sb\ \texttt{\}}}$$

$\boxed{\vdash Ps \Rightarrow \Gamma}$

$$\textsf{t-param-nil}\ \frac{}{\vdash \cdot \Rightarrow \cdot} \qquad \textsf{t-param-cons}\ \frac{\vdash Ps \Rightarrow \Gamma}{\vdash x{:}t, Ps \Rightarrow x : t, \Gamma}$$

$\boxed{\Psi; \Phi; \Gamma \vdash Sb}$

$$\textsf{t-state-body}\ \frac{\Psi; \Phi; \Gamma \vdash Hs \qquad \Phi; \Gamma \vdash \tau}{\Psi; \Phi; \Gamma \vdash Hs\ \tau}$$

$\boxed{\Phi; \Gamma \vdash \tau}$

$$\textsf{t-timeout-empty}\ \frac{}{\Phi; \Gamma \vdash \cdot} \qquad \textsf{t-timeout}\ \frac{\Gamma \vdash e : \underline{\texttt{nat}} \qquad \Phi; \Gamma \vdash Tr}{\Phi; \Gamma \vdash \underline{\texttt{timeout}}\ e\texttt{:}\ Tr\texttt{;}}$$

$\boxed{\Psi; \Phi; \Gamma \vdash Hs}$

$$\textsf{t-handlers}\ \frac{\Psi; \Phi; \Gamma \vdash h_1\ \cdots\ \Psi; \Phi; \Gamma \vdash h_n}{\Psi; \Phi; \Gamma \vdash h_1 \ldots h_n}$$

$\boxed{\Psi; \Phi; \Gamma \vdash h}$

$$\textsf{t-handler}\ \frac{\Psi(Ch) = t \qquad x : t, \Gamma \vdash g \qquad \Phi; x : t, \Gamma \vdash Tr}{\Psi; \Phi; \Gamma \vdash \underline{\texttt{when}}\ x\ \underline{\texttt{on}}\ Ch\ g\ \underline{\texttt{do}}\ Tr\texttt{;}}$$

$\boxed{\Gamma \vdash g}$

$$\textsf{t-guard-empty}\ \frac{}{\Gamma \vdash \cdot} \qquad \textsf{t-guard}\ \frac{\Gamma \vdash e : \underline{\texttt{bool}}}{\Gamma \vdash \underline{\texttt{where}}\ e}$$

$\boxed{\Phi; \Gamma \vdash Tr}$

$$\textsf{t-trans-payoff}\ \frac{\Phi; \Gamma \vdash Se \qquad \Gamma \vdash e : \underline{\texttt{int}}}{\Phi; \Gamma \vdash Se\ \underline{\texttt{with}}\ e}$$

$$\textsf{t-trans-if}\ \frac{\Gamma \vdash e : \underline{\texttt{bool}} \qquad \Phi; \Gamma \vdash Tr_1 \qquad \Phi; \Gamma \vdash Tr_2}{\Phi; \Gamma \vdash \underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ Tr_1\ \underline{\texttt{else}}\ Tr_2}$$

$\boxed{\Phi; \Gamma \vdash Se}$

$$\textsf{t-sexp-call}\ \frac{\Phi(Sn) = (t_1, \ldots, t_n) \qquad \Gamma \vdash e_1 : t_1\ \cdots\ \Gamma \vdash e_n : t_n}{\Phi; \Gamma \vdash Sn(e_1, \ldots, e_n)}$$

$\boxed{\Gamma \vdash e : t}$

$$\textsf{t-exp-var}\ \frac{\Gamma(x) = t}{\Gamma \vdash x : t} \qquad \textsf{t-exp-sub}\ \frac{\Gamma \vdash e : \underline{\texttt{nat}}}{\Gamma \vdash e : \underline{\texttt{int}}}$$

$$\textsf{t-exp-nat}\ \frac{\mathbf{n} \geq 1}{\Gamma \vdash \mathbf{n} : \underline{\texttt{nat}}} \qquad \textsf{t-exp-int}\ \frac{}{\Gamma \vdash \mathbf{n} : \underline{\texttt{int}}}$$

$$\textsf{t-exp-true}\ \frac{}{\Gamma \vdash \underline{\texttt{true}} : \underline{\texttt{bool}}} \qquad \textsf{t-exp-false}\ \frac{}{\Gamma \vdash \underline{\texttt{false}} : \underline{\texttt{bool}}}$$

$$\textsf{t-exp-opr}\ \frac{\Gamma \vdash e_1 : t_1 \qquad \Gamma \vdash e_2 : t_2 \qquad \overline{o}(t_1, t_2) = t}{\Gamma \vdash e_1\ o\ e_2 : t}$$

$$\textsf{t-exp-if}\ \frac{\Gamma \vdash e_1 : \underline{\texttt{bool}} \qquad \Gamma \vdash e_2 : t \qquad \Gamma \vdash e_3 : t}{\Gamma \vdash \underline{\texttt{if}}\ e_1\ \underline{\texttt{then}}\ e_2\ \underline{\texttt{else}}\ e_3 : t}$$

The operator for types is defined as:

$$\overline{+}(t_1, t_2) = \begin{cases} \underline{\texttt{nat}} & \text{if } t_1 = t_2 = \underline{\texttt{nat}} \\ \underline{\texttt{int}} & \text{otherwise} \end{cases}$$

$$\overline{-}(t_1, t_2) = \underline{\texttt{int}}$$

$$\overline{*}(t_1, t_2) = \begin{cases} \underline{\texttt{nat}} & \text{if } t_1 = t_2 = \underline{\texttt{nat}} \\ \underline{\texttt{int}} & \text{otherwise} \end{cases}$$

$$\overline{/}(t_1, t_2) = \underline{\texttt{int}}$$

$$\overline{=}(t_1, t_2) = \begin{cases} \underline{\texttt{bool}} & \text{if } t_1 = t_2 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Translation

Having defined typing, we can define a mapping from well-typed contracts to automaton contracts. The intuition behind this translation is to map each state class to a set of game states (one for each instantiation of the variables in the class). Timeouts are modeled by extending each game state with a counter, which records the number of contract transitions since the last "proper" transition (either a triggered channel handler or a triggered timeout handler).

First we define a set for each type:

$$[\![\underline{\texttt{nat}}]\!] = \mathbb{N} \quad [\![\underline{\texttt{int}}]\!] = \mathbb{Z} \quad [\![\underline{\texttt{bool}}]\!] = \{\mathbf{t}, \mathbf{f}\}$$

These sets are lifted to variable contexts:

$$[\![x_1 : t_1, \ldots, x_n : t_n]\!] = \left\{\gamma : \{x_i\}_{i \in \{1,\ldots,n\}} \to \bigcup_{i=1}^{n} [\![t_i]\!] \mid \forall i.1 \leq i \leq n \Rightarrow \gamma(x_i) \in [\![t_i]\!]\right\}$$

(i.e., $[\![\Gamma]\!]$ represents the set of all maps assigning (type-correct) values to the variables of Γ). To define the contract, we define a series of translation functions which operate on well-typed contracts.

For a well-typed contract body:

$$Cb = \underline{\texttt{in}}\ d_1 \dots d_{n_1}\ \underline{\texttt{out}}\ d'_1 \dots d'_{n_2}\ \underline{\texttt{start}}\ Se\texttt{;}\ s_1 \dots s_k$$

where

$$s_i = \underline{\texttt{state}}\ Sn_i\texttt{(}x_1\texttt{:}t_1, \dots, x_{n_i}\texttt{:}t_{n_i}\texttt{)\{}\ Sb_i\ \texttt{\}}$$

for $i = 1, \dots, k$, we define $[\![Cb]\!]$ to be the automaton contract:

$$(\ulcorner d_1 \dots d_{n_1}\ d'_1 \dots d'_{n_2} \urcorner, G, [\![Se]\!](\emptyset), \rho)$$

where

$$G = \ulcorner s_1 \urcorner \cup \dots \cup \ulcorner s_k \urcorner$$

$$\rho(\langle Sn_i, t, v_1, \dots, v_{n_i}\rangle, m) = \begin{cases} (\langle Sn_i, t+1, v_1, \dots, v_{n_i}\rangle, 0) & \text{if } q = \star \\ q & \text{otherwise} \end{cases}$$

$$\text{where } q = [\![Sb_i]\!](\{x_1 \mapsto v_1, \dots, x_{n_i} \mapsto v_{n_i}\}, m, t)$$

The translation of channel declarations and state classes are given below:

$$\ulcorner Ch_1\texttt{:}t_1\texttt{;} \dots Ch_n\texttt{:}t_n\texttt{;} \urcorner = \{Ch_1, \dots, Ch_n\}$$

$$\mathcal{A}_{Ch_i} = [\![t_i]\!] \cup \{\varepsilon\}, \quad i \in \{1, \dots, n\}$$

$$\ulcorner \underline{\texttt{state}}\ Sn_i\texttt{(}x_1\texttt{:}t_1, \dots x_{n_i}\texttt{:}t_{n_i}\texttt{)\{}\ Sb_i\ \texttt{\}} \urcorner = \{\langle Sn_i, t, v_1, \dots, v_{n_i}\rangle \mid t \in \mathbb{N} \wedge v_j \in [\![t_j]\!]\}$$

The intuition mentioned above is reflected in the definition of the state class translation and the contract transition function ρ: a state class Sn_i denotes the set of game states $\{\langle Sn_i, t, v_1, \dots, v_{n_i}\rangle \mid t \in \mathbb{N} \wedge v_j \in [\![t_j]\!]\}$, where t is the counter used for timeouts and $v_1, \dots, v_{n_i}$ are the instantiated values. In the definition of ρ, the internal counter is incremented, when no "proper" transition is made (which is encoded as the predicate $q = \star$), and when a proper transition is made, the internal counter is initialized to 1 (which will be seen in the translation of transitions).

Translation for state bodies (below C is the set of channels in Ψ):

$$\text{For } \Psi; \Phi; \Gamma \vdash Sb$$

$$[\![Sb]\!] : [\![\Gamma]\!] \times \mathcal{M}_C \times \mathbb{N} \to (G \times \mathbb{Z}) + \{\star\}$$

$$[\![h_1 \dots h_n\ \tau]\!](\gamma, m, t) = \begin{cases} [\![\tau]\!](\gamma, t) & \text{if } [\![h_1 \dots h_n]\!](\gamma, m) = \star \\ (g, k) & \text{if } [\![h_1 \dots h_n]\!](\gamma, m) = (g, k) \end{cases}$$

Translation for handlers (below C is the set of channels in Ψ):

$$\begin{array}{c}\text{For } \Psi;\Phi;\Gamma \vdash h_1 \ldots h_n \\ [\![h_1 \ldots h_n]\!] : [\![\Gamma]\!] \times \mathcal{M}_C \to (G \times \mathbb{Z}) + \{\star\} \\ [\![\cdot]\!](\gamma, m) = \star \\ [\![h\ Hs]\!](\gamma, m) = \begin{cases} [\![Hs]\!](\gamma, m) & \text{if } [\![h]\!](\gamma, m) = \star \\ (g, k) & \text{if } [\![h]\!](\gamma, m) = (g, k) \end{cases}\end{array}$$

$$\begin{array}{c}\text{For } \Psi;\Phi;\Gamma \vdash h \\ [\![h]\!] : [\![\Gamma]\!] \times \mathcal{M}_C \to (G \times \mathbb{Z}) + \{\star\} \\ [\![h]\!](\gamma, m) = \begin{cases} \star & \text{if } m(Ch) = \varepsilon \\ \star & \text{if } m(Ch) = v \neq \varepsilon \wedge [\![g]\!](\gamma[x \mapsto v]) = \mathbf{f} \\ [\![Tr]\!](\gamma[x \mapsto v]) & \text{if } m(Ch) = v \neq \varepsilon \wedge [\![g]\!](\gamma[x \mapsto v]) = \mathbf{t} \end{cases} \\ \text{where } h = \underline{\texttt{when}}\ x\ \underline{\texttt{on}}\ Ch\ g\ \underline{\texttt{do}}\ Tr\texttt{;}\end{array}$$

Translation for guards:

$$\begin{array}{c}\text{For } \Gamma \vdash g \\ [\![f]\!] : [\![\Gamma]\!] \to \{\mathbf{t}, \mathbf{f}\} \\ [\![\cdot]\!](\gamma) = \mathbf{t} \\ [\![\underline{\texttt{where}}\ e]\!](\gamma) = [\![e]\!](\gamma)\end{array}$$

Translation for timeouts:

$$\begin{array}{c}\text{For } \Phi;\Gamma \vdash \tau \\ [\![\tau]\!] : [\![\Gamma]\!] \times \mathbb{N} \to (G \times \mathbb{Z}) + \{\star\} \\ [\![\cdot]\!](\gamma, t) = \star \\ [\![\underline{\texttt{timeout}}\ e\texttt{:}\ Tr\texttt{;}]\!](\gamma, t) = \begin{cases} \star & \text{if } t < [\![e]\!](\gamma) \\ [\![Tr]\!](\gamma) & \text{if } t \geq [\![e]\!](\gamma) \end{cases}\end{array}$$

Translation for state transitions:

$$\begin{array}{c}\text{For } \Phi;\Gamma \vdash Tr \\ [\![Tr]\!] : [\![\Gamma]\!] \to G \times \mathbb{Z} \\ [\![Se\ \underline{\texttt{with}}\ e]\!](\gamma) = ([\![Se]\!]\gamma, [\![e]\!]\gamma) \\ [\![\underline{\texttt{if}}\ e\ \underline{\texttt{then}}\ Tr_1\ \underline{\texttt{else}}\ Tr_2]\!](\gamma) = \begin{cases} [\![Tr_1]\!](\gamma) & \text{if } [\![e]\!](\gamma) = \mathbf{t} \\ [\![Tr_2]\!](\gamma) & \text{if } [\![e]\!](\gamma) = \mathbf{f} \end{cases}\end{array}$$

Translation for state expressions:

$$\begin{array}{c}\text{For } \Phi;\Gamma \vdash Se \\ [\![Se]\!] : [\![\Gamma]\!] \to G \\ [\![Sn\texttt{(}e_1, \ldots, e_n\texttt{)}]\!](\gamma) = (Sn, 1, [\![e_1]\!](\gamma), \ldots, [\![e_n]\!](\gamma))\end{array}$$

Translation for expressions:

$$
\begin{aligned}
&\text{For } \Gamma \vdash e : t \\
&\qquad [\![e]\!] : [\![\Gamma]\!] \to [\![t]\!] \\
[\![\mathbf{n}]\!](\gamma) &= \mathbf{n} \\
[\![\underline{\texttt{true}}]\!](\gamma) &= \mathbf{t} \\
[\![\underline{\texttt{false}}]\!](\gamma) &= \mathbf{f} \\
[\![x]\!](\gamma) &= \gamma(x) \\
[\![e_1 \ o \ e_2]\!](\gamma) &= [\![o]\!]([\![e_1]\!], [\![e_2]\!]) \\
[\![\underline{\texttt{if}} \ e_1 \ \underline{\texttt{then}} \ e_2 \ \underline{\texttt{else}} \ e_3]\!](\gamma) &= \begin{cases} [\![e_2]\!](\gamma) & \text{if } [\![e_1]\!](\gamma) = \mathbf{t} \\ [\![e_3]\!](\gamma) & \text{if } [\![e_1]\!](\gamma) = \mathbf{f} \end{cases}
\end{aligned}
$$

Translation for operators:

$$
\begin{aligned}
[\![\texttt{+}]\!](n_1, n_2) &= n_1 + n_2 \\
[\![\texttt{-}]\!](n_1, n_2) &= n_1 - n_2 \\
[\![\texttt{*}]\!](n_1, n_2) &= n_1 \ n_2 \\
[\![\texttt{/}]\!](n_1, n_2) &= \left\lfloor \frac{n_1}{n_2} \right\rfloor \\
[\![\texttt{=}]\!](n_1, n_2) &= \begin{cases} \mathbf{t} & \text{if } n_1 = n_2 \\ \mathbf{f} & \text{if } n_1 \neq n_2 \end{cases}
\end{aligned}
$$

Example 1.6.2. If we take the contract `c` from Example 1.6.1 and perform the translation we get the following automaton contract:

$$
\begin{aligned}
\texttt{c} &= (C, G, g_0, \rho) \\
C &= \{\texttt{a}, \texttt{b}\} \\
G &= \{\langle \texttt{init}, t\rangle \mid t \in \mathbb{N}\} \\
&\quad \cup \{\langle \texttt{snd}, t, v\rangle \mid t \in \mathbb{N} \wedge v \in \mathbb{Z}\} \\
&\quad \cup \{\langle \texttt{stop}, t\rangle \mid t \in \mathbb{N}\} \\
g_0 &= \langle \texttt{init}, 1\rangle \\
\rho(\langle \texttt{init}, t\rangle, m) &= \begin{cases} (\langle \texttt{init}, t+1\rangle, 0) & \text{if } m(\texttt{a}) = \varepsilon \\ (\langle \texttt{snd}, 1, v\rangle, 0) & \text{if } m(\texttt{a}) = v \end{cases} \\
\rho(\langle \texttt{snd}, t, v\rangle, m) &= \begin{cases} (\langle \texttt{snd}, t+1, v\rangle, 0) & \text{if } m(\texttt{b}) = \varepsilon \wedge t < 10 \\ (\langle \texttt{stop}, 1\rangle, -1) & \text{if } m(\texttt{b}) = \varepsilon \wedge t \geq 10 \\ (\langle \texttt{stop}, 1\rangle, 0) & \text{if } m(\texttt{b}) = v + 1 \\ (\langle \texttt{stop}, 1\rangle, -1) & \text{if } m(\texttt{b}) \neq v + 1 \end{cases} \\
\rho(\langle \texttt{stop}, t\rangle, m) &= (\langle \texttt{stop}, t+1\rangle, 0)
\end{aligned}
$$

1.6.2 PBC

We now show how the contracts of Example 1.4.7 can be written more compactly in the contract language. We first repeat the two (informal) contracts negotiated by P with C and S respectively:

> C *can request the price of sending greeting card* g *to phone number* n*, and* P *can reply with a price* p*. Subsequently* C *can accept or reject. If* C *accepts,* P *has to send the MMS before at most* t *time units. If* P *fails to do so,* P *is assigned a penalty of* 1.

> P *can request an MMS to phone number* n *with content* c *at a price* $f(n, c)$ *(for some predefined rate function* f*). Subsequently* S *must send the MMS before* t' *time units. If* S *fails to do so,* S *is assigned a penalty of* 1.

We assume the contract language is extended with strings, abstract datatypes (using an ML-like syntax), in order to tag the different message types, and that any free variables in a `where` clause is a binding constructor. The formalized contracts are presented below.

```
// Communication with client (C)
datatype comm1 = req of int * string | accept | reject;
datatype comm2 = price of int;

// Communication with MMS gateway (S)
datatype comm3 = req of int * string;

// Communication with client (C) and MMS gateway (S)
datatype mms = mms of int * string;

// Contract with C
contract c1
{
  in  a : comm1;
  out c : comm2;
  out d : mms;
  start init();

  state init()
  {
    when x on a where x = req(n,g) do req_received(n,g);
  }

  state req_received(n : int, s : string)
  {
    when x on c where x = price(p) do price_offered(n,g,p);
  }

  state price_offered(n : int, g : string, p : int)
  {
    when x on a where x = reject do stop();
```

```
    when x on a where x = accept do accepted(n,g,p) with p;
  }

  state accepted(n : int, g : string, p : int)
  {
   when x on d where x = mms(n,g) do stop();
   timeout t : stop() with -p-1;
  }
}

// Contract with S
contract c2
{
  in  d : mms;
  out b : comm3;
  start init();

  state init()
  {
    // f computes MMS cost
    when x on b where x = req(n,g) do send_mms(n,g) with -f(n,g);
  }

  state send_mms(n : int, g : string)
  {
    // note: n and g are not free in mms(n,g)
    when x on d where x = mms(n,g) do stop();
    timeout t' : stop() with f(n,g)+1;
  }
}
```

The ongoing example formalized in the contract language above, has illustrated the concepts of compositionality (at principal level) and assuming responsibility for success. However, it does not illustrate the adversarial nature of system composition, as P has no intentions of "gaming" the client C.

In order to illustrate this, we consider a new version of the scenario with principals P, C, and S, however, this time we will not provide full formalizations; we only briefly sketch the ideas. Once again we consider the contracts from the viewpoint of P:

The game describing the contract with the C is illustrated in Figure 1.6 (left). A is the starting state, from which the client can request the MMS greeting (this time the greeting card is fixed for simplicity). This service has a certain cost, which is modeled as a payoff when the MMS is sent. The service provider guarantees to send the greeting MMS after at most 5 minutes (depicted as a timeout in the figure); otherwise the client gets a discount of twenty percent. If the MMS has not been sent after a total of 15 minutes, the client gets a full refund plus an additional twenty percent of the price. This contract illustrates both performance and absolute guarantees; initially the service provider has a 5 minutes deadline for sending the MMS, however if this deadline is exceeded, then an extended deadline of 10 minutes

becomes active, but complying with this deadline is still an indication of "poorer performance".

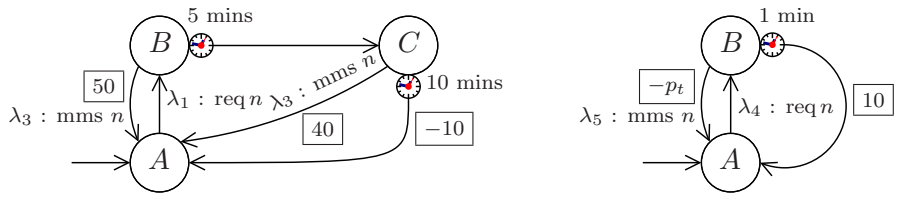


Figure 1.6: The service provider's contract with the client, c_{PC} (left) and the service provider's contract with the MMS gateway, c_{PS} (right).

The service provider's contract with the MMS gateway is illustrated in Figure 1.6 (right). The starting state is A, from which the provider can request the sending of an (arbitrary) MMS, at a cost p_t. The subscripted t means that the price depends on the variable t, which refers to the current time (this can be encoded by adding a counter to the contract states, as we have seen earlier). The value of p_t is defined to be

$$p_t = \begin{cases} 15, & 20 \leq \mathit{hour}(t) < 6 \\ 30, & \text{otherwise} \end{cases}$$

i.e., if an MMS is requested between 20:00 and 06:00 then the price is 15, otherwise it is 30 (discount during night hours). If the MMS gateway fails to deliver the MMS within at most one minute, a small penalty is assigned to the MMS gateway.

Now the service provider has to implement a program/strategy for sending MMS greetings for the client. A straightforward implementation is to request sending from the gateway right away upon request from the client, similar to the implementation earlier. This will produce a profit of either 35 (successful delivery, night hours), 20 (successful delivery, day hours) or 0 (unsuccessful delivery). But a more clever approach is for the service provider to *game* the client, by deliberately postponing delivery of greetings requested between 19:46 and 20:00; such a strategy will produce a profit of either 35, 25 or 0 (as opposed to the straight-forward strategy which will produce either 20 or 0). We note that postponing requests received between 19:45 and 19:46 is not safe, since the MMS gateway has a possible delay of one minute, which in the worst case means that a penalty of twenty percent has to be paid to the client and the price of the MMS has to be paid to the gateway.

1.6.3 Hoare Logic

In this section we show how to model regular Hoare triples. We assume some imperative language of commands with a small-step operational semantics (e.g., IMP as defined by Winskel [68]). Formally we have commands ($c \in \mathbf{Com}$), stores ($\sigma \in \mathbf{Store}$) and a (deterministic) step relation on configurations: $\rightarrow \subseteq (\mathbf{Com} \times \mathbf{Store}) \times (\mathbf{Com} \times \mathbf{Store})$.

Intuitively, we model a command c as an automaton with two channels: One channel for input of the start store, and one channel for output of the resulting store. One step in the small-step semantics, $\langle c, \sigma\rangle \rightarrow \langle c', \sigma'\rangle$, corresponds to an automaton transition without output. When $\langle c, \sigma\rangle \not\rightarrow$, the automaton continuously outputs the result state σ' (which may be the result of either a "properly" ending computation, or c being stuck). This intuition is formalized below.

Definition 1.6.3. *The transformation* $[\![\cdot]\!] : \mathbf{Com} \times \mathcal{C} \times \mathcal{C} \rightarrow \mathfrak{A}$ *is defined by* $[\![c, \alpha, \beta]\!] = (\{\alpha\}, \{\beta\}, S, s_0, \delta_o, \delta_t)$, *where:*

$$
\begin{aligned}
\mathcal{A}_\alpha &= \mathcal{A}_\beta = \mathbf{Store} \cup \{\varepsilon\} \\
S &= \{\langle c', \sigma'\rangle \mid c' \in \mathbf{Com} \wedge \sigma' \in \mathbf{Store}\} \cup \{\star\} \\
s_0 &= \star \\
\delta_o(\star)(\beta) &= \varepsilon \\
\delta_o(\langle c', \sigma'\rangle)(\beta) &= \begin{cases} \sigma' & \text{if } \langle c', \sigma'\rangle \not\rightarrow \\ \varepsilon & \text{otherwise} \end{cases} \\
\delta_t(\star, m) &= \begin{cases} \star & \text{if } m(\alpha) = \varepsilon \\ \langle c, \sigma\rangle & \text{if } m(\alpha) = \sigma \end{cases} \\
\delta_t(\langle c', \sigma'\rangle, m) &= \begin{cases} \langle c', \sigma'\rangle & \text{if } \langle c', \sigma'\rangle \not\rightarrow \\ \langle c'', \sigma''\rangle & \text{if } \langle c', \sigma'\rangle \rightarrow \langle c'', \sigma''\rangle \end{cases}
\end{aligned}
$$

A Hoare triple consists of a pre- and a postcondition:

$$\{A\}\, c\, \{B\}$$

where A and B are from a set of assertions (**Assn**) which is a predicate on stores:

$$\mathbf{Assn} \subseteq \mathbf{Store}$$

We write $\sigma \models A$ if $\sigma \in A$. Partial correctness of a Hoare triple is written:

$$\models \{A\}\, c\, \{B\}$$

and is defined by

$$\forall \sigma.\ (\sigma \models A \wedge \langle c, \sigma\rangle \rightarrow^* \langle c', \sigma'\rangle \not\rightarrow) \Rightarrow \sigma' \models B$$

A Hoare triple will be modeled by a contract, which produces negative payoff if the result store does not satisfy B, whenever the starting store satisfies A. Partial correctness then corresponds to contract conformance.

Definition 1.6.4. *Define a transformation on assertions:*

$$[\![\cdot]\!] : \mathbf{Assn} \times \mathbf{Assn} \times \mathcal{C} \times \mathcal{C} \to \mathfrak{C}$$

by:

$$[\![A, B, \alpha, \beta]\!] = (\{\alpha, \beta\}, G, g_0, \rho)$$

where

$$\begin{aligned}
G &= \{\mathtt{start}, \mathtt{run}, \mathtt{end}\} \\
g_0 &= \mathtt{start} \\
\rho(\mathtt{start}, m) &= \begin{cases} (\mathtt{end}, -1)^1 & \text{if } m(\beta) \neq \varepsilon \\ (\mathtt{start}, 0) & \text{if } m(\alpha) = \varepsilon \\ (\mathtt{run}, 0) & \text{if } m(\alpha) = \sigma \wedge \sigma \models A \\ (\mathtt{end}, 0) & \text{if } m(\alpha) = \sigma \wedge \sigma \not\models A \end{cases} \\
\rho(\mathtt{run}, m) &= \begin{cases} (\mathtt{run}, 0) & \text{if } m(\beta) = \varepsilon \\ (\mathtt{end}, 0) & \text{if } m(\beta) = \sigma \wedge \sigma \models B \\ (\mathtt{end}, -1) & \text{if } m(\beta) = \sigma \wedge \sigma \not\models B \end{cases} \\
\rho(\mathtt{end}, m) &= (\mathtt{end}, 0)
\end{aligned}$$

We know briefly sketch how the connection from contract conformance to partial correctness can be established formally. We also sketch how compositionality of contract conformance does indeed generalize compositionality of partial correctness. The result is not surprising and needs a fair amount of work, but nevertheless it shows the model at work. We do not give any proof in the following, we only state what needs to be proved.

The goal is to prove the following:

$$\text{If } \models \{A\}\, c_1\, \{C\} \text{ and } \models \{C\}\, c_2\, \{B\} \text{ then } \models \{A\}\, c_1; c_2\, \{B\}$$

First one would need to obtain soundness of the transformations with respect to partial correctness.

Proposition 1.6.5. $\models \{A\}\, c\, \{B\}$ *if and only if* $\models [\![c, \alpha, \beta]\!] : [\![A, B, \alpha, \beta]\!]$.

To derive compositionality one would have to prove that parallel composition of transformed commands corresponds to sequencing of commands.

[1] By the automaton translation, this case cannot happen.

Proposition 1.6.6. $[\![c_1, \alpha, \gamma]\!] \parallel [\![c_2, \gamma, \beta]\!] \equiv [\![c_1; c_2, \alpha, \beta]\!]$

The last part would be to insert a (dualized) subcontract in order fulfill a larger contract.

Proposition 1.6.7. *If*

$$\models [\![c, \gamma, \beta]\!] : [\![C, B, \gamma, \beta]\!]$$

then

$$\models [\![c, \gamma, \beta]\!] : \overline{[\![A, C, \alpha, \gamma]\!]}, [\![A, B, \alpha, \beta]\!]$$

Now in order to derive compositionality of partial correctness, assume that

$$\models \{A\}\, c_1\, \{C\} \quad \text{and} \quad \models \{C\}\, c_2\, \{B\}$$

Then by soundness (Proposition 1.6.5) we would get that

$$\models [\![c_1, \alpha, \gamma]\!] : [\![A, C, \alpha, \gamma]\!] \quad \text{and} \quad \models [\![c_2, \gamma, \beta]\!] : [\![C, B, \gamma, \beta]\!]$$

and then by insertion of dualized subcontract (Proposition 1.6.7):

$$\models [\![c_1, \alpha, \gamma]\!] : [\![A, C, \alpha, \gamma]\!] \quad \text{and} \quad \models [\![c_2, \gamma, \beta]\!] : \overline{[\![A, C, \alpha, \gamma]\!]}, [\![A, B, \alpha, \beta]\!]$$

and then by Theorem 1.5.17 (compositionality of contract conformance) we get that

$$\models [\![c_1, \alpha, \gamma]\!] \parallel [\![c_2, \gamma, \beta]\!] : [\![A, B, \alpha, \beta]\!]$$

which by parallel composition and sequencing (Proposition 1.6.6) yields

$$\models [\![c_1; c_2, \alpha, \beta]\!] : [\![A, B, \alpha, \beta]\!]$$

and finally using soundness (in the reverse direction) the result will follow

$$\models \{A\}\, c_1; c_2\, \{B\}$$

1.6.4 Session Types

In this section we consider session types [28]. Instead of modeling the full system (with channel passing and dynamic channel creation), we concentrate on a smaller part, which we extend with timing constraints.

The fragment we are considering contains: receiving (?), sending (!), branching ($[\dots]$), selection ($\{\dots\}$) and recursion (μX). Only base values can be sent, not (session) channels. The regular session types are extended with an annotation $a \in \mathbb{N} \cup \{\infty\}$ which expresses that this part of the communication must take place before a time units. If $a = \infty$ then there are no

timing constraints – similar to ordinary session types. We use an ordering, $(\cdot < \cdot) \subseteq \mathbb{N} \cup \{\infty\} \times \mathbb{N} \cup \{\infty\}$, which is the standard order on $\mathbb{N}$ extended with $n < \infty$ for all $n \in \mathbb{N}$.

The types are the following:

$$\begin{aligned} A &::= n \mid \infty \\ B &::= \mathbf{Unit} \mid \mathbf{Int} \\ \underline{S} &::= X \mid S \\ S &::= ?^A B.\underline{S} \mid !^A B.\underline{S} \mid \\ &\quad [l_1 : \underline{S}_1, \ldots, l_n, \underline{S}_n]^A \mid \{l_1 : \underline{S}_1, \ldots, l_n, \underline{S}_n\}^A \mid \\ &\quad \mu X.S \mid \mathbf{End} \end{aligned}$$

where $\mathcal{L} \ni l$ is a countable set of labels. Note that we forbid infinite non-progressing loops – like for example $\mu X.X$ – in the syntax. We will write $[l_i : s_i]_I^a$ for $[l_1 : s_1, \ldots, l_n : s_n]^a$ with $I = \{1, \ldots, n\}$ and the same for $\{l_i : s_i\}_I^a$.

As an example, the following session type describes the protocol for a math server, which continuously accepts requests to either negate a number or add two numbers[2]. The client (from the viewpoint of whom the session type is described) can end communication by selecting the quit branch.

$$\mu X.\{\mathsf{neg} : !\mathbf{Int}.?\mathbf{Int}.X,\ \mathsf{add} : !\mathbf{Int}.!\mathbf{Int}.?\mathbf{Int}.X,\ \mathsf{quit} : \mathbf{End}\}$$

The dual session type (i.e., from the viewpoint of the server) is formed by interchanging sends and receives, and selection and branching, respectively

$$\mu X.[\mathsf{neg} : ?\mathbf{Int}.!\mathbf{Int}.X,\ \mathsf{add} : ?\mathbf{Int}.?\mathbf{Int}.!\mathbf{Int}.X,\ \mathsf{quit} : \mathbf{End}]$$

A type environment for a process is a finite mapping from channels (ranged over by α, β, γ) to session types.

$$\Sigma \ni \sigma : \mathcal{C} \to_{\text{fin}} S$$

The typing environment describes the protocol which should be followed when communicating on the channels of the typing environment (and hence the associated session types are from the viewpoint of the process).

Now it would be possible to specify a programming language and give typing rules using session types, but for brevity we omit this here. Instead we focus on capturing the intended meaning of the session types in our model. So

[2]Note that – unlike our contract model – session types cannot be used to describe that the result is indeed the negation and sum, respectively. This would require "dependent session types", which to the best of our knowledge do not exist.

we take a given type environment and show what it denotes in our model of contracts.

The idea behind the mapping is that we let the game state be the current session types, and when a legal action occurs on a channel, the corresponding session type is reduced (for instance if $\alpha : !^{27}\mathbf{Int}.s$ and 5 is sent on α, then in the next game state $\alpha : s$). Furthermore, the game state contains a mapping from channels to numbers, representing how long since the last communication on that channel. Lastly we have a special game state (`end`) which represents either contract violation or success (depending on the payoff under the transition to `end`).

To get the rule function, we check each channel to see if the correct sender is sending or whether a deadline has passed. If there is an error on a channel, the game transitions to the `end` state, otherwise the timing function is updated. The payoff is always 0, unless the process makes an error (where the payoff is -1). Hence well-typedness for session types corresponds to contract conformance.

The actual modeling is presented below. First we map the base types into sets, and define an action set for the channels:

$$\begin{aligned} \ulcorner \mathbf{Unit} \urcorner &= \{()\} \\ \ulcorner \mathbf{Int} \urcorner &= \mathbb{Z} \\ \mathcal{A} &= \ulcorner \mathbf{Unit} \urcorner \cup \ulcorner \mathbf{Int} \urcorner \cup \mathcal{L} \cup \{\varepsilon\} \end{aligned}$$

Given a type environment, σ, we define the contract $\ulcorner \sigma \urcorner$ by:

$$\ulcorner \sigma \urcorner = (C^\sigma, G, g_0^\sigma, \rho)$$

where

$$\begin{aligned} C^\sigma &= \{\alpha^+ \mid \alpha \in \mathrm{dom}(\sigma)\} \cup \{\alpha^- \mid \alpha \in \mathrm{dom}(\sigma)\} \\ \mathcal{A}_\alpha &= \mathcal{A} \text{ (for all } \alpha \in C^\sigma) \\ G &= \{\langle \sigma, t \rangle \mid \sigma \in \Sigma \wedge t : \mathrm{dom}(\sigma) \to \mathbb{N}\} \cup \{\mathtt{end}\} \\ g_0^\sigma &= \langle \sigma, t \rangle, \text{ where } t(\alpha) = 0, \text{ for all } \alpha \in C^\sigma \\ \rho(\langle \sigma, t \rangle, m) &= \begin{cases} (\mathtt{end}, -1) & \text{if } \exists \alpha.\ f(\sigma(\alpha), t(\alpha), m(\alpha^+), m(\alpha^-)) = \mathtt{err} \\ (\mathtt{end}, 0) & \text{if } \exists \alpha.\ f(\sigma(\alpha), t(\alpha), m(\alpha^+), m(\alpha^-)) = \mathtt{done} \\ (\langle \sigma', t' \rangle, 0) & \text{if } \forall \alpha.\ f(\sigma(\alpha), t(\alpha), m(\alpha^+), m(\alpha^-)) = (s_\alpha, n_\alpha) \wedge \\ & \qquad \sigma'(\alpha) = s_\alpha \wedge t'(\alpha) = n_\alpha \end{cases} \\ \rho(\mathtt{end}, m) &= (\mathtt{end}, 0) \end{aligned}$$

The auxiliary function $f : \ S \times \mathbb{N} \times \mathcal{A} \times \mathcal{A} \to (S \times \mathbb{N}) + \{\mathtt{err}, \mathtt{done}\}$, used

above, has the value $f(s,t,v_1,v_2)$ defined by cases on s:

$$s = ?^a b.s' : \begin{cases} \texttt{err} & t < a \wedge v_1 \neq \varepsilon & \text{(erroneous send)} \\ (s',0) & \text{if } t < a \wedge v_2 \in \ulcorner b \urcorner & \text{(receive)} \\ (?^a b.s', t+1) & \text{if } t < a \wedge v_2 = \varepsilon & \text{(no receive)} \\ \texttt{done} & \text{otherwise} & \text{(environment timeout)} \end{cases}$$

$$s = !^a b.s' : \begin{cases} \texttt{done} & t < a \wedge v_2 \neq \varepsilon & \text{(environment send)} \\ (s',0) & \text{if } t < a \wedge v_1 \in \ulcorner b \urcorner & \text{(send)} \\ (!^a b.s', t+1) & \text{if } t < a \wedge v_1 = \varepsilon & \text{(no send)} \\ \texttt{err} & \text{otherwise} & \text{(timeout)} \end{cases}$$

$$s = [l_i : s_i]^a_I : \begin{cases} \texttt{err} & t < a \wedge v_1 \neq \varepsilon \\ (s_i,0) & \text{if } t < a \wedge \exists j \in I.\ v_2 = l_j \\ ([l_i : s_i]^a_I.s, t+1) & \text{if } t < a \wedge v_2 = \varepsilon \\ \texttt{done} & \text{otherwise} \end{cases}$$

$$s = \{l_i : s_i\}^a_I : \begin{cases} \texttt{done} & t < a \wedge v_2 \neq \varepsilon \\ (s_i,0) & \text{if } t < a \wedge \exists j \in I.\ v_1 = l_j \\ (\{l_i : s_i\}^a_I.s, t+1) & \text{if } t < a \wedge v_2 = \varepsilon \\ \texttt{err} & \text{otherwise} \end{cases}$$

$$s = \mu X.s' : f(s'[\mu X.s'/X], t, v_1, v_2)$$

$$s = \mathbf{End} : \begin{cases} (\mathbf{End}, t+1) & \text{if } v_1 = v_2 = \varepsilon \\ \texttt{err} & v_1 \neq \varepsilon \\ \texttt{done} & v_2 \neq \varepsilon \end{cases}$$

Because channels in our model are unidirectional, we split each bidirectional channel into two channels in the modeling, the positive one being output from the process and the negative one being the input. We note that only the start state and the channels are dependent on the concrete σ, i.e., the rule function and the set of game states are the same for all typing environments. This is really no surprise, as session types are evaluated using a reduction semantics, hence ρ represents the reduction rules, and the game states represent the possible (residual) configurations/session types.

1.6.5 Quality of Service

In this last example, we illustrate how a contract can take into account that communication may be unreliable (e.g., packet loss). The example shows a series of contracts for gradually more unreliable channels. We do not wish to model completely unreliable channels, but instead channels which have some guaranteed throughput. The kind of unreliability we consider is packet loss, but it could just as well be manipulation of values.

The unreliable medium is modeled as a principal; in the context of IP communication, this could for instance be the ISP providing a Quality of Service (QoS). The contracts contain two channels; one for input of messages, and one for output (with potential loss). The contracts are binary; the only

payoffs are 0 and -1, with -1 meaning a breach of the QoS agreement. For reference we start with a contract for a reliable channel with no delay (all contracts are described from the viewpoint of the ISP):

```
contract cReliable
{
  in  a : int;
  out b : int;
  start wait();

  state wait()
  {
    when x on a do snd(x);
  }

  state snd(n : int)
  {
    when x on b do
      if x = n then wait() else stop() with -1;
    timeout 1 : stop() with -1;
  }
}
```

The next contract models a channel which has a delay of at most 10 time units:

```
contract cDelay
{
  in  a : int;
  out b : int;
  start wait();

  state wait()
  {
    when x on a do snd(x);
  }

  state snd(n : int)
  {
    when x on b do
      if x = n then wait() else stop() with -1;
    timeout 10 : stop() with -1;
  }
}
```

Now we model a channel which has a delay of at most 10, and can throw at most two consecutive packets away:

```
contract c2drop
{
  in  a : int;
  out b : int;
  start wait(2);

  state wait(p : int)
  {
    when x on a do snd(x,p);
  }

  state snd(n : int,p : int)
```

```
  {
    when x on b do
      if x = n then wait(2) else stop() with -1;
    timeout 10 :
      if p = 0 then stop() with -1 else wait(p - 1)
  }
}
```

The last model is a channel where – in a given time period of 100 time units (sliding window) – at most 10 packets sent may be dropped. This model corresponds to guaranteeing a throughput of 90% if the channel is fully utilized. To express this contract, we use the abstract contract model instead of the contract language:

$$
\begin{aligned}
c &= (\{\alpha, \beta\}, G, g_0, \rho) \\
\mathcal{A}_\alpha = \mathcal{A}_\beta &= \mathbb{Z} \cup \{\varepsilon\} \\
G &= \{\langle n, l\rangle \mid n \in \mathbb{N} \wedge l \in (\mathbb{N} \times \mathbb{Z} \times \{\bot, \top\})^*\} \cup \{\mathtt{stop}\} \\
g_0 &= \langle 0, \cdot \rangle \\
\rho(\langle n, l\rangle, m) &= \begin{cases} \varphi(n, l), & m(\alpha) = \varepsilon \wedge m(\beta) = \varepsilon \\ \varphi(n, (n, v, \bot) : l), & m(\alpha) = v \wedge m(\beta) = \varepsilon \\ \varphi(n, \mathrm{send}(l, v)), & m(\alpha) = \varepsilon \wedge m(\beta) = v \\ \varphi(n, (n, v_1, \bot) : \mathrm{send}(l, v_2)), & m(\alpha) = v_1 \wedge m(\beta) = v_2 \end{cases} \\
\rho(\mathtt{stop}, m) &= (\mathtt{stop}, 0)
\end{aligned}
$$

The semantics of "send" and φ are given below:

$$
\begin{aligned}
\mathrm{send}(\cdot, v) &= \cdot \\
\mathrm{send}((n, v, r) : l, v') &= \begin{cases} (n, v, \top) : l & \text{if } r = \bot \wedge v = v' \\ (n, v, r) : \mathrm{send}(l, v') & \text{otherwise} \end{cases} \\
\varphi(n, l) &= \begin{cases} (\mathtt{stop}, -1) & \text{if } |\{n_i \mid n_i \geq n - 100 \wedge r_i = \bot\}| > 10 \\ (\langle n + 1, l\rangle, 0) & \text{otherwise} \end{cases} \\
& \quad \text{where } l = (n_1, v_1, r_1) : \ldots : (n_m, v_m, r_m)
\end{aligned}
$$

The idea behind the model is that we record all incoming packets in the game state and whether they have been forwarded. The game state is a list of packets (integers) with a time stamp and a bit representing if that packet has been forwarded ($\top$ means forwarded, $\bot$ means not forwarded).

When an element is received, it is added to the list with the current time and forward bit $\bot$. When a packet is forwarded, the latest unforwarded packet with the same value gets updated (the bit is set to $\top$). The "send" function handles this update.

In each time step the number of unforwarded packets in the interval from the current time and 100 time units back are counted, and if the total is greater than 10, the QoS contract is breached. The check is handled by the φ function.

1.7 Summary & Related Work

1.7.1 Summary

In this chapter we have we have developed and described a model for extending the programming-by-contract paradigm to a distributed and concurrent environment. A main contribution of the chapter is the shift from cooperative, intra-company decomposition of a specification, to an adversarial model of composition with different parties. This shift has sparked a game-theoretic view of contracts, and through a generalized payoff measure, we are able to model for instance quality of service, degrees of fulfillment, local optimization, etc. The main model is based on I/O automata and contracts for these. The model is chosen for simplicity, as one of the future goals is to look into verification and certification of software for this model. But even though the model is simple, it is still very expressive. We have shown how to express other software specifications (for instance session types), and we believe that the model can – to some extent – be used for expressing business contracts and workflows as well.

1.7.2 Related Work

The amount of work in relation to design by contract, concurrency, and distributed environments is vast. In this section we describe several other contributions, and how they relate to/differ from our model.

As mentioned in the introduction of the chapter, the inspiration for our work is (classical) programming by contract for sequential programs. In this context, a program specification consists of pre- and postconditions:

$$\{A\}\, c\, \{B\}$$

The kind of specification above may seem unrelated to the notion of adversarial composition, underlying the nature of a two-person game with payoffs that we have considered. However, Hoare triple validity can itself be seen as a game between two players; the implementer of the program and an environment. The rules of the game are simple: whenever the environment can make a move (i.e., choose a store) such that A is satisfied in that store, then the move of the programmer – executing c – must be such that B is satisfied in the resulting store. If this is the case, c is a *winning strategy* – meaning exactly that the Hoare triple is valid.

The game-theoretic interpretation of specifications is also closely related to the theory of *game semantics* [2, 12]. In game semantics a program c again denotes a *strategy* for playing a game against an opponent. The game then

specifies a *type* (for instance **Nat** or **Nat** → **Nat**), and c is a winning strategy exactly when it is well typed (with the given type).

In object oriented programming[3] there has been some investigation of specifying behavior of distributed objects. Exton and Chen have discussed methods for specifying interfaces for remote method invocations [13]. As in our model, the internal state must be hidden in the interface. But this is from a code-encapsulation principle, rather than because of different administrative principals. Helm et al. have introduced a contract language (called Contracts [23]) for specifying behavior of compositions of objects. This both includes interface specification (variables, methods) but also *causal obligations* which make it possible to specify that a series of actions must be taken in response to some event. A concrete class can then conform with the contract if it implements the interface and satisfies the causal obligations. The method is tied into object oriented programming, and there is no account for distribution across different platforms.

Our model of communication – I/O automata – is quite different from the more traditional process calculi: we have already elaborated on these differences, but to sum up, our model is concerned with directed, point-to-point channels, rather than shared bidirectional channels; and perhaps more important; processes have no means of *refusing* input on a channel. These differences mean that contracts need only describe point-to-point, actual communication, and not shared communication with possible refusal of input.

One of the inspirations for our work is session types [28], which – like our model – is concerned with distributed communication. There are several key differences between session types and our approach. Being based on the π-calculus, the points about process calculi, mentioned above, also apply here. With session types it is not possible to specify absolute timing guarantees, which we argued is necessary due to the adversarial nature of distributed computing. Hence if a channel has session type "send integer", then it means: "If the channel is ever accessed, then it must be via the send of an integer". Thus by performing for instance an internal, infinite loop, the channel is never accessed, and therefore the contract is not violated. In session types it is also not possible to specify dependencies between received and sent values, nor is it possible to specify dependencies between different channels (i.e., requiring interleaving of messages on different channels). However, session types have the possibility of reasoning about dynamically created channels, which we have postponed to future work. For specific implementations and to verify/certify concrete programs, it would be interesting to look at (dependent) session types as a method for proving contract

[3]The original article on design by contract by Meyer [43] is based on the object oriented language Eiffel.

conformance in our model. In other words, the session type system would be used as a *sound* axiomatization for proving contract conformance (which is the approach taken in [31]).

Castagna et al. have used CCS for describing contracts for web services [11]: by using subtyping for contracts they formalize compatibility between a service and a client. They focus on subtyping and the work is therefore concerned with proving conformance, and not about describing behavior. A somewhat related approach is taken by Rajamani and Rehof, who type π-calculus processes via CCS processes [59]. Here an assume-guarantee reasoning principle is applied for modularized reasoning about processes in the π-calculus, which – as in our model – can be seen as generalized pre- and postconditions.

Other related work is in the area of compositional development methods in the presence of concurrency, where the focus is more on capturing behavior of concurrently running processes and not so much about distribution and responsibility. Jones has provided an overview of the approaches taken to compositional reasoning in concurrency [35], where he argues that there is still much research to be done if such approaches shall be applicable in practice. It is not the purpose of our work to try and solve this problem, but more to model it. Future work is devoted to looking at how to prove conformance, maybe with inspiration from the rely/guarantee conditions covered by Jones. Another approach to concurrency is taken by Hooman, who focuses on developing Hoare rules for real-time systems [29]. The approach starts with an abstract extensional model, which uses a dense time model, different form our discrete model. In his model, processes are allowed to produce an infinite amount of observable events in finite time. The model is instantiated to channel based communication through a set of axioms and Hooman presents a set of rules for a programming language to prove conformance.

1.7.3 Future work

The model presented in this chapter is *work in progress*. There are several places where future work can extend and improve our model. In this section we list possible directions for future work:

- Dynamic changes: in our model the communication topology is static. A main direction for future work is to look at how to extend the model with dynamic changes to the communication topology. This includes:
  - dynamically negotiated contracts,
  - dynamically created implementations, and

  - dynamically created principals.

- Verified/certified code: the second main direction of the work is to investigate whether the model can be used as a basis for verified or certified code [48].

- Refinement of time: in our model the time domain is fixed for the entire network. A direction for future work is to look at how to relax this condition, to make the model more realistic. One possibility is to assign different speeds to different links/channels, as in the IOTA model.

- Contract/automaton relationship: contracts and automata are very similar; it would be interesting to investigate this relationship.

- Contract subtyping: some contracts are weaker (impose less requirements, give more assumptions) than others. It would be interesting to consider a subtyping relation for contracts to formalize the strength of contracts. Back and von Wright have shown that contracts together with a *refinement relation* form a lattice [6]. The refinement relation (which is based on the *refinement calculus* by the same authors [5]) express exactly the intuitive idea of subcontract/subtype; it would therefore be interesting to investigate the possibilities of applying Back and von Wright's ideas to our work.

- Contract language: the language presented in Section 1.6.1 is far from an optimal language. Future work is to look at a more expressive language, or possible other languages for different contracts. It would be interesting to consider the possibilities for developing a *declarative* language, rather than the current imperative/state based language.

- Implementation: in order for our approach to be truly validated, we would – at some point – like to see an actual implementation of contracts and contract monitoring. One approach could be to have contract monitoring as an optional debugging facility, similar to runtime assertion checks in for instance JML [37]. (JML is a realization of programming/design by contract in Java.)

- Examples: bigger and more realistic examples for modeling should be investigated.

Chapter 2

Contracts in Enterprise Systems

2.1 Introduction and Motivation

This chapter is devoted to a survey on *contract support* in Enterprise Resource Planning (ERP) systems. The interest in electronic contract support has grown over the last decade: empirical studies conducted by the Aberdeen Group [53, 54] suggest that contract lifecycle management (CLM) will be a critical key to success for businesses in the near future. CLM is a broad term used to cover the activities of systematically and efficiently managing (1) contract creation, (2) contract negotiation, (3) contract approval, (4) contract execution, and (5) contract analysis. The last part involves maximizing financial and operational performance, and ensuring contract compliance.

In the studies [53] it is reported that around 80 percent of the enterprises (220 participants) are exercising only manual, or partially automated contract management activities. The implication for not automating CLM is a lower rate of compliant transactions (i.e., actions according to contract), and as a result of this, potential financial penalties: "the average savings of transactions that are compliant with contracts is 22%" [53, p. 1]. In accordance with these observations, the survey [54] shows that 58 percent of the enterprises (258 participants) have assigned high priority to automated CLM (in contrast, only 8 percent have assigned low priority). Hence the interest in efficient and reliable CLM is evidently present in the industry.

In this survey we will primarily consider business/commercial contracts[1], and hence our survey is not intended to include "general purpose" contracts (for instance employment contracts, non-disclosure agreements, etc.). The reason for this restriction is the scope of the framework project 3gERP [1] that our work is subject to. The scope of our survey has been further restricted, compared to the original project proposal [32]: rather than considering both contract- and workflow formalization, we focus on contract formalization only. The reason for this restriction is simply the surprisingly vast amount of work within the contract area, which has made it impossible to consider workflows as well.

Our survey is divided into two parts; the first part (Section 2.2) is a survey on existing approaches in the scientific community to contract formalization. In this part we have systematically explored the scientific contributions in the area, by searching through relevant on-line repositories, and by going through the bibliographies of cited, related work. The second part (Section 2.3) is a survey on approaches taken in commercial products, which has been conducted primarily by means of an on-line search. In this part we try to categorize the different features the individual products offer, which we will

[1] Whenever we write *business* or *commercial* contract, we refer to contracts involving exchange of goods/services between two (or more) parties.

later be able to use as a benchmark for evaluating our own, future approach. However, for almost all of the commercial products, a technical, in-depth description of the product is not available, hence we can only describe the *claimed* features of these products.

The studies conducted by the Aberdeen Group focus mostly on the potential winnings of using an automated CLM system (by means of a series of monetary figures on contract compliance; resources/hours spent on contract management; discounts and savings, etc. – we will not replicate the figures here). From the point of view of our survey, the interesting part in the Aberdeen studies is the list of "required actions", which are recommendations for enterprises seeking to utilize CLM. These recommendations serve as a list of requirements for (future) CLM systems; below we summarize the – in our opinion – most important of these requirements:

(R1) Establish standardized and formal contract management processes, including a standard language for contracts accessible via libraries and templates;

(R2) Clearly define protocols for the complete contracting process and contract administration (e.g., contract execution); and

(R3) Use reporting and analytic capabilities on contract data to gain competitive advantage.

In our opinion, the three points above capture the very essence of what a CLM system should support (R1+R2) and why CLM is of interest (R3). R1 is perhaps the most important requirement (and the one which will have main focus in this survey): by expressing contracts in a standard language – with agreed-upon formal semantics – a contract should unambiguously denote the set of rules, by which the contracting parties are supposed to act. Furthermore, the benefit of having an unambiguous semantics is the possibility to carry out the analyses of R3, without having to make qualified guesses as to what the contract means. Finally, having unambiguous contracts should make it possible to automatically assign *blame* in case the contract is breached.

Based on our agreement with the importance of the requirements above, we therefore evaluate the different approaches throughout the survey, with respect to these. Furthermore, we have included a sample sales contract in Appendix C, which we will use as a benchmark contract when evaluating the approaches with respect to requirement R1 (i.e., we show how/if the contract can be encoded). The sample contract is a template, we therefore consider a particular instance of it, which is included in full in Appendix D. In this instance we consider the third payment option of Paragraph 4 only (half payment upon receipt, with the remainder due within 30 days of

delivery). Furthermore, we extend the sample contract with a penalty clause (Paragraph 11). We include this paragraph in order to check for the ability to encode so-called *contrary-to-duty* (or *reparation*) obligations. Contrary-to-duty obligations have the consequence that violating a paragraph, need in fact not mean a breach of contract; instead a secondary paragraph is activated. Of course this paragraph may itself be violated, which will mean a breach of contract (unless the reparation paragraph has itself a reparation paragraph).

When we evaluate an approach with respect to the sample contract, we also consider if the language used for encoding the contract enjoys the *isomorphism principle*[2]. By the isomorphism principle is meant that the encoding is in a one-to-one correspondence with the informal contract; one paragraph in the paper contract corresponds to a separate component in the formalization, and any dependencies between paragraphs in the paper contract are also present between the corresponding components. The reason why the isomorphism principle is interesting is immediate: a (small) change in the paper contract will imply a (small) change in the formalized contract (and vice versa), and furthermore it makes it more feasible for domain experts to carry out the formalization directly. We refer to the masters thesis by Nielsen for more perspectives on the isomorphism principle in the context of value-added tax (VAT) formalization [49].

One aspect of contracting, namely electronic contract signing (in relation to R2), will be left out in our survey. We see this subject as orthogonal to the main focus of our survey (R1), and the theoretical topics related to this subject – for instance cryptography – are also rather distant from the topics of our survey.

[2]Following the principle of isomorphism introduced in the context of legal text formalization [7].

2.2 Contract Formalization

In this section we present a survey on the current status of research in the areas related to contract lifecycle management. Most of the related work in this area is referred to as *electronic contracting*, by which is meant formalization of (business) contracts in computer systems, to enable (automatic) validation, optimization, performance analysis, etc. The work we have found relevant within this area ranges from models based on deontic logic, over Prolog-based formalizations and process-oriented formalizations to more informal XML-based approaches.

We continuously evaluate the strengths and weaknesses of the individual approaches (with respect to the requirements R1-R3 from Section 2.1). The content of this section will provide valuable background information when we want – at a later stage – to design a domain specific language for modeling commercial contracts (Section 2.4).

2.2.1 Requirements for Contract Lifecycle Management

Unlike what we will consider in the following subsections, the first paper we consider is an overview paper (survey) by Tan et al., with the strikingly related title "A Survey of Electronic Contracting Related Developments" [65]. But unlike our survey, Tan et al. concentrate not on unambiguous formalization of contracts (R1); rather they focus on (business) requirements for electronic contracting/CLM systems (R2+R3). Furthermore, the aim of our survey is to give technical descriptions of the different approaches, which is not the aim of their survey.

The requirements gathered by Tan et al. – which are based on the Electronic Market Reference Model [61] – are used to evaluate various electronic contracting systems (mostly commercial, but also a few scientific), with the conclusion that no single system supports all requirements. We include the (in our opinion) most important requirements of Tan et al. below. We see these requirements as an elaboration of the requirements R1-R3 from the introduction, and therefore categorize each of Tan et al.'s requirements with respect to R1-R3:

1. Validation service (R3): a service for ensuring the validity of contracts (with respect to law). A prerequisite for this service is electronic access to national/international law;

2. Negotiation service (R1+R2): an environment for electronic contract negotiation. According to Tan et al., this service relies on a facility for contract drafting and version management. Furthermore it is suggested to use a protocol for contract negotiation (but no concrete protocol is suggested);

3. Monitoring service (R2): a service that observes the actions of contracting parties, and indicates non-performance of contracted obligations (breach of contract);

4. Repository service (R1): a store for predefined contract templates and contract instances;

5. Decision support (R3): a system that is able to: (i) interpret the meaning of legal clauses, and (ii) perform "what if analysis" based on input from the user (e.g., "do I violate the contract by postponing payment until next Friday?"); and

6. Communication infrastructure (R2): an authenticated, secure platform for contract execution. A reliable monitoring "bank" is needed as legal evidence in case of legal disputes between contracting parties.

As we shall see in the rest of this chapter, there seems to be a general agreement that the services/requirements listed above should be supported by CLM systems. We return to this list when evaluating commercial products in Section 2.3.

2.2.2 Deontic Logic

In this subsection we briefly introduce standard deontic logic (SDL), based on the descriptions in [41, 56]. SDL is a special kind of modal logic used for expressing obligatory and permissible statements, and hence an obvious candidate for modeling the dictating nature of contracts. Various work exists on expressing business contracts in deontic-like models – which we will return to later – this subsection serves as preliminary reading.

SDL is classical[3] propositional logic [30, Chapter 1] extended with two modalities: O (obligation) and P (permission). The formulae $\mathrm{O}\phi$ and $\mathrm{P}\phi$ should be read "it is obligatory that ϕ" and "it is permitted that ϕ", respectively. Consider the following simple example of an informal sales contract:

> The buyer is permitted to order from the seller. If the buyer orders from the seller, then the seller is obliged to deliver the goods to the buyer.

In SDL this sentence can be encoded as

$$(\mathrm{P}\mathit{order}) \wedge (\mathit{order} \rightarrow \mathrm{O}\mathit{deliver\_goods})$$

[3]Whenever we write "classical" in the context of logic we refer to classical reasoning (as opposed to intuitionistic reasoning).

where *order* and *deliver_goods* are propositional symbols denoting the actions "order" and "delivery of goods" respectively. Formally SDL can be generated by the grammar

$$\phi \quad ::= \quad \bot \mid p \mid \phi \rightarrow \phi \mid \mathrm{O}\phi$$

with the (usual) classical abbreviations

$$\neg\phi \equiv \phi \rightarrow \bot \qquad \phi \vee \psi \equiv \neg\phi \rightarrow \psi \qquad \phi \wedge \psi \equiv \neg(\neg\phi \vee \neg\psi)$$

and the dual relation between obligations and permissions; $\mathrm{P}\phi \equiv \neg\mathrm{O}\neg\phi$. Furthermore, a prohibition modality (F) can be expressed via $\mathrm{F}\phi \equiv \mathrm{O}\neg\phi$, which also relies on classical reasoning ("by not doing ϕ one does $\neg\phi$").

As is usual in modal logics, the semantics of SDL is given by means of *Kripke models*. A Kripke model, $\mathcal{M}$, for SDL is a triple

$$\mathcal{M} = \langle W, d, V \rangle$$

where W is a non-empty set of *worlds*, $d \subseteq W \times W$ is the deontic accessibility relation, and $V \subseteq \mathit{Prop} \times W$ is the valuation function for propositional symbols (i.e., $(p, w) \in V$ means that p holds in world w). Without getting into a philosophical discussion on the possible-worlds semantics of SDL, we mention the following "intuitive" interpretation based on [69]:

A world $w \in W$ represents a possible state, which need not be the actual state of affairs. The deontic accessibility relation, d, relates the possible worlds: whenever $(w, w') \in d$, w' *could* have been the actual state of affairs instead of w (perhaps surprisingly, d need not be symmetric nor reflexive). Therefore whenever ϕ is obligatory in a world w, then ϕ has to hold in *all* possible worlds (and dually for permissibility). One requirement is that the accessibility relation is *serial*, meaning that

$$\forall w_1 \in W. \exists w_2 \in W. (w_1, w_2) \in d$$

i.e., there are no "terminal" worlds.

The reason why we have quoted "intuitive" above, is that we do not find the *alternative world* formulation very intuitive: one might be tempted to think of the worlds as representing moments in time (where the deontic relation specifies the set of possible next states); in this interpretation $\mathrm{O}\phi$ would mean that ϕ should hold in the next moment in time. But this interpretation is not correct (there exist extensions of SDL to include temporal aspects for this purpose). One of the problems is that we have found no concrete instance of a Kripke model, and the work we will see later on, has no formal mapping into deontic logic. With this "disclaimer" in mind, we continue – for the sake of completeness – the introduction to SDL below.

A formula ϕ is said to be *satisfied* in the model $\mathcal{M} = \langle W, d, V \rangle$, written $\mathcal{M} \models \phi$, whenever $\mathcal{M} \models_w \phi$ for all $w \in W$. The latter relation is defined by structural induction on ϕ (note how the definition of satisfiability of $\mathrm{O}\phi$ reflects the informal possible-worlds definition above):

$$\begin{array}{lll} \mathcal{M} \not\models_w \bot & & \\ \mathcal{M} \models_w p & \text{iff} & (p, w) \in V \\ \mathcal{M} \models_w \phi \to \psi & \text{iff} & \mathcal{M} \models_w \psi \text{ whenever } \mathcal{M} \models_w \phi \\ \mathcal{M} \models_w \mathrm{O}\phi & \text{iff} & \mathcal{M} \models_{w'} \phi \text{ whenever } (w, w') \in d \end{array}$$

A formula ϕ is said to be valid, written $\models \phi$, whenever $\mathcal{M} \models \phi$ for all models $\mathcal{M}$. With these definitions, it can be shown that SDL is a *KD*-style modal logic [30, Chapter 5], meaning that the following holds:

$$\models \mathrm{O}(\phi \to \psi) \to \mathrm{O}\phi \to \mathrm{O}\psi \qquad (K)$$

$$\models \mathrm{O}\phi \to \neg\mathrm{O}\neg\phi \qquad (D)$$

The K-property is present in all normal modal logics, and it says that if a material conditional is obligatory, and its antecedent is obligatory, then so is its consequent. The D-property (D for deontic) says that there can be no conflicts (i.e., it is not possible for both ϕ and $\neg\phi$ to be obligatory). Since $\mathrm{P}\phi \equiv \neg\mathrm{O}\neg\phi$, an alternative reading of the D-property is that whenever ϕ is obligatory then ϕ is permitted (which one would expect). The validity of (D) is easily seen to rely on d being serial.

Besides from (K) and (D), a series of valid statements and derived rules can be shown, some important of which are:

$$\models \mathrm{O}(\phi \wedge \psi) \to \mathrm{O}\phi \wedge \mathrm{O}\psi \qquad (2.1)$$

$$\models \mathrm{O}\phi \wedge \mathrm{O}\psi \to \mathrm{O}(\phi \wedge \psi) \qquad (2.2)$$

$$\text{If } \models \phi \text{ then } \models \mathrm{O}\phi \qquad (2.3)$$

(2.1 - 2.2) state that O distributes over conjunction, and (2.3) states that if a formula ϕ is valid then it is also valid that ϕ is obligatory. In some sense, the Kripke-style semantics has been constructed so as to make the properties K, D and (2.1-2.3) hold. We note also that the converse of (2.3) does not hold:

$$\text{If } \models \mathrm{O}\phi \text{ then } \models \phi \qquad \text{(UNSOUND)}$$

It is easy to verify that the above does not hold (it would hold if all worlds were reachable from some other world, in which case d would be called *functional*). Intuitively it also makes sense: even though ϕ *should* hold, we cannot be sure that ϕ actually holds.

This concludes our brief introduction to deontic logic. We return to the subject in the following subsections, where examples of modeling contracts in deontic-like systems are presented.

2.2.3 A Logic Model for Electronic Contracting

The first approach we consider, by Lee, is – to the best of our knowledge – the first attempt to express business contracts in a formal language [38]. The motivation for Lee's work is the desire for unambiguous expression of contract terms, with the purpose of enabling various kinds of contract analyses (for instance detection of contradictory conditions, hypothetical/"what if" reasoning, etc.).

The approach taken by Lee is to express contracts as formulae in a formal logical model. The logical model contains elements of temporal logic, predicate logic and deontic logic. The following aspects are identified as being crucial for modeling contracts:

(1) Standard predicate logic: logical negation, conjunction, disjunction, and implication are used everywhere in contracts (albeit in an informal manner). Logical predicates can be used to encode sentences such as "if Jones delivers goods and Smith pays money then the contract is terminated" via $D(J,G) \wedge P(S,M) \rightarrow T(C)$;

(2) Relative temporal constraints: contracts often stipulate the *sequence* of actions to be carried out. To encompass this requirement, Lee suggest to use "logic of change", which introduces a new connective for referring to the *next* state in time (i.e., similar to the temporal connective "neXt" of LTL [30, Chapter 3]);

(3) Absolute temporal constraints: contracts often include absolute deadlines (for instance delivery of goods before a certain date). The suggested solution is to include a special temporal connective to accommodate such absolute constraints. Time is modeled discretely, with a rather coarse granularity (days);

(4) Deontic aspects: the need to express obligations/permissions is identified. The solution is to use (ideas from) deontic logic (Section 2.2.2); and

(5) Performative aspects: the last aspect considered is contract negotiation. Essentially it is suggested to have formalized protocols for contract negotiation (with messages such as "offer", "accept", "reject" and "counter-offer"), which – in some sense – are themselves contracts ("contract producing" contracts). The exact contract protocol will vary from situation to situation: for instance a contract offer may involve payment in some situations.

The presentation [38] does not give a full semantics for the proposed contract language. Furthermore, many of the ideas are used in later approaches

– which we will describe in greater detail in the following subsections – we therefore postpone evaluation of the ideas to these subsections. We do recognize, however, the importance of the aspects identified above.

2.2.4 HP Labs: e-contracts

We now consider the *e-contracts framework* [10] developed at HP Labs Bristol, UK. The e-contracts framework is a machine interpretable model for expressing business-to-business contracts. A contract is characterized as follows [10, p. 1]:

> "... contracts define rights and obligations of parties as well as conditions under which they arise and become discharged. The rights and obligations concern either states of the affairs or actions that should be carried out. Often contracts also specify secondary (reparation) obligations that come into force when a party does not carry out an obligation. The essence of contracts is the definition of commitment states that is imposed on contracting parties. These states come into force and become discharged as a result of actions that the parties carry out or as a result of an occurrence of an external event such as expiration of a deadline".

This definition captures very well the essence of a contract; a contract is a means for describing rights and obligations between parties, and – perhaps even more importantly – a contract may describe the (secondary) obligations that come into force when a party does not carry out its (primary) obligations (i.e., contrary-to-duty obligations).

An electronic contract (e-contract) consists of four parts:

(1) Contract identification number;

(2) Mapping between roles and legal entities (roles can be thought of as free variables in the contract body, and legal entities are the contracting parties);

(3) Contract validity period; and

(4) Contract body (behavioral specification): a set of normative statements describing the expected behavior of the various roles defined in the informative section.

Parts 1–3 are not that interesting with respect to our survey – though part 2 indicates that e-contracts are in fact templates, which can be instantiated

with actual parties. The interesting part is 4, which describes the obligations/rights in the contract. The normative statements are inspired by deontic logic (Section 2.2.2), but unfortunately there is no formal mapping of normative statements to formulae in deontic logic (no formal semantics is provided).

What follows is therefore our interpretation of the informal description [10]. Normative statements have the form

$$l : f \rightarrow D_{i_1,i_2}(a < T)$$

where l is a label, f is a predicate (which may refer to other statements via their labels), D is a deontic operator (either obligation (O), permission (P) or prohibition (F)), i_1 and i_2 are roles, a is the action to (not) perform and T is a deadline. The intuitive reading of a statement is:

> "When f holds, i_1 is obliged/permitted/prohibited by i_2 to achieve/perform a before T".

One question arises, when considering the deontic operators above: why are permissions mentioned explicitly? We know from deontic logic that $\mathrm{P}\phi \equiv \neg\mathrm{O}\neg\phi$, and since deontic logic is based on classical reasoning, the absence of an obligation $\mathrm{O}\neg\phi$ should imply that ϕ is permitted. Since this discussion is relevant for all of the deontic-based approaches, we postpone it to Section 2.4. (We also note that a prohibition from performing action a is equivalent to an obligation not to perform a, but normative statements do not have negation of actions, hence prohibition is included as a primitive.)

We illustrate the idea behind normative statements by means of an encoding of the sample sales contract in Appendix D.

Example 2.2.1 (Sales contract)**.** Consider the set of normative statements $\{l_3, l_{5.1}, l_{5.2}, l_7, l_{11}\}$ defined by:

$$\begin{aligned}
l_3 &: \mathit{init} \rightarrow O_{\text{seller,buyer}}(\mathit{deliver_goods} < \text{2009-09-02}) \\
l_{5.1} &: \mathit{fulfilled}(l_3) \rightarrow O_{\text{buyer,seller}}(\mathit{pay_first_half} < \mathit{delivery_date}) \\
l_{5.2} &: \mathit{fulfilled}(l_{5.1}) \rightarrow O_{\text{buyer,seller}}(\mathit{pay_second_half} < \mathit{delivery_date} + 30\ \mathit{days}) \\
l_7 &: \mathit{fulfilled}(l_3) \rightarrow P_{\text{buyer,seller}}(\mathit{claim_for_damages} < \mathit{delivery_date} + 14\ \mathit{days}) \\
l_{11} &: \mathit{not_fulfilled}(l_3) \rightarrow O_{\text{seller,buyer}}(\mathit{pay_penalty} < \text{2009-09-03})
\end{aligned}$$

E-contracts use propositional constants to denote (real world) actions. The list of symbols used in the contract above are *init* (marks the contract start), *deliver_goods* (seller delivers goods to buyer), *pay_first_half*/*pay_second_half* (buyer pays first/second half), *claim_for_damages* (buyer claims seller for damaged goods), and *pay_penalty* (seller pays penalty). Hence propositional symbols implicitly encode *who* performs the action, which is different from

the proposal of Section 2.2.3, where for instance *deliver_goods* would be encoded $D(S, B, g)$. The advantage of using propositional symbols is that the logical model need not be generalized to full predicate logic, while the disadvantage is the inability to reuse predicates (e.g., D above) and to quantify over (and refer to) specific individuals (e.g., g above). Another problem with the encoding above, is the fact that the actual values (for instance the book name and the price) are not mentioned explicitly – information which is necessary if contracts are to be *valuated* (cf. requirement R3).

The set of statements above is a one-to-one encoding of the paragraphs in the sample sales contract (Appendix D); statement l_i encodes Paragraph i (with the exception of Paragraph 5, where the third payment method is encoded via two individual statements $l_{5.1}$ and $l_{5.2}$). The variable *delivery_date* is *dependent* on the time the goods are delivered (i.e., the day where l_3 is fulfilled): In the original example [10, p. 3] a similar dependency is used, but the dependency is not clear from the set of statements (like above, in which a dependency to l_3 was expected).

The model of time is not known (discrete or continuous?), and e-contracts do not include "$\leq$" in normative statements. Hence when we write for instance *pay_first_half* $<$ *delivery_date* in $l_{5.1}$, then we should probably write *pay_first_half* $<$ *delivery_date* $+\,\delta$, for some suitable δ. (In l_3 this is no problem, as "on or before September 1st" can be achieved via "before September 2nd".)

The e-contracts approach has some good ideas: first of all contracts are formalized in a manner which resembles closely how informal contracts are typically structured (as witnessed by the example above). In other words, e-contracts enjoy the isomorphism principle mentioned in the introduction. However, the approach also lacks some important aspects – most importantly a formal semantics; for instance what exactly does *not_fulfilled* mean? Another question related to this discussion is whether satisfiability is decidable; it is easy to construct a non-satisfiable (invalid) contract:

$$l_1 : \mathit{true} \rightarrow O_{i_1,i_2}(a < t)$$
$$l_2 : \mathit{true} \rightarrow F_{i_1,i_2}(a < t)$$

The contract above stipulates that i_1 is both obliged to – and prohibited from – performing action a before time t, which is impossible to fulfill. In relation to contract analysis (requirement R3), at least it should be possible to detect unsatisfiability of contracts as the one above (in the example above it is obvious that the contract cannot be fulfilled, but it is easy to imagine more complex, unsatisfiable contracts that cannot readily be detected unsatisfiable).

So far we have evaluated the e-contract approach with respect to requirements R1 and R3. In relation to the requirement R2, the e-contracts framework introduces two protocols: the Contract Negotiation Protocol (CNP) and the Contract Fulfillment Protocol (CFP). CNP is only described very briefly; in the negotiation phase CNP is used for negotiating the final clauses (i.e., normative statements) of the contract – in other words the contract is constructed incrementally using CNP. CFP is used when a contract has been agreed upon, and subsequently has to be executed. CFP is based on the *speech act theory*[4] [62], which results in a protocol in which participants exchange messages (events), which will alter the state of the contract. However, as for the e-contract language, CFP is only introduced by means of a single example and hence the exact types of messages used in the protocol are not described (and the ones in the example are not explained). We include the example below (slightly modified to suit the contract of Example 2.2.1).

Example 2.2.2. Consider the contract from Example 2.2.1. A possible list of CFP messages exchanged in the context of this contract could be:

1. *inform*(buyer, seller, *accept norm*(buyer,$l_{5.2}$))
2. *request*(buyer, seller, *acknowledge norm executed*(buyer,$l_{5.2}$))
3. *inform*(seller, buyer, *acknowledge norm executed*(buyer,$l_{5.2}$))

First buyer informs seller that he intends to perform the action prescribed by $l_{5.2}$ (i.e., pay the second half for the order), secondly buyer asks (requests) seller to acknowledge that $l_{5.2}$ has been fulfilled (i.e., the order has been paid), and finally seller informs buyer that he has fulfilled $l_{5.2}$ (i.e., that seller has received her payment).

The example above suggests that CFP is a handshaking protocol, i.e., the parties agree upon what has happened (in the real world) by means of requests/acknowledgments. But the interesting question is how events (i.e., CFP-agreed-upon events) are matched against running contracts; certainly not all messages make sense for a contract. A so called "CFP Manager" is mentioned very briefly – which supposedly "validates the syntactic and semantic correctness of the exchanged messages" [10, p. 6] – but again no explanation is given, so it is not possible to elaborate further on this.

Based on the discussions above, we have summarized the pros/cons of the e-contracts approach in the table below.

[4]Very briefly put, the speech act theory takes the viewpoint that "by saying something, you do something". In the context of contract execution, the parties therefore *say* (inform) what they have done, in order to make sure that each participant has a synchronized view of the current contract state

Pros

- Business contracts are formalized in a small language (based on deontic logic)
- The contract language enjoys the isomorphism principle; paragraphs are encoded as normative statements (obligations, permissions and prohibitions)
- Support for contrary-to-duty obligations
- Explicitly defined protocols for contract negotiation and contract execution

Cons

- No formal semantics provided
- A consequence of the lacking semantics: it is not clear which dependencies are possible between normative statements
- A consequence of the lacking semantics: it is not clear whether it is possible to perform interesting analyses (e.g., check for satisfiability)
- It is not clear whether the language can encode a large class of business contracts (only one example provided)
- Values (for instance the price of the goods in a contract) are only encoded implicitly via propositional symbols; this may be problematic for contract analysis

2.2.5 Event-Condition-Action based Contracts

The next approach we consider is a model for business-to-business (B2B) contracts by Goodchild et al. [16]. Their model is based on the *event-condition-action* paradigm known from active databases [8, Chapter 9] (more on this below). Goodchild et al. recognizes a contract as [16, p. 1]:

> "A contract is a legally enforceable agreement in which two or more parties commit to certain obligations in return for certain rights".

In a more detailed description, a contract is identified with the following set of elements [16, p. 2]:

(1) The description of parties involved, including: names, addresses, roles, etc;

(2) The definition and interpretation of terms used in the contract;

(3) The jurisdiction under which the validity, correctness, and enforcement of the contract will operate;

(4) The duration and territory[5] of the contract, which defines the times and places at which the contract is in force;

(5) The nature of consideration, for instance, fees, services rendered, goods exchanged, rights granted, etc; and

(6) The obligations associated with each role, which is expressed in terms of the criteria over the considerations. This includes terms and conditions for invoicing and payment such as warranties, delivery, liability, rejection, termination and accounting provisions.

Most of the elements identified above, are present in the sample contract of Appendix D, with the exception of elements 2 and 3 (we believe the elements 2 and 3 may often not be mentioned explicitly in contracts; for instance a sales contract between two Danish companies will implicitly be subject to the Danish Sale of Goods Act). Elements 1, 4 and 5 mostly deal with "meta information" (i.e., parameters which do no affect the normative contents of the contract), with the exception of deadlines. The interesting element is 6, which is the actual obligations dictated by the contract (corresponding to the paragraphs 3, 5, 7, and 11 of Appendix D).

As in the e-contracts of HP Labs (Section 2.2.4), a contract consists of a set of normative statements, called *policies* (the obligations/rights of element 6 above). A policy specifies that a legal entity is either forbidden or obliged to perform actions under certain conditions – similar to deontic logic, but without the permissibility modality. The BNF grammar for policies is as follows (keywords are underlined; $[\cdot]$ denotes optionality, and $\cdot^*$ denotes zero or more occurrences):

Policy	::=	*VariableDeclaration**
		when *Condition*
		Action
		must [not] occur where *Condition*
		otherwise *Trigger*
Action	::=	action(*ActionName*, *Actor*, *Audience*, *Time*, *Body*)
Trigger	::=	trigger(*ActionName*, *Audience*, *Body*)

The grammar above is taken directly from the article by Goodchild et al. – unfortunately the categories *VariableDeclaration*, *Condition*, *ActionName*, *Actor*, *Audience*, *Time* and *Body* are not specified.

[5]Note that the territory of a contract refers to what geographical areas the contract covers. Whereas the jurisdiction of a contract refers to which location's laws the contract is subject to. For example, the territory of a contract may cover all trade between two parties in Brisbane, and the jurisdiction may be covered by the laws of the State of Queensland.

What is even more unfortunate, is the fact that no formal semantics is provided for the language – as for e-contracts (Section 2.2.4) only a single example is provided. Rather than presenting a formal semantics (e.g., mapping policies to formulae in a variant of deontic logic), it is shown how the language is represented in XML[6]. A simplified version of the example is included below.

Example 2.2.3. Below is a fragment of a sales contract:

```
P = Contract.Purchaser;
S = Contract.Supplier;
when Contract.State == 'initial'
   action(send_purchase_order, P, S, t, b)
must occur where
   t >= Contract.Start and
   t <= Contract.Start + 7 days and
   order_is_valid(b)
otherwise
   trigger(send_notice_of_breach, *,
           "Purchaser failed to send valid purchase order")
```

This policy involves two participants: a purchaser and a supplier. The policy uses the predefined variable "Contract", which (supposedly) refers to the contract that the policy is a part of (there is no clear definition of what a contract contains except a set of policies, but from the example it is clear that it contains other information – such as participants – as well). The event-condition-action paradigm is evident in the example: when the *event* "contract signing" takes place, the *action* "send purchase order" must occur with the *condition* "date of order between date of signing and 7 days after signing, and purchase order is valid" being fulfilled. If this policy is breached, a notice of contract breach is sent to all contract parties.

Another thing we notice in the example above, is that contracts contain *state*. There is no description of what the state may contain (and when/how it changes), but from the example above it is clear that the state can be used to control the flow of a contract. In Section 2.4 we will discuss some of the disadvantages of having such a state-based contract semantics.

With the intuition given above, we now sketch how the sample contract of Appendix D could be encoded in the B2B policy language. Unfortunately the grammar for the language is unknown, hence we use pseudo-notation (enclosed in $\{\cdot\}$) to represent the needed conditions.

[6]It seems to be a general tendency – in particular in industry – that an XML-based representation of data is by definition a formalization. In some respect this is valid, but an XML representation is only a static/structural description of data, not a formalization of what the data means, and in particular how it may be *executed*. An example of such a formalization is the Contract Expression Language [15].

Example 2.2.4 (Sales contract)**.** Below is how we expect the sales contract from Appendix D could be encoded in the language by Goodchild et al.:

```
S = Contract.Seller;
B = Contract.Buyer;

when Contract.State == 'initial'
   action(deliver_goods, S, B, t)
must occur where
  t <= 2009-09-01
otherwise
   trigger(send_notice_of_breach, *,
           "Seller failed to deliver goods")

when { delivery has occurred }
   action(pay_first_half, B, S, t)
must occur where
  { t same day as delivery date }
otherwise
   trigger(send_notice_of_breach, *,
           "Buyer failed to pay first half")

when { first half paid }
   action(pay_second_half, B, S, t)
must occur where
  { t is no later than 30 days of delivery }
otherwise
   trigger(send_notice_of_breach, *,
           "Buyer failed to pay second half")
```

The encoding above clearly lacks many details (which – as mentioned – is due to the lacking grammar). Even more problematic, two paragraphs are missing from the encoding above: buyer's right to claim for damages, and the penalty assigned to seller in case of failure to deliver. The first paragraph is not encodable in the language, simply because the permissibility modality is missing, and it cannot be achieved via dualization either (cf. Section 2.2.2 this would require an ability to negate policies). The second paragraph can perhaps be encoded via the `trigger` construct, but this construct is not covered in the paper [16].

Since many details of the contract language are left out in the presentation [16], it is hard to fully evaluate the model they have provided. What is positive, though, is the movement towards development of a compact language for expressing policies (c.f. requirement R1), which enjoys the isomorphism property (cf. Section 2.1). But in our opinion, the model still lacks a formal semantics, and furthermore it seems to express only a proper subset of the contracts expressible in the e-contracts framework described in Section 2.2.4 (c.f. the discussion on permissibility above).

The discussion has been related to requirement R1, i.e., development of a standard language for contracts. In relation to requirements R2 and R3,

Goodchild et al. have identified a set of *roles* associated with contract establishment and contract monitoring (execution). These roles (or functionalities) are as follows:

- Contract repository: a repository for storing standard contracts and standard clauses;
- Notary: a store for *signed instances* of contracts (i.e., running contracts);
- Contract monitor: on-line monitoring of contract instances, in particular detection of contract violations;
- Contract validator: ensuring the creation of legally valid contract instances only, by means of four aspects:
  - Competence: the parties must have proper capacity/competence;
  - Clarity: the contract must be *unambiguous*;
  - Legal purpose: the contract should not conflict with law; and
  - Consideration: the contract must clearly state what is exchanged between the parties
- Contract negotiator: Negotiation of contracts in the pre-contractual phase

The roles identified above are very similar to those identified in the survey of Section 2.2.1, hence we will not elaborate further on these roles here. Though we note that the contract validator role contains an interesting aspect, namely *clarity*: Goodchild et al. claim that clarity (unambiguity) is very difficult to verify by a computer. But the whole idea behind formalizing contracts is *exactly* that contracts denote – unambiguously – the rules to which the signatories are assigned. Hence by assigning a formal semantics, the clarity aspect becomes trivial (which is probably the reason why the clarity check is included, as no formal semantics is provided).

We sum up our discussion by listing the pros and cons below:

Pros

- A clear division of contracts into contract elements: participants, definitional terms, jurisdiction, duration, nature of consideration, and obligations
- A small language for expressing policies (enjoying the isomorphism property), based on the event-condition-action paradigm

- Identification of a set of *roles* needed for CLM (which is in agreement with requirements for CLM seen elsewhere)

Cons

- No clear definition of the contract language
- No formal semantics provided
- Only a single example of a formalized contract
- No built-in support for contrary-to-duty obligations (can perhaps be encoded)
- Seems to express only a proper subset of the contracts of the e-contracts framework (Section 2.2.4)

2.2.6 The Business Contract Language

We now consider the Business Contract Language (BCL) [39, 47], introduced by Milosevic et al. (with several follow-up papers, some of which we refer to later). The goal of BCL is to enable efficient *enterprise contract management*, which – according to the definition [47, p. 1] – seems to be another term for contract lifecycle management.

Milosevic et al. claim that most CLM systems today follow the traditional database approach of ERP systems: contracts are not explicitly represented in the system; they only exist implicitly by means of data spread across various tables in the database (see also Section 2.3.2).

The solution proposed by Milosevic et al. is to construct a domain-specific language (DSL) for expressing business contracts (c.f. requirement R1), which is based on event-based monitoring/execution of business activities (c.f. requirement R2). Briefly summarized, a BCL contract consists of a set of *roles* together with a set of *policies* – which is hence very similar to the e-contracts approach (Section 2.2.4) and the B2B contracts approach (Section 2.2.5). The roles define the parties involved in the contract, and the policies define the obligations/rights agreed upon by the parties.

A policy specifies either an obligation, a permission or a prohibition; to whom the policy applies (i.e., a role); and the temporal constraints that the policy is subject to. Thus again this approach is similar to those of Section 2.2.4 and Section 2.2.5, but the actual encoding is much closer to deontic logic (as we shall see below). The presentations of BCL in the seminal papers contain no formal semantics, and the BCL language is only presented fragment-wise by means of examples[7]. Governatori and Milosevic

[7] To our knowledge there is no complete specification of BCL publicly available.

seek to formalize BCL by mapping it to a fragment of deontic logic extended with contrary-to-duty obligations in later work [18, 19]. We base the detailed description of BCL below on this work (which unfortunately also lacks some formalism).

As mentioned above, BCL is designed with the purpose of enabling event-based contract monitoring. An *event*, $e \in E$, is defined to be

(1) An action performed by one of the signatories of the contract, or

(2) A temporal occurrence (e.g., the passing of a deadline), or

(3) A change in the contract state, or

(4) Contract violation.

(1) are "real world events", i.e., actions that are actually performed; (2) and (3) are "control events" used for executing the contract; and (4) are special events raised when policies are violated. In terms of deontic logic (this is *our* interpretation), events correspond to propositional symbols, whose truth value depend on what has actually happened. In other words, what happens in the real world is modeled by assigning truth values to the propositional symbols in the deontic model – or put more directly: the real world is represented by a Kripke world (c.f. Section 2.2.2). Hence events are represented in a manner similar to the e-contracts framework of Section 2.2.4.

An *event pattern*, $ep \in EP$, is

(1) A logical relation between events (e.g., $\neg e$, $e_1 \wedge e_2$, $e_1 \vee e_2$), or

(2) A temporal relation between events (e.g., e_1 *before* e_2), or

(3) A temporal constraint on event patterns (e.g., ep *before* t, where t is an instance in time)

From the last two possibilities above, it is evident that SDL is not a sufficient logical model, due to the temporal aspects. (We will return to this discussion later.)

Event patterns are used when defining policies – for instance a policy might require that a certain role is obliged to perform either the action represented by e_1 or the action represented by e_2, which is achieved via the event pattern $e_1 \vee e_2$. Policies contain the following elements:

(1) A policy name;

(2) A role (i.e., to whom does the policy apply);

(3) The modality of the policy: either an obligation, a permission or a prohibition;

(4) A trigger, which defines when the policy is active. A trigger consists of a set of event patterns;

(5) An optional guard which – like the trigger – specifies when the policy is active. Unlike a trigger, a guard can refer to the contract state and to other policies (i.e., violation of these); and

(6) The obliged/permitted/prohibited behavior dictated by the policy (an event pattern)

Before presenting the grammar for BCL policies, we note that the policy structure above resembles the informal structure of typical contracts (c.f. the paragraphs of Appendix D). This approach was also seen in e-contracts (Section 2.2.4) and the B2B contracts (Section 2.2.5), hence the pros/cons mentioned there apply here as well.

The BNF grammar for BCL contracts is summarized below (keywords are underlined; $[\cdot]$ denotes optionality, $\cdot^+$ denotes one or more occurrences, and $\cdot^*$ denotes zero or more occurrences).

$$
\begin{array}{lll}
\textit{EP} & ::= & \underline{\texttt{not}}\ E \mid E\ \underline{\texttt{and}}\ E \mid E\ \underline{\texttt{or}}\ E \mid E\ \underline{\texttt{before}}\ E \mid \textit{EP}\ \underline{\texttt{before}}\ T \\
\textit{Policy} & ::= & \underline{\texttt{Policy}}:\ \textit{Name} \\
 & & \underline{\texttt{Role}}:\ \textit{Role} \\
 & & \underline{\texttt{Modality}}:\ \underline{\texttt{Obligation}} \mid \underline{\texttt{Permission}} \mid \underline{\texttt{Prohibition}} \\
 & & \underline{\texttt{Trigger}}:\ \textit{EP}^+ \\
 & & [\underline{\texttt{Guard}}:\ \textit{StateExp} \mid \underline{\texttt{violated}}(\textit{Name})] \\
 & & \underline{\texttt{Behavior}}:\ \textit{EP} \\
\textit{Contract} & ::= & \textit{Policy}^*
\end{array}
$$

Example 2.2.5 (Sales contract)**.** In order to better understand the BCL language, we have encoded the sales contract in Appendix D below. For easy reference, we have labeled the BCL policies `P`i, where i refers to Paragraph i in the sample contract. The payment method of Paragraph 5 is encoded via two separate policies (`P`5.1 and `P`5.2).

The following events are used:

`init`	: A special event indicating contract initialization
`deliver_goods`	: Delivery of goods
`pay_first_half`	: First half payment for goods
`pay_second_half`	: Second half payment for goods
`claim_for_damages`	: Claim for damages
`pay_penalty`	: Seller's payment of penalty

As was the case for B2B contract (Section 2.2.5) and e-contracts (Section 2.2.4), the actual values/resources are not mentioned explicitly in the events above. For instance it is not evident that the `pay_first_half` event actually means payment of 150 DKK. The contract encoding is presented below:

```
Policy: P3
 Role: Seller
 Modality: Obligation
 Trigger: init
 Behavior: delivery_goods before 2009-09-01

Policy: P5.1
 Role: Buyer
 Modality: Obligation
 Trigger: deliver_goods
 Behavior: pay_first_half

Policy: P5.2
 Role: Buyer
 Modality: Obligation
 Trigger: pay_first_half
 Behavior: pay_second_half before 30 days

Policy: P7
 Role: Buyer
 Modality: Permission
 Trigger: deliver_goods
 Behavior: claim_for_damages before 14 days

Policy: P11
 Role: Seller
 Modality: Obligation
 Guard: violated(P3)
 Behavior: pay_penalty before 2009-09-02
```

As mentioned above, Governatori and Milosevic seek to formalize BCL by mapping it to an extended fragment of deontic logic, called Formal Contract Logic (FCL). But no semantics is provided for FCL, and only policies are mapped to FCL (not full contracts, which – despite our simplification in the grammar above – contain various other parts as well). FCL basically extends deontic logic with two features: contrary-to-duty obligations and modality annotations. The former is the "real" extension, the latter is just a matter of annotating modality operators with roles, e.g., "seller is obliged to ϕ" would be written $O_{\text{seller}}\phi$, rather than $O\phi$. However, the annotations make the modalities more clear, and they make it possible to assign blame for violations by inspecting the deontic modalities only, and not the propositional symbols used to represent events (in fact it need not be the one carrying out an action that is the one responsible for doing it). Similar annotations were seen in the e-contracts framework of Section 2.2.4.

Contrary-to-duty obligations are used to model policies that become active, when other policies are violated (c.f. the `violated` construct of BCL). To accommodate these kinds of policies, FCL introduces a new binary connective $\otimes$, where $\mathrm{O}\phi \otimes \mathrm{O}\psi$ is interpreted "$\mathrm{O}\psi$ is the reparation of the violation of $\mathrm{O}\phi$". FCL does not extend SDL to include $\otimes$ at arbitrary positions in formulae, instead FCL restricts the set of well-formed formulae by dividing syntactic categories into *literals*, *modal literals*, $\otimes$*-expressions* and *policies*. The complete FCL grammar is presented below (we omit the role annotations on deontic modalities):

l	::=	$p \mid \neg p$	(literals)
ml	::=	$\mathrm{O}l \mid \neg\mathrm{O}l \mid \mathrm{P}l \mid \neg\mathrm{P}l$	(modal literals)
$\otimes$-exp	::=	$ml \mid \mathrm{O}l_1 \otimes \cdots \otimes \mathrm{O}l_n \mid$	
		$\mathrm{O}l_1 \otimes \cdots \otimes \mathrm{O}l_n \otimes \mathrm{P}l_{n+1}$	($\otimes$-expressions)
ϕ	::=	$l \rightarrow \phi \mid ml \rightarrow \phi \mid \otimes$-$exp$	(policies)

We first note that the left operand of $\otimes$ is always an obligation: this is simply because only obligations can be breached – for instance it would not make sense to write $\mathrm{P}l_1 \otimes \mathrm{O}l_2$, since a permission cannot be violated. However, it does make sense to use permissions for reparations of violated obligations, which is the reason the construct $\mathrm{O}l_1 \otimes \cdots \otimes \mathrm{O}l_n \otimes \mathrm{P}l_{n+1}$ is included. We then note that policies in general have the form (we use t as an abbreviation for the union of the two categories l and ml)

$$t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_n \rightarrow \otimes\text{-}exp$$

Each t_i is the antecedent for the rule (or *trigger*, to use the terminology of BCL), which dictates when the $\otimes$-expression (behavior, BCL) becomes active.

As mentioned, there is no formal semantics for FCL, hence the mapping of BCL to FCL presented by Governatori and Milosevic does not really count as a semantics for BCL. Furthermore, this mapping is only sketched briefly, in particular the translation of temporal constraints in event patterns is not described, which is probably because FCL does not include temporal aspects. However, the FCL notation is closer to that of SDL, and it opens up for the possibility of extending the Kripke-style semantics of SDL to accommodate the new $\otimes$-connective. In fact work has been done to extend SDL with contrary-to-duty obligations [56], but it would be outside the scope of this survey to go further into detail[8].

[8] We do note, however, that an "obvious" candidate for expressing $\mathrm{O}\phi \otimes \mathrm{O}\psi$, namely $\mathrm{O}(\phi \vee \psi)$, does not give the desired effect (according to Governatori and Rotolo [21]), since it does not make it evident that ψ is a reparation clause for ϕ. Instead Governatori and Rotolo suggest the intuitive reading: "$[\mathrm{O}\phi \otimes \mathrm{O}\psi]$ corresponds to a disjunction $[\mathrm{O}(\phi \vee \psi)]$ where the order of disjuncts matters" [21, p. 6].

We return to FCL in Section 2.2.11, where a system for reasoning about contrary-to-duty obligations in FCL is presented.

So far the discussion has been focused on the primary requirement R1. In relation to requirement R2, Milosevic et al. describes the "Business Contract Architecture", which is a collection of modules constituting the CLM system. The modules are (more or less) identical to those of the B2B framework (Section 2.2.5) and hence also to those identified in the business requirement survey of Section 2.2.1. We will therefore not go into further detail with the overall CLM framework, but just take notice of the agreement – across the different approaches – as to which components a CLM system should support.

Finally in relation to requirement R3, the presentations we found on BCL has remarkably few examples of analyses performed on contracts (for instance contract satisfiability, as mentioned in earlier sections). One example though, is a paper by Governatori et al. [22], which addresses the important question of whether a given business process is *compliant* with a business contract (written in FCL). Essentially, a business process is characterized by the set of event patterns it generates (trace semantics), and compliance is matter of testing whether all event patterns "satisfy" the given FCL contract: but since there is no formal semantics for FCL, there are no results about contract compliance implying formal contract satisfaction in all possible executions. In particular Governatori et al. introduce *sequences of events*, which suggests a temporal ordering of events, but neither a semantics for the deontic aspects nor for the temporal aspects of FCL is given.

The notion of an event pattern "satisfying" an FCL policy is refined into *ideal*, *sub-ideal*, *non-ideal* and *irrelevant* situations, with the intuitive interpretation that a policy is either fulfilled by complying with the primary obligation, fulfilled by complying with one of the contrary-to-duty obligations, not fulfilled, or trivially fulfilled due to premises (triggers, BCL) not being met, respectively. That is, given an FCL policy (we only consider the case where the last reparation clause is an obligation):

$$t_1 \rightarrow \cdots \rightarrow t_n \rightarrow Ol_1 \otimes \cdots \otimes Ol_m$$

and a sequence of events S, the situation is either

- Ideal: if $t_1, \ldots, t_n$ and l_1 are satisfied in the model generated by S; or
- Sub-ideal: if $t_1, \ldots, t_n$ are satisfied, l_1 is not satisfied, but some l_i $(1 < i \leq m)$ is satisfied in the model generated by S; or
- Non-ideal: if $t_1, \ldots, t_n$ are satisfied and all l_i $(1 \leq i \leq m)$ are not satisfied in the model generated by S; or

- Irrelevant: if some t_i $(1 \leq i \leq n)$ is not satisfied in the model generated by S

(Above we refer to a notion of being "satisfied in the model generated by S", which we cannot account for formally, due to the lack of semantics.)

The division into ideal, sub-ideal, non-ideal and irrelevant situations above justifies the claim mentioned earlier that a reparation clause should not be encoded as just an alternative clause (i.e., $\mathrm{O}\phi \otimes \mathrm{O}\psi \stackrel{\mathrm{def}}{=} \mathrm{O}(\phi \vee \psi)$), since in such a case it would not be possible to distinguish the alternative clauses. However, in the context of business contracts, we are not *a priori* convinced that this distinction is justified: an "ideal" situation is not necessarily one in which all contracts are fulfilled via primary obligations, an ideal situation (for one of the parties) might be one in which some contract clauses are violated deliberately, because such a situation is more profitable overall (cf. Chapter 1). We do therefore not yet discard the idea of encoding reparation clauses in business contracts by means of disjunctions. (We will return to this discussion in Section 2.4.)

We end the discussion on BCL by summing up the pros and cons encountered during the presentation.

Pros

- A policy-based DSL for expressing contracts, which satisfies the isomorphism principle
- Focus on support for contrary-to-duty obligations
- Work towards giving BCL a semantics based on extended deontic logic
- Event-based execution of contracts
- Work towards analyzing compliance between business processes and contracts

Cons

- No proper formal semantics (FCL has no formal semantics, and it seems to be too inexpressive to encompass all of BCL; for instance temporal aspects – and arbitrary boolean connectives – in event patterns, and full contracts with state)
- Not clear which analyses can be performed on BCL contracts (for instance validity)
- Values/resources are not mentioned explicitly in events
- Not clear if full-blown deontic logic extended with contrary-to-duty obligations and temporal aspects will become too expressive (with respect to contract analysis)

2.2.7 FCML

Unlike what we have seen so far, the goal of the work by Peyton-Jones and Eber is not to express general, commercial contracts – instead the focus is on formalizing the subset of *financial* contracts [55]. We have chosen to include it here anyway, as many of the aspects and ideas are applicable also to commercial contracts (in fact the work presented in Section 2.2.8 is based on the work by Peyton-Jones and Eber). For easy reference we adopt the name FCML (Financial Contract Markup Language), introduced by Højsgaard [34].

To set the scene, we start with an example of a financial contract [55, p. 1] (we have indented the textual presentation for easier reading):

D_0 : The holder of this contract has the right to choose
on 30 June 2000 between:
  D_1 : Both of:
    D_{11} : Receive £100 on 29 Jan 2001.
    D_{12} : Pay £105 on 1 Feb 2002.
  D_2 : An option exercisable on 15 Dec 2000 to choose one of:
    D_{21} : Both of:
      D_{211} : Receive £100 on 29 Jan 2001.
      D_{212} : Pay £106 on 1 Feb 2002.
    D_{22} : Both of:
      D_{221} : Receive £100 on 29 Jan 2001.
      D_{222} : Pay £112 on 1 Feb 2003.

Each D_i represents – in principle – a contract by itself, hence D_0 is composed by subcontracts D_1 and D_2 (which are themselves composed by subcontracts). This is different from what we have seen so far: a business contract is usually divided into a set of atomic policies, with no explicit representation of the dependencies between the policies. For instance, in the sample contract of Appendix D, Paragraph 5 (payment) is contingent upon successful execution of Paragraph 3 (delivery), but this *sequencing* of paragraphs is not clear in the informal representation.

In FCML such dependencies are made explicit via a *compositional* model for expressing financial contracts, which has resulted in a small "contract combinator" language. This approach differs from how financial contracts are normally expressed (where terms like *swaps* and *zero-coupon discount bond* are used as atomic contracts), but the benefit of specifying such contracts in terms of contract combinators is a much easier task of providing a formal semantics[9] (and hence also contract analysis).

FCML contracts are always *bilateral*, i.e., contracts are between two parties: the holder of the contract (from the point of whom the contracts are

[9]This is well-known in the area of programming language semantics.

described) and the counter-party. This is also different from what we have seen so far, where contracts in general are allowed to have multiple participants. The benefit of considering bilateral contracts only has already been elaborated extensively in Chapter 1 (see for instance Example 1.4.7, which models a multi-party situation by means of bilateral contracts), but whether all business contracts can be expressed purely by means of bilateral contracts is yet to be investigated (Section 2.4).

As witnessed in the example contract above, the execution of a contract depends on choices by the contracting parties, but not all choices are necessarily up to the signatories to decide; for instance a contract might depend on the exchange rate of a particular currency (on a particular day). To accommodate such external factors, FCML introduces a notion of *observables*, which are time-varying values. One may think of observables as measures that can be *probed* at any instant in time; for instance the weather measured in Celsius degrees in Copenhagen.

The full language of FCML is shown below. We do not include the full definition of observables (the intuition above should be sufficient) – we use o as a general notation for observables. The comments in parentheses describe the rights/obligations associated with acquiring the corresponding contract:

c	::=	*zero*	(no rights/obligations)
	|	$one(k)$	(right to one unit of currency k)
	|	$give(c)$	(reverse the rights and obligations of c)
	|	$and(c_1, c_2)$	(immediately acquire both c_1 and c_2)
	|	$or(c_1, c_2)$	(immediately acquire either c_1 or c_2, but not both)
	|	$cond(o, c_1, c_2)$	(immediately acquire c_1 if o holds, otherwise c_2)
	|	$scale(o, c)$	(immediately acquire c where all amounts are scaled by o)
	|	$when(o, c)$	(immediately acquire c as soon as o holds)
	|	$anytime(o, c)$	(acquire c once, anytime o holds)
	|	$until(o, c)$	(immediately acquire c, but abandon c once o holds)

We immediately note the compactness of the language: only ten constructors are included (and in fact the *zero* construct should be encodable via scaling of the *one* contract with the 0-constant observable). In relation to requirement R1 we therefore see many strengths in FCML (with respect to modeling of financial contracts): Peyton-Jones and Eber also show how some of the normal contract templates (for instance zero-coupon discount bond) can be expressed in FCML, which witnesses its applicability. One weakness, however, is the lack of a formal execution semantics for FCML; Peyton-Jones and Eber suggest to construct an operational semantics for the evolution of contracts (for instance the contract $or(c_1, c_2)$ should evaluate to either c_1 or c_2 based on the choice of the holder), but this aspect is left as future work.

One interpretation, though, is that of a two-person game (in the context of game theory [42]); we briefly sketch such a game-theoretic semantics below (this is *our* interpretation), where the two players are referred to as Holder and Opponent, respectively:

$zero$:	The game has ended
$one(k)$:	Holder wins one unit of currency k from Opponent
$give(c)$:	Initiate game c where Holder and Opponent switch roles
$and(c_1, c_2)$:	Games c_1 and c_2 are initiated (in parallel)
$or(c_1, c_2)$:	Holder chooses one of c_1 and c_2, which is then initiated
$cond(o, c_1, c_2)$:	If o holds, game c_1 is initiated, otherwise game c_2 is initiated
$scale(o, c)$:	Initiate game c where all amounts are scaled by o
$when(o, c)$:	The game c is initiated as soon as o holds
$anytime(o, c)$:	Holder can initiate game c (once) anytime o holds
$until(o, c)$:	Game c is initiated, but immediately ended once o holds

An interesting aspect, which is revealed in the game-theoretic interpretation above, is the fact that contracts (games) are always *deterministic*. By deterministic we mean that it is always clear, who makes the next move. One might be tempted to think of $cond(o, c_1, c_2)$ as a nondeterministic choice between either of games c_1 and c_2, but at any point in time (which is what matters), it is always deterministic which of c_1 and c_2 that are chosen. (In the game-theoretic analogy, one can think of observables as asking an unbiased referee for how to progress.)

Having this kind of determinism means that more qualified mathematical/statistical analyses can be performed on the contracts of FCML (cf. requirement R3): for instance the expected value of $give(c)$ can be interpreted as the negation of the expected value of c, while the expected value of $or(c_1, c_2)$ can be interpreted as the maximum of the expected values of c_1 and c_2. And in the case of observables, statistical forecasts can be applied (where possible); for instance the exchange rate of a particular currency might be predictable to some extent. (For the expected values of the remaining constructs – which are less dependent of determinism – we refer to [55, Figure 4].)

Example 2.2.6. For the sake of illustration, we now show how the contract in the beginning of this subsection can be written in FCML. We introduce the following notational conveniences:

$$\text{receive\_on}(d, a) \stackrel{\text{def}}{=} \underline{\text{when}}(isdate(d), \underline{\text{scale}}(a, \underline{\text{one}}(GBP)))$$

$$\text{pay\_on}(d, a) \stackrel{\text{def}}{=} \underline{\text{when}}(isdate(d), \underline{\text{give}}(\underline{\text{scale}}(a, \underline{\text{one}}(GBP))))$$

where d is a date, a is an amount, *isdate* is a map from dates to observable truth values (with the idea that $isdate(d)$ holds exactly when the date is d), and GBP denotes the British currency. With these definitions the contract can be expressed very compactly:

```
when(2000-06-30,
     or(and(receive_on(2001-01-29,100),
                pay_on(2002-02-01,105)),
        when(2000-12-15,
             or(and(receive_on(2001-01-29,100),
```

```
            pay_on(2002-02-01,106)),
    and(receive_on(2001-01-29,100),
            pay_on(2003-02-01,112))))))
```

We will not try to make an encoding of the sales contract of Appendix D in FCML, as FCML is tailored to financial contracts. In the next subsection we will see an approach to extend FCML to model commercial contracts, where the encoding will be presented. We conclude the introduction to FCML by noting that FCML has no considerations with respect to requirement R2: the focus is purely on contract formalization (R1) and contract valuation/analysis (R3).

Pros

- Compositional specification of financial contracts, in a small contract combinator language
- Explicit use of observables for external (time-varying) parameters
- Game-theoretic interpretation possible (with deterministic choices)
- Well-suited for mathematical/statistical analyses

Cons

- No operational (execution) semantics
- Restricted to financial contracts

2.2.8 CCML & POETS

Following the last section's discussion on FCML, we now proceed to the extension of FCML to commercial contracts, called CCML (Commercial Contract Markup Language) [4]. The language is introduced by Andersen et al. as a candidate for contract support in ERP systems, and is later used as a core component in the *process-oriented event-driven transaction system* (POETS) [24].

CCML contracts govern exchange of resources (money, goods and services) between multiple parties (based on McCarthy's REA (*Resources, Events and Agents*) accounting model [40]). The extension with respect to FCML is (1) exchange of resources rather than money only, (2) multiple parties, and (3) view-independence. Resources are defined in the context of POETS as being either a unit resource (for instance a car) or a compound resource (for instance one car and four winter wheels). Parties define the participants in

the contract, and following the REA ontology they are referred to as *agents*. View-independence simply means that a contract is not described from the viewpoint of a particular agent (as was the case for FCML).

Like FCML, CCML contracts are constructed compositionally via a set of contract combinators. The grammar for CCML is presented below (the comments in parentheses describe the informal obligations denoted by each construct):

p	::=	letrec $\{f_i[\vec{X}_i] = c_i\}_{i=1}^m$ in c	(obligations of c, in the context of contract templates $\{f_i[\vec{X}_i] = c_i\}_{i=1}^m$)
c	::=	Success	(no obligations)
	|	Failure	(breached contract)
	|	$c_1 + c_2$	(either obligations c_1 or obligations c_2)
	|	$c_1 \parallel c_2$	(both obligations c_1 and obligations c_2)
	|	$c_1; c_2$	(first obligations c_1 then obligations c_2)
	|	$f(\vec{a})$	(obligations of c_i with $\vec{a}$ substituted for $\vec{X}_i$, where $f[\vec{X}_i] = c_i$ is in the context)
	|	transmit$(A_1, A_2, R, T\|P).c$	(first obligation for agent A_1 to transfer resource R to agent A_2, at a point in time T such that predicate P is satisfied. Then obligations of c)

As mentioned, the comments in parentheses are only informal, but both a denotational- and an operational semantics are provided for CCML. This is hence the first approach we have considered, in which a formal semantics of contract execution has been presented (cf. R1 and R2). We will not duplicate the semantics here, but only mention that the denotational semantics is *trace based*, meaning that a contract essentially denotes a set of *accepting* event traces, where an event takes the form

$$\text{transmit}(a_1, a_2, r, t)$$

(Meaning that agent a_1 transmitted resource r to agent a_2 at time t.)

The operational semantics is defined as a labeled transition system, where $c \xrightarrow{e} c'$ means that contract c *evolves* to contract c' under event e – i.e., when event e occurs, c' will be the *residual* obligations with respect to the original contract c (as was suggested for FCML).

We now consider the CCML language in more detail: the Success contract is self-explanatory, and for the transmit construction we note that it is the binding constructor for the variables A_1, A_2, R and T (with scope in P and c). Furthermore the predicate language for P is not fixed, but it is expected to contain the usual logical connectives and temporal relations. The final thing we note about transmit is the fact that it is always the first agent, A_1, that is the one responsible for the transmission of resource R to agent A_2. Hence it is not possible to specify that some third agent, A_3, is in fact the

one responsible for the transmit (which is possible in BCL, cf. the discussion on annotated deontic operators).

As mentioned earlier, one of the ideas behind CCML is view-independence; unfortunately this means that contracts are not deterministic, with respect to choices and blame-assignment. For instance, what does the simple contract "Failure" mean? Supposedly it means breach of contract, but there is no indication of *who* has breached the contract. Another problem is the nondeterministic choice; consider the simple contract:

$$\text{transmit}(A_1, A_2, R, T|P).c + \text{transmit}(A_2, A_1, R, T|P).c$$

where $P \equiv A_1 = a_1 \wedge A_2 = a_2 \wedge R = r \wedge t \leq T$ and $c \equiv$ Success.

Informally this contract says that either agent a_1 must transmit resource r to agent a_2, on or before T, or vice versa. But what if neither a_1 nor a_2 transmit the resource – who is then to blame? This problem is not present in FCML, as it is always the task of one of the agents to chose *which* branch of the choice that is active[10] (we referred to this property as deterministic contract execution).

Using parallel composition, it is possible to write a similar contract, in which it is also impossible to blame a unique agent (just replace the choice operator with the parallel operator in the example above). One way of dealing with this problem is by insisting on a particular sequencing instead of parallelism, or by thinking of $c_1 \| c_2$ as simply two individual contracts; the latter approach is possible since there are no synchronization primitives between parallel contracts (as in process calculi), hence c_1 and c_2 cannot affect each other.

The last aspect in CCML is contract templates (for lack of a better name we have called the syntactic category p since we think of it as a *program* with the templates being *procedures*). Templates are very interesting, since they are allowed to be mutually recursive (hence the "letrec" keyword); this means that contracts can be infinite (which we have not seen so far):

```
letrec
{
 f[x] = transmit(A1,A2,R,T | A1=a1 and A2=a2 and R=r and T<=x).
        f(x + 1 day)
}
in
f(2009-09-01)
```

The contract above says that agent a_1 is obliged to transfer resource r to agent a_2 *each day* starting from September 1st 2009 (we assume the term

[10] A similar approach is taken by Back and von Wright [6], who use contracts to describe the intended behavior of a collection of agents (but in a computational context). Here all obligations – in particular choices – are annotated with agents, making it explicit who is the one "in charge".

language includes basic date arithmetic). As is well-known from programming languages, (unbounded) recursion is a very powerful tool, which comes at the cost of reduced automatic reasoning (in this case contract analysis, cf. requirement R3). Nissen [50] hence shows that *contract equivalence* is in general undecidable, and that *contract failure* (i.e., equivalence with the Failure contract) is also undecidable when considering a small predicate language (Presburger arithmetic extended with inequalities). The latter is particular problematic, since this means that it is impossible to decide whether a (running) contract is in fact breached (provided the predicate language is sufficiently expressive).

Example 2.2.7 (Sales contract)**.** We now illustrate an encoding of the template contract (Appendix D) in CCML. Following the original CCML article we will use

$$\text{transmit}(a_1, a_2, r, T|P)$$

as an abbreviation for

$$\text{transmit}(A_1, A_2, R, T|P \wedge A_1 = a_1 \wedge A_2 = a_2 \wedge R = r)$$

We already mentioned in Section 2.2.7 that policy-based (informal) contracts do not always mention the implicit assumptions concerning sequencing of obligations. For instance buyer's obligation to pay for goods is not supposed to be active before seller has actually delivered the goods. In CCML such dependencies are made explicit, and we end up with the following encoding:

```
letrec
{
 claimfordamages[seller,buyer,goods,notice,deadline] =
   transmit(buyer, seller, notice, T | T <= deadline and
                                       #(goods,broken,T)).Success
   +
   Success

 sale[seller,buyer,goods,payment,penalty,d1,d2,d3,d4,notice] =
   (transmit(seller, buyer, goods, T | T <= d1).
    transmit(buyer, seller, payment/2, T' | T' = T).
    (claimfordamages(seller, buyer, goods, notice, T + d2)
     ||
     transmit(buyer, seller, payment/2, T'' | T'' <= T + d3)))
   +
   transmit(seller, buyer, penalty, T | T <= d1 + d4).
   Success
}
in
 sale("YourBooks.com", "Tom Hvitved",
      "Introducing Game Theory and Its Applications",
      300 DKK, 30 DKK, 2009-09-01, 14 days, 30 days, 1 day,
      "Claim for damages notice")
```

In the formalization above, we have utilized parallel composition to encode that buyer's claim for damages may occur simultaneously while the second

payment is yet to be paid. The encoding shows that the informal contract perhaps lacks some aspects; for instance what are the obligations on seller if buyer makes a claim (in the encoding above there are no obligations)? And what happens with the second payment if buyer claims for damages before it has been paid (in the encoding above buyer is still obliged to pay the second half)?

In the `claimfordamages` template we use c + Succes to encode c as an optional contract, i.e., seller can either choose to claim for damages or not. CCML uses a notion of observables similar to FCML, which is used in the expression `#(goods,broken,T)`: intuitively this expression denotes the value of the property "broken" of the resource "goods" at time "T" (i.e., whether the goods are broken at time T).

The contrary-to-duty paragraph (Paragraph 11) is encoded as a choice: either seller must transmit the goods, or a penalty will be assigned. Hence primary and secondary objectives cannot be distinguished (unlike earlier approaches); and as mentioned earlier, we do not *a priori* see this as a problem (we postpone this discussion to Section 2.4).

We conclude the example by noting that CCML mentions values/resources explicitly in the contract (unlike the logic based approaches seen earlier); the benefit of this is that contract analyses can be based on the actual goods/services mentioned in the contract – for instance contract valuation will not be possible without such information.

Even though CCML is capable of handling multiple parties in contracts, we have not found any examples that illustrate the need for this capability. In the POETS article an encoding of a multiparty situation has been included [24, p. 396]:

```
Sale[vendor, customer, resource, pinfo as (p, t), deadline] =
  transmit(vendor, customer, resource, T | T <= deadline) ||
  (inform (vendor, customer, (resource, pinfo), T').
   (transmit (TaxAuth, vendor, -t(resource), _ ) ||
    transmit (customer, vendor, (p + t)(resource), T''
              | T'' <= T' + 8 days)))
```

First of all CCML is extended with an "inform" construct, which emphasizes that the transfer is of pure information (as opposed to linear resources, which cannot be duplicated). The informal reading of the contract is [24, p. 396]: "the first transmit expresses an obligation on the vendor to deliver the resource to the customer by the given deadline. The vendor must also send an invoice to the customer, which then results in an obligation by the tax authorities to collect the VAT amount for the invoiced resources and by the customer to pay the vendor the agreed-upon price, plus VAT".

This example includes a third agent: the tax authorities. However, we do not really see this as a contract; for instance, buyer should not be bothered

with the fact that vendor plans to pay VAT to the tax authorities. A more realistic formalization would (in our opinion) be a separation of the contract into two bilateral contracts: one between the vendor and the customer (the sales part), and one between the vendor and the tax authorities (the VAT part). We think of the `Sale` template above more as a business process, describing how the vendor plans to carry out sales, rather than a contract. We postpone the discussion of whether bilateral contracts are in general sufficient to Section 2.4.

The discussion so far has been concentrated on CCML/POETS with respect to requirements R1 (contract formalization) and R3 (contract analysis). We mentioned that CCML uses an event-based reduction semantics for contract execution (R2), and the POETS architecture is based on this idea: contracts written in CCML describe the *external* interface to the ERP system, i.e., the interface between the business and its "customers". Contracts are executed by matching events with the current set of active contracts, and if the event matches a contract, it is recorded in the database, and the relevant contract is rewritten (according to the operational semantics) to represent the residual obligations. The lifecycle of a contract in POETS is very simple: (1) the contract is *started*, meaning that the agents of the contract have agreed upon the contract; (2) the contract is *running*, meaning that the contract is executed according to the trace semantics mentioned above; and (3) the contract is *ended*, either because there are no obligations left, or if the contract is breached. POETS does not model the contract negotiation phase.

The event-based CLM solution proposed in POETS was also proposed in the BCL framework (Section 2.2.6), however due to the formal reduction semantics of CCML, the proposal of POETS is more concrete. We conclude this subsection by mentioning that POETS proposes support for contract analysis via a domain specific language, which can query the state of CCML contracts, and calculate for instance a task list of outstanding obligations. Furthermore, the reduction semantics of CCML should make it possible to perform "what if" analyses as well, by querying the residual contracts resulting from a hypothetical event. We refer to [24, Fig. 9] for an overview of the POETS architecture, which summarizes our textual presentation.

Pros

- Compositional specification of contracts (and contract templates), in a small declarative language with formal semantics
- Reduction semantics makes ongoing contract analysis – and "what if" analysis – possible
- Sequential dependencies are made explicit

- Domain-specific description of events based on the REA ontology, where values/resources are mentioned explicitly

Cons

- Contract formalization does not enjoy the isomorphism principle
- CCML is perhaps too expressive with respect to contract analyses (unbounded recursion)
- Nondeterministic contracts (with respect to blame and contract alternatives)
- Perhaps more flavor of business processes rather than contracts
- No modeling of contract negotiation

2.2.9 IST Contract Project

The IST Contract Project[11] is a research project funded by the European Commission. The project aims to cover both theoretical and practical aspects of:

(1) Specification of electronic business-to-business interactions in terms of contracts,

(2) Dynamic establishment and management of contracts at runtime in a digital business environment,

(3) Application of formal verification techniques to collections of contracts in a digital business environment, and

(4) Application of monitoring techniques to contract implementation in order to help provide the basis for business confidence in e-Business infrastructures.

The goal of the IST Contract Project is in other words to create the theoretical and practical foundations for a CLM system; requirement R1 covers goal 1, requirement R2 covers goals 2 and 4, and requirement R3 covers goal 3. At the time of writing, the IST Contract Project has yet to come up with a concrete model for expressing contracts; the current status is a sketch of a formal model [51], which we describe briefly below.

As in many of the previous approaches, a contract is identified with a set of normative statements, which specify what the agent(s) of the contract may or must perform. The purpose of the formalization is to encode these

[11] `http://www.ist-contract.org`

norms, and as we have seen earlier, the following requirements are identified: (1) norms must be monitorable in order to detect fulfillment/violation, (2) norms must be formally verifiable, and (3) norms must support contrary-to-duty obligations.

We will not include the grammar for contracts here – and hence no encoding of the sample contract in Appendix D – since the proposal is very similar to the other paragraph based formalizations we have seen so far. Norms contain five components: the norm type, the activation condition, the norm goal, an expiration condition, and a norm target. The norm type corresponds to a deontic modality; either an obligation or a permission. The activation condition specifies when the norm is active. The norm goal specifies the action that may/must be performed. The expiration condition is an explicit specification that the norm has become inactive. Finally the norm target represents the agent(s) to whom the norm applies. Hence with the exception of the expiration part, a norm is similar to the event-condition-action based policies of Section 2.2.5. An operational semantics for the language is sketched, which is based on a notion of *normative states*. Intuitively, a normative state divides the set of norms into those that are active, those that are inactive, and those that are expired (which are different from the inactive norms, as they cannot become active). When a normative state NS "steps" to new normative state NS', then the three classes of norms are updated according to the actions that have been performed. Unfortunately, there is no formal account for this stepping relation, since many of the details underlying it are left out. Due to the similarities with earlier approaches, and the fact that we found nothing conceptually new in the IST Project proposal, we will not conclude with a list of pros/cons; however, we do mention that an implementation of the contract platform exists[12], but we have not had the time to investigate the features of this implementation further.

2.2.10 The Contract Language $\mathcal{CL}$

The contract language $\mathcal{CL}$ [57] is introduced by Prisacariu and Schneider. Like earlier approaches covered in our survey, $\mathcal{CL}$ is a logic-based encoding of contracts, relying on deontic modalities. But unlike the previous approaches, a formal mapping of $\mathcal{CL}$ to an extended version of the propositional μ-calculus [36], called $\mathcal{C}\mu$, is provided. $\mathcal{CL}$ includes obligations (O), permissions (P) and prohibitions (F), and the deontic modalities can only be applied to *actions* (i.e., things that can be performed). The reason for this restriction is a claim that contracts are about describing what may/must/-must not be *performed* (which is referred to as *ought-to-do*), and not what

[12] `http://ist-contract.sourceforge.net/`

may/must/must not be the *state of affairs* (*ought-to-be*). *A priori* we find this argument valid, however it should be investigated if this claim holds for a set of representative business contracts (Section 2.4).

An atomic action, a, encodes a real-world event (for instance seller delivering goods to buyer), and is hence similar to the propositional encodings seen earlier. Atomic actions can be composed in parallel ($a \,\&\, b$), meaning that a and b occur simultaneously. (General) actions can be composed sequentially ($\alpha\beta$) and disjunctively ($\alpha + \beta$), meaning first α then β, and either α or β, respectively.

Prisacariu and Schneider argue that certain properties on the deontic modalities are desired, these include:

- $\neg\mathrm{O}(\alpha)$, i.e., it is *not* obligatory to perform α, does not occur in business contracts,
- $\neg\mathrm{P}(\alpha)$ should be written $\mathrm{F}(\alpha)$, and
- $\neg\mathrm{F}(\alpha)$ should be written $\mathrm{P}(\alpha)$.

Hence negation of deontic operators is not needed. The authors also argue that only certain connectives (action compositions) between (under) deontic operators make sense, we refer to [57, Section 2.2] for this discussion.

Besides actions, which are always performed by some agent, $\mathcal{CL}$ includes *assertions*, which correspond to the observables of FCML and CCML (i.e., assertions represent "facts"). We include a simplified version of the grammar for $\mathcal{CL}$ below; the simplification is with respect to disjunction of deontic modalities, which we leave out (see [57, Definition 1] for the complete grammar):

α	::=	a	(atomic action)
	|	$a_1 \,\&\, a_2$	(simultaneous atomic actions)
	|	$\alpha_1\alpha_2$	(action sequencing)
	|	$\alpha_1 + \alpha_2$	(action alternation)
Contract	::=	$D; C$	
C	::=	ϕ	(assertion)
	|	$\mathrm{O}(\alpha)$	(obligation to perform α)
	|	$\mathrm{P}(\alpha)$	(permission to perform α)
	|	$\mathrm{F}(\delta)$	(prohibition from performing δ, where δ is a $+$-free action)
	|	$C_1 \wedge C_2$	(both C_1 and C_2 must hold)
	|	$[\alpha]C$	(if α is performed, then C must hold)
	|	$\langle\alpha\rangle C$	(it must be possible to perform α, after which C must hold)
	|	$C_1 \,\mathcal{U}\, C_2$	(C_1 must hold until C_2 holds)
	|	$\bigcirc C$	(C must hold in the next moment)

The exact format of D is not presented, but intuitively it contains definitions of actions and assertions. Alternation (+) under prohibitions are not allowed (cf. the discussion above). $[\alpha]C$ and $\langle\alpha\rangle C$ are used to express conditions on actions (see the descriptions above), and $C_1 \,\mathcal{U}\, C_2$ and $\bigcirc C$ are the usual temporal connectives.

As mentioned in the beginning, $\mathcal{CL}$ is assigned a formal semantics by mapping it to an extension of propositional μ-calculus, called $\mathcal{C}\mu$. The semantics of $\mathcal{C}\mu$ is state-based, meaning that each formula denotes a set of states, in which the formula holds. The relation between states is dependent of the set of actions that occur, meaning that the relation is essentially a labeled transition system; $s \xrightarrow{\alpha} t$. Hence for instance the formula $[\alpha]\phi$ holds in state s exactly when ϕ holds in state t (provided that $s \xrightarrow{\alpha} t$). We will not go into detail with $\mathcal{C}\mu$, but we briefly consider how the deontic operators are modeled: a simple obligation $\mathrm{O}(a)$ in $\mathcal{CL}$ is modeled as a formula $\langle a\rangle O_a$ in $\mathcal{C}\mu$, where

> "[The propositional constant O_a is] interpreted ... as a set of states where the constant proposition holds. The intuition of the obligation constants is that when the system is in a state s and by action a it gets to a state t where O_a holds then we may conclude that in the state s the system has the obligation to execute action a" [57, p. 181].

Unfortunately there is no mention of how these propositional constants are assigned values, or when (and how) the transition relation – which is used to decide in which states a formula holds – is defined.

$\mathcal{CL}$ does not support real-time constraints (deadlines), hence in the encoding of the sample contract (Appendix D), the temporal aspects are restricted to the sequence in which the events should occur. $\mathcal{CL}$ does support the $\mathcal{U}$ntil operator, which means that *liveness* properties such as "buyer must pay eventually" can be expressed. However, we are not sure whether such guarantees are relevant in contracts, where obligations are typically time-bounded (i.e., all violations of an obligation ought to be detectable in finite time). (In fact it seems that the temporal connectives $\bigcirc$ and $\mathcal{U}$ have been removed in a more recent presentation [14].)

Example 2.2.8 (Sales contract)**.** As mentioned above, the encoding of the sample contract does not take the temporal aspects into account (for instance the deadline for delivery). We use the same names for actions as in Example 2.2.5, and we furthermore annotate each clause with a reference to

the corresponding paragraph of Appendix D:

$$\begin{aligned} p_3 &: \mathrm{O}(\mathit{deliver\_goods}) \wedge \\ p_{5.1} &: [\mathit{deliver\_goods}]\mathrm{O}(\mathit{pay\_first\_half}) \wedge \\ p_{5.2} &: [\mathit{pay\_first\_half}]\mathrm{O}(\mathit{pay\_second\_half}) \wedge \\ p_7 &: [\mathit{deliver\_goods}]\mathrm{P}(\mathit{claim\_for\_damages}) \wedge \\ p_{11} &: [\overline{\mathit{deliver\_goods}}]\mathrm{O}(\mathit{pay\_penalty}) \end{aligned}$$

The encoding above uses the *action negation* operator $\overline{\alpha}$, which denotes "... the action given by all the immediate traces that take us outside the trace of α" [57, p. 179]. (The definition of $\overline{\alpha}$ is not presented in [57].)

Hence a contrary-to-duty obligation is encoded by matching all actions that do not satisfy the primary obligation, i.e., $\mathrm{O}(\alpha) \wedge [\overline{\alpha}]C$ means that C is a reparation obligation for the obligation to perform α.

We conclude our brief introduction to $\mathcal{CL}$ by mentioning that work is currently being done to try and utilize *model-checking* technologies for analysis of contracts written in $\mathcal{C}\mu$ (and therefore $\mathcal{CL}$) [58]. The reason why model-checking is possible (and an obvious candidate for contract analysis, cf. requirement R3), is the fact that the semantic model is based on an extension of the μ-calculus, for which model checkers already exist. The close connection to the μ-calculus can also be seen in the structure of the $\mathcal{CL}$ language, which does not resemble so much the structure of informal contracts, as seen in earlier approaches.

Finally we mention that it has recently come to our attention that an alternative treatment of the $\mathcal{CL}$ semantics has been presented [14], where $\mathcal{CL}$ is given a trace semantics, rather than a mapping to $\mathcal{C}\mu$. Unfortunately we have not had the time to analyze this semantics further, so we postpone that to future work (related to requirement R1).

Pros

- A small language for expressing ought-to-do obligations, permissions and prohibitions
- Analysis of which connectives are needed between and under deontic operators
- Proposes satisfiability via model-checking

Cons

- No real-time aspects

- No contract negotiation
- No contract monitoring
- (Not clear whether the semantic model faithfully interprets the deontic modalities)

2.2.11 RuleML for Business Contracts

RuleML [60] is an XML-based family of languages for expressing *rules*, with the purpose of making them exchangeable between different systems. RuleML divides the set of rules into two categories: *reaction rules* and *derivation rules*. The former are event-condition-action based specifications, and the latter are used for describing facts (and as part of these, integrity constraints). RuleML does not include deontic and defeasible (the ability for one rule to be overruled by another rule or fact) aspects, and is hence not directly aimed at describing business contracts. To this end, Governatori extends RuleML to cover such notions [17]; an extension which is later referred to as DR-CONTRACT [20]. RuleML is "semantically neutral" [17, p. 20], meaning that there is no semantics for execution of RuleML specifications. Hence the actual formalization in RuleML does not provide much new to our survey (it is merely a structuring of contract clauses); but what *is* interesting, is the introduction of the *Defeasible Deontic Logic of Violation* (DDLV) [20], which we consider below.

DDLV contains four different kinds of knowledge: facts, strict rules, defeasible rules, and a superiority relation (the former two correspond to the derivation rules of RuleML). Facts represent indisputable knowledge, i.e., knowledge that cannot at a later stage be overruled. For instance in the sample contract of Appendix D it is a fact that the price of the book is 300 DKK, which is modeled as a predicate:

$$Price(Book, 300)$$

Strict rules are used for concluding new facts; when the antecedents for a strict rule are known to be facts, then so is the conclusion. For instance a contract might include the following definition of premium customers:

$$TotalExpensesAtLeast(X, 5000) \rightarrow PremiumCustomer(X)$$

which states that customers who have spent at least 5000 DKK are premium customers. Defeasible rules are rules, which may be overruled by some other (defeasible) rule, hence the conclusion of a defeasible rule may be invalid under certain conditions. For instance the contract of Appendix D could have said that the price of the book was 300 DKK, unless the customer was

a premium customer (5% discount). This would be written in DDLV as

$$r : \top \Rightarrow \mathit{Price}(\mathit{Book}, 300)$$
$$r' : \mathit{PremiumCustomer}(X) \Rightarrow \mathit{Price}(\mathit{Book}, 285)$$

But how do we know which rule overrules the other? This is the content of the last knowledge type, the superiority relation, which would relate $r' > r$ (meaning that rule r' overrules rule r).

In accordance with the two type of "facts" above (indisputable facts and defeasible facts), DDLV introduces a set of inference rules (but written in a somewhat non-standard way) for deriving these two kinds of facts. The interesting consequence is that conflicts can be detected; in particular *prima facie* conflicts, i.e., conflicts that have resulted from an underspecification of the superiority relation. For instance if it was not specified that $r' > r$ or $r > r'$ in the rules above, then a conflict would be detected.

After describing the defeasible parts of DDLV, Governatori introduces deontic modalities to DDLV, in a manner equivalent to the system of Formal Contract Logic (FCL, also by Governatori) seen in Section 2.2.6 – we will therefore not replicate the description here. But another interesting topic is discussed, namely *contract normalization*. By contract normalization is meant a transformation of DDLV contracts into equivalent contracts in *normal form* (we will not give the complete definition here). The intuition behind this transformation is that the set of rules that define a contract, may contain implicit reparation clauses, as well as clauses that are *subsumed* by other clauses. An example illustrating the first case is as follows:

$$r_1 : \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{deliver\_goods}) \otimes \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{pay\_penalty})$$
$$r_2 : \neg \mathit{deliver\_goods}, \neg \mathit{pay\_penalty} \Rightarrow \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{pay\_large\_penalty})$$

r_1 is the usual delivery clause from Appendix D, with the contrary-to-duty obligation that seller has to pay a penalty. The second rule says that if seller has neither delivered nor paid the penalty, then seller has to pay an even bigger penalty. But then the two rules can be *merged* into a single rule, expressing that the obligation to pay a large penalty is a contrary-to-duty obligation for r_1:

$$r_{1+2} : \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{deliver\_goods}) \otimes \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{pay\_penalty}) \otimes \mathrm{O}_{\mathrm{Seller,Buyer}}(\mathit{pay\_large\_penalty})$$

(In the example above it is readily seen that r_2 represents a contrary-to-duty obligation for r_1, but one can imagine less obvious, implicit reparation clauses.)

An example illustrating subsumption is if the contract includes both rules r_1 and r_{1+2}. Then the normative content of r_1 is subsumed by the content

of r_{1+2}, and r_1 can therefore be removed. A normal form is achieved by applying the two procedures of *merging* and *subsumption* (again we remind the reader that we have only hinted at the intuition behind these two procedures), until a fixed-point is reached. (Such a fixed-point always exists according to Governatori and Pham, and it is unique [20].)

There are several benefits of being able to convert contracts to unique normal forms: first of all it is satisfactory from a theoretical viewpoint, that equivalence of contracts is decidable (unlike CCML, Section 2.2.8). From a more practical viewpoint it means that reparation clauses in contracts can easily be identified (they will always occur after an $\otimes$-operator), hence the domain experts can easily check if the formalization contains the intended (reparation) obligations (R1). Also contract analysis can benefit from normal forms, by restricting attention to these forms only; for instance calculation of a list of outstanding obligations can be performed by considering the heads of $\otimes$-chains only (R3).

The remaining aspects of the DR-CONTRACT framework is a mapping of DDLV contracts in normal form to an extended version of RuleML; but as mentioned in the beginning, this is merely an XML representation of (more or less BCL) contracts, so we will not elaborate further on this (nor show how the sample contract can be encoded in the extended version of RuleML).

Pros

- Explicit modeling of the definitional terms in contracts, utilizing defeasible reasoning
- Reasoning about contracts via conversion to normal form

Cons

- Is conceptually BCL/FCL ("nothing new" on the language side)
- RuleML has no execution semantics

2.2.12 INCAS

The last paper we consider is by Tan and Thoen, who introduce a Prolog-based expert system called INCAS (INCoterms Advise System) [64]. This system is quite different from what we have seen so far, as the focus is not on actual contract formalization, but on contract *querying* (i.e., contract analysis, cf. R3). By this we mean that INCAS is a system for asking certain questions about what can be concluded from a contract, based on a set of premises (for instance who bears the risk for damage of goods under certain conditions?).

INCAS is tailored to the domain of trade contracts, i.e., contracts that stipulate the conditions under which a buyer and a seller trade goods. To this end, INCAS is based on *Incoterms*, which is a set of 13 predefined trade agreements, defined by the International Chamber of Commerce[13]. The Incoterms are not suitable for encoding of the sample contract, we will hence not try to do so. An Incoterm specifies the obligations of the buyer and the seller; for instance the FOB ("Free On Board") Incoterm specifies that the seller has fulfilled the obligation to deliver goods, when the goods have crossed the rail of the ship at the port of delivery, and the buyer is responsible for clearing the goods for export.

The interesting aspect of INCAS, is the utilization of Prolog to achieve defeasible reasoning (as in DR-CONTRACT, Section 2.2.11). Tan and Thoen describe an Incoterm example, where the buyer is liable for damage to the traded goods when the goods have left the port of origin. But this rule is overruled if the seller has not packaged the goods adequately. (This is somewhat related to the concept of contrary-to-duty obligations seen earlier.)

We will not go through all details of the implementation in Prolog, but briefly sketch some of the ideas. INCAS defines three kinds of predicates for dealing with the different kinds of obligations found in Incoterms; one of these is obligation to perform a certain action, which is denoted by the Prolog predicate `oblige_act(Action, Party, Contract, Term)`. One example of a Prolog rule for this predicate is:

```
oblige_act(export,buyer,contract,Term)
   :- in_list(Term,["EXW","FAS","DEQ"]).
```

which specifies that the buyer is obliged to clear the goods for export in the contract, if the Incoterm is either EXW ("Ex Works"), FAS ("Free Alongside Ship") or DEQ ("Delivered Ex Quay"). INCAS introduces a notion of *events*, which describe actions that have been performed, or facts that have emerged. When events are added to the Prolog knowledge base, new things may be concluded from the contracts in the system. Intuitively, if an event

```
performed(export,buyer,"20090901-01").
```

is added to the system, then the buyer has cleared the goods for export in the contract with identifier "20090901-01", and is hence no longer obliged to do so. Tan and Thoen do not describe the interplay between `performed` events and the `oblige_act` predicate, and since Prolog does not allow negation of goals, i.e.,

```
not(oblige_act(Action,Party,Contract,_))
   :- performed(Action,Party,Contract).
```

[13] `http://www.iccwbo.org/incoterms/`

we believe that all `oblige_act` rules have to explicitly include a subgoal that the obligation has not actually been performed, i.e.,

```
oblige_act(export,buyer,contract,Term)
   :- in_list(Term,["EXW","FAS","DEQ"]),
      not(performed(export,buyer,contract).
```

The actual formalizations of the Incoterms in INCAS are a bit more complicated than the above, but not conceptually different; hence the overall idea is a goal-based description of obligations (cf. R1), which can be compared with the event-condition-action driven contract formalizations of Section 2.2.5. Defeasible reasoning is achieved via Prolog's "negation as failure" principle, i.e., if Prolog fails to prove that a predicate P holds, then it is automatically assumed that $\neg P$ holds. This principle is utilized in INCAS by introducing an `exception` predicate, so for instance the obligation to clear goods for export above can be restated to:

```
oblige_act(export,buyer,contract,Term)
   :- in_list(Term,["EXW","FAS","DEQ"]),
      not(performed(export,buyer,contract),
      not(exception(export,buyer,contract,_)).

exception(export,buyer,contract,export_ban)
   :- fact(export_ban, seller, contract).
```

The above states that buyer is obliged to clear goods for export, unless an exception to this rule is active. Such an exception occurs for instance if there is an export ban for the goods in the contract (`fact` refers to the observable events mentioned earlier).

The negation-as-failure principle (also known as the "closed world assumption") is interesting, since it means that only events that actually take place need be described; this discussion is related the question of whether things that are not explicitly prohibited are necessarily permitted (and vice versa) – a discussion which we will return to in Section 2.4.

Pros

- Utilizes Prolog semantics for contract analysis (outstanding obligations)
- Explicit use of closed-world assumption

Cons

- Restricted to Incoterms
- No temporal aspects

2.3 Commercial Products

In this section we present a list of commercial products for contract lifecycle management. We have not been able to obtain technical details for any of the products, nor have we had the time to investigate possibilities for obtaining a copy of the software products. We hence list the features that these products *claim* to support, but without going into detail about how it is supported. We will not devote a subsection to each of the products, as they have many features in common; rather we describe them all in a single subsection, and provide a matrix comparison of the different products.

We conclude with a subsection demonstrating how commercial contracts – and therefore the workflows for fulfilling them – are encoded implicitly in a state-of-the-art ERP system, and analyze the problems associated with such an encoding.

2.3.1 Contract Lifecycle Management Systems

The purpose of this section is to analyze the features of commercial CLM systems. Our analysis is based on a sample of 14 software products, which has resulted in a comprehensive list of CLM features. The commercial products we have considered are (we omit trademark notices):

- Blueridge Software: Contract Assistant[14] (CA)
- CobbleStone Systems: ContractInsight[15] (CI)
- Moai: CompleteSource Contract Management[16] (CS)
- Eceteon: Contraxx[17] (CX)
- Emptoris: Contract Management Solutions[18] (EM)
- Great Minds Software: Contract Advantage[19] (GM)
- IntelliSoft Group: IntelliContract[20] (IC)
- Ketera: Contract Management[21] (KE)
- Open Text: Contract Management[22] (OT)
- 8over8: ProCon Contract Management[23] (PC)
- SAP: SAP CLM[24] (SA)

[14] http://www.blueridgesoftware.bz
[15] http://www.cobblestonesystems.com
[16] http://www.moai.com
[17] http://www.ecteon.com
[18] http://www.emptoris.com
[19] http://www.greatminds-software.com
[20] http://www.intellisoftgroup.com
[21] http://www.ketera.com
[22] http://www.opentext.com
[23] http://www.8over8.com
[24] http://www.sap.com

– StatsLog Software Corporation: StatsLog[25] (SL)
– Procuri: TotalContracts[26] (TC)
– Upside Software: UpsideContract[27] (UC)

For easy reference, we have labeled each product with a two-letter code, e.g., "KE" refers to Ketera's Contract Management software. Based on the descriptions found at the websites of the commercial products above, we have gathered a list of CLM features below. We have not included "technology related features", such as the database on which a system runs, or the particular technology used for the user interface; the list intentionally only includes "CLM features". The CLM features are (in no particular order):

(a) Centralized contract repository for storing pending, running, and finished contracts;

(b) Possibility of restricting access to contracts/data in the CLM system (user security);

(c) E-mail notifications/alarms/alerts and runtime monitoring of contracts;

(d) Reporting and analytics;

(e) Search-capabilities;

(f) Template-based contract creation;

(g) Compositional construction of contracts from other subcontracts;

(h) Workflows for contract approval/review;

(i) Contract compliance and adherence to business standards;

(j) Contract negotiation;

(k) Task-list with outstanding obligations/rights;

(l) Versioning system;

(m) Full auditing trail (cf. the *Sarbanes-Oxley Act*[28]);

(n) Integration with ERP system(s); and

(o) Secure messaging in contract execution/collaboration.

[25] http://www.statslog.com
[26] http://www.procuri.com
[27] http://www.upsidesoft.com
[28] http://www.sarbanes-oxley.com

We notice that the features above have many similarities with the requirements identified by Tan et al. in Section 2.2.1[29]; however none of the CLM systems we have considered have support for legal validation of contracts (Tan et al.'s first requirement). The features above also contain aspects that are not covered by Tan et al., these include; restricted access (b), workflow for approval/review (h), full auditing trail (m), and integration with ERP systems (n). The most notable feature missing from Tan et al.'s survey is the auditing trail (m): presumably this is because the Sarbanes-Oxley Act (2002) was not in existence at the time of Tan et al.'s survey.

In relation to requirement R1 from the introduction of this chapter, we have found no evidence that any of the CLM products above use a domain-specific language for full formalization of contracts. In most cases contracts seem to be represented as electronic copies of the paper contracts (for instance in Microsoft Word format); extended with various meta data for monitoring (for instance expiration date, date of delivery, responsible agents, etc.).

We conclude this subsection with a matrix comparison of the CLM products above (Figure 2.1). The matrix illustrates where we found the features (a)-(o), and hence who claims to support them. The semantics of a "✓" is that the product claims to include that feature, and a missing "✓" means that the feature is not mentioned explicitly in the product description (though it may actually be supported).

	CA	CI	CS	CX	EM	GM	IC	KE	ME	OT	PC	SA	SL	TC	UC
a	✓			✓	✓	✓	✓	✓		✓	✓	✓		✓	
b	✓							✓		✓	✓				
c	✓	✓				✓	✓	✓			✓	✓		✓	✓
d	✓			✓	✓	✓	✓	✓		✓	✓			✓	
e	✓	✓	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓
f		✓			✓	✓	✓				✓	✓	✓	✓	✓
g		✓													✓
h		✓				✓	✓			✓	✓	✓		✓	✓
i			✓	✓	✓			✓		✓		✓			✓
j			✓	✓											
k										✓					✓
l		✓					✓					✓		✓	✓
m							✓	✓			✓	✓		✓	
n										✓		✓		✓	
o											✓				

Figure 2.1: Features comparison matrix (CLM products horizontally, features vertically).

[29]Requirement 2 (negotiation service): {j,l}; Requirement 3 (monitoring service): {c}; Requirement 4 (repository service): {a, e, f, g}; Requirement 5 (decision support): {d, k, i}; Requirement 6 (communication infrastructure): {o}.

2.3.2 Microsoft Dynamics NAV

The final commercial product we consider is the ERP system Microsoft Dynamics NAV[30]. NAV is an interesting ERP system with respect to our research project, 3gERP [1], since it is one the major products in the market for small- and medium sized enterprises. The purpose of this subsection is to demonstrate how business contracts are handled in Microsoft Dynamics NAV; or more precisely, how the workflows for fulfilling the business contracts are implemented. (In Section 2.4 we discuss the connection between contracts and workflows in more detail.)

Microsoft Dynamics NAV is based on a classical three-tier architecture; a database layer for storage of "business information", an application layer for "business logic", and a presentation layer. The database layer consists of a relational database (Microsoft SQL Server), and the business logic of the application layer is written in the (imperative) domain-specific language "C/AL".

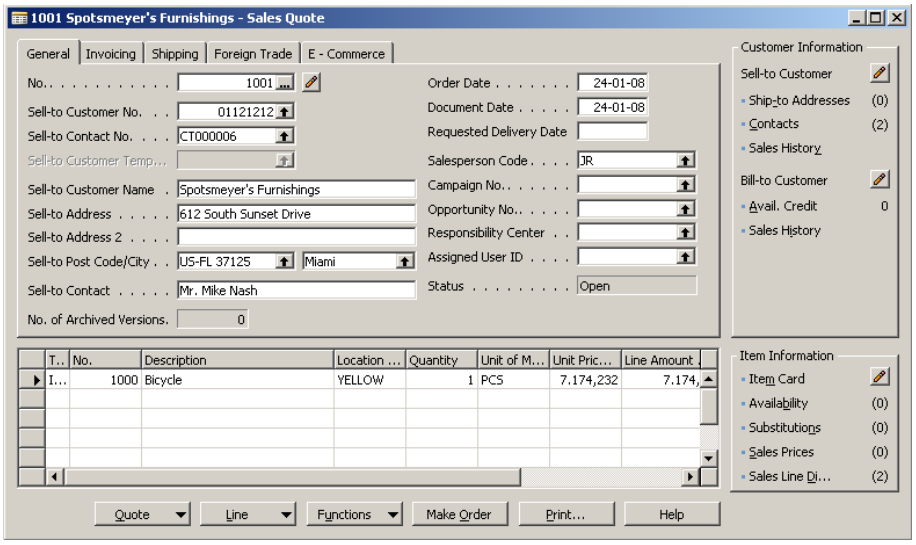


Figure 2.2: A sales quote in Microsoft Dynamics NAV.

We take as starting point the *sales module* of NAV; a sale in NAV starts with a quote being set up (Figure 2.2). Setting up a quote means that a row is created in a table (called "Sales Header"), which holds more or less *all* data present in the form of Figure 2.2 (with the exception of the list of items in the quote). In relation to contracting, some of the interesting information recorded in the row includes:

[30] `http://www.microsoft.com/dynamics/`

- Payment terms (enumeration), for instance cash on delivery or payment 14 days after delivery;
- Payment discount (percentage);
- Shipment method (Incoterm);
- Prepayment (percentage) and prepayment due date; and
- Delivery date.

Thus the row, r, representing the quote from Figure 2.2 may for instance satisfy

$$\pi_{\mathsf{Payment\ terms}}(r) = \texttt{COD}$$

which means that the quote is based on the payment term "cash on delivery". The option chosen for payment will necessarily have influence on the sales workflow, but the problem is that this dependency is only reflected in the C/AL code, and only by inspecting this code will the "workflow content" of the different payment terms become apparent. Besides from the problem that the workflow is not represented explicitly, it also means that if a new "payment term type" is to be introduced in the system, then the C/AL code has to be extended to accommodate such a new type (and the programmer must know *where* among the million lines of code the extension should be inserted)[31].

The example above illustrates that the contractual terms are encoded by means of an enumeration type (which represents a set of predefined term types), and that the choice of payment type will affect the control flow in the C/AL code. But the "Sales Header" table also contains columns that more directly represent the workflow of a sale, for instance:

- Document type (enumeration), e.g., quote, order or invoice; and
- Shipment status (boolean).

For instance if

$$\pi_{\mathsf{Document\ Type}}(r) = \texttt{Quote}$$

then the sale is at the quote stage, meaning that the order has not yet been confirmed (once the order is confirmed the document type changes; $\pi_{\mathsf{Document\ Type}}(r) = \texttt{Order}$). Again we see that this approach is not very flexible: if the sales process for some reason changes (for instance to include approval from a sales manager), then the document type has to be extended

[31] For instance the version of Microsoft Dynamics NAV we have obtained does not include a payment term suitable for expressing the payment option of the sample contract in Appendix D, hence such an option would have to be implemented by a C/AL programmer.

with a new enumeration type, and the C/AL code has to make sure that a quote cannot be converted to an order before it has been approved. More abstractly one may therefore think of the C/AL code as representing the *transitions* in a workflow, and the database as representing the *states* in the workflow; the problem then is that the connection between the two can only be seen by inspecting the C/AL code, which is not very feasible for business experts.

Hence the conclusion is that the ERP system Microsoft Dynamics NAV does indeed support workflows, but not to the extent that we are seeking. Instead of representing contracts/workflows explicitly as first-class values in the system, they are divided into state (the database) and control flow (the C/AL code), with no evident connection between the two. Workflow "types" are introduced in an ad hoc manner into the system, by extending the table(s) representing the state, and by making changes to the C/AL code. This step hence requires an (expert) C/AL programmer, rather than a domain expert in contracting/business workflows.

The introduction to Microsoft Dynamics NAV here has necessarily been very brief; for a more detailed description we refer to our report [33], which is targeted at computer scientists.

2.4 Summary & Future Work

We have presented in this chapter a comprehensive survey on tools and technologies for electronic contract lifecycle management. The survey is in itself interesting, since we have covered the current status of research, and presented the technical aspects of the different approaches; to the best of our knowledge, no such survey existed prior to our work. Our emphasis has been on the theoretical foundations for electronic contracting, but we have also included a selection of commercial CLM products, and based on these a set of features relevant for CLM systems.

As mentioned in the preface, the longer term purpose of this survey is to provide a solid foundation for constructing a domain-specific language (and semantic model) for commercial contracts. Based on the approaches we have covered, we find that there is still work to be done with respect to contract formalization, hence we still find it relevant to construct our own model and DSL for commercial contracts.

One of the considerations that has reoccurred mostly in each section, is whether contracts are encoded in a language with formal semantics (requirement R1). So why is this property so important? First of all – as mentioned in the introduction of this chapter – providing contracts with a formal semantics means that contracting parties can agree on *exactly* what the contract means. We saw for instance in the example contract from Appendix D that there was no explicit mention of buyer's obligation to pay for the goods (Paragraph 5) being contingent upon seller delivering the goods (Paragraph 3). Even though it may seem obvious that this is the intention, it is dangerous to assume that all contract parties have the same intentions.

So the next question is: what characterizes a "good" formal model for commercial contracts? First of all, we find it very important that contract execution be deterministic: when an event/action with relevance to the contract occurs, then the evolution/execution of the contract should result in a (new) unique contract state. This state may for instance indicate a breach of contract, or represent a new set of obligations. In most of the work we have considered, contract execution seems to be deterministic (with the exception of CCML in Section 2.2.8). The benefit of having deterministic contract execution is that the contracting parties only have to agree on *what has happened*: if such an agreement can be reached, then the state of affairs will be uniquely determined (and one party can even provide a formal *proof* that the other party has breached the contract, in case of a lawsuit). The problem of agreeing upon what has happened in the real world, seems to be present no matter which contract model is used; we therefore have to assume that such an agreement can be reached (for instance via a handshaking protocol as suggested in the e-contracts framework of Section 2.2.4,

or via the use of a reliable monitoring "bank", as suggested by Tan et al. in Section 2.2.1).

As witnessed by our survey, contract modeling has drawn attention from various scientific fields; logics, information science, and computer science (programming languages). Accordingly, it has been proposed to use logical models, active database-inspired models, and process calculi-inspired models for expressing contracts:

The logical models all seem to be inspired by deontic logic, and the pioneering work by Lee (Section 2.2.3). Deontic logic undoubtedly seems as a perfect candidate for expressing contracts (obligations, permissions, and prohibitions), but the original intention of deontic logic seems to be different from what is needed in contracts: deontic modalities are not bound to agents, who are expected to behave in a particular way; rather, the deontic modalities seem to be concerned with more abstract states of affairs (cf. the *ought-to-be* versus *ought-to-do* discussion in Section 2.2.10). This problem has also been identified in some of the approaches we have considered, and the solutions have been to annotate deontic modalities with agents, and to restrict deontic modalities to actions. But the link back to the semantics of deontic logic is missing – perhaps because the *alternative world* semantics of SDL is not suited for contracts. Another problem with the logical formulations is that we have found no account for what it means to *execute* a logically modeled contract.

In contrast to this, contract execution seems to be the main focus of the event-condition-action (ECA) based approaches that we have considered. In these models, a contract is modeled as a set of rules together with a contract *state*. The state of the contract determines which rules are active, and contract execution is a matter of updating the contract state based on the events that occur. However, one of the problems of this approach is (in our opinion) that it can be difficult to understand (and analyze) the control flow of a state-based contract. (Though the connection between contract state and contract rules/transitions is much more evident than in the formalization seen in Section 2.3.2.)

In the process-calculi inspired approach we have seen (CCML), the problem above is not present: simply because the contract term *is the state* (one of the characteristics of process calculi). Here the contract represents all current obligations, and any obligations that are contingent upon other obligations being met, are explicitly sequenced accordingly. But the problem with this particular solution is that it does not satisfy the requirement of deterministic contract execution, mentioned above.

So what we propose instead of the approaches above, is a game-theoretic model for expressing contracts (inspired by the work of Chapter 1). In our

opinion, a game-semantic model provides a more faithful interpretation of contracts, than the proposals above: (a) a game consists of a set of players (= contract parties), (b) each player seeks to "win" (= maximize profit, avoid penalties, etc.), (c) the game describes the rules and "who wins what from whom" (= the normative content of a contract), and (d) the game state changes based on the players' moves (= the normative content changes when parties act).

To make this idea more concrete, consider the contract from Appendix D. This contract may be interpreted as a game with two players: the seller and the buyer. Initially the game is in a state where seller is expected to deliver goods to buyer; if seller makes the delivery on time, the game will transition to a new state, where buyer is expected to pay the first half. If seller fails to deliver the goods, the game will transition to a state where seller is expected to pay a penalty (but note that seller has not *lost* the game). If seller pays the penalty, the game ends; but if seller fails to pay the penalty, the game is terminated with an indication that seller has lost (and perhaps an indication of how much seller owes buyer).

This model may seem very close to the event-condition-action based proposal, and indeed there are some similarities. But unlike the ECA models (which include obligations, permissions and prohibitions as primitives), we are interested in an abstract (denotational) model, which merely describes how the game transitions, based on the moves of the players. Hence in this model there is *a priori* no obligations, permission or prohibitions, but the transition/rule function may describe a situation which can be *interpreted* as an obligation. But does this mean that we want to code business contracts as an abstract mathematical object (game)? No! The intention is that the game-theoretic model will be used to describe the denotation of a contract, but we still need a domain-specific language for describing contracts (we return to the DSL considerations later).

The first topic of future research is therefore to construct a game-theoretic (denotational) model for business contracts; the model introduced in Chapter 1 is a step in the right direction, but it is not tailored to the commercial domain (the contracts of Chapter 1 are more suited for computational services based on (micro)payments). In particular the contracts of Chapter 1 signifies monetary transactions (via payoffs) as opposed to general resources, which is not desirable. Based on what we have learned in our survey, we gather a list of aspects which should be taken into consideration when defining the game-theoretic model:

- Should the semantic model distinguish between primary and secondary (contrary-to-duty) obligations? From a pure game-theoretic point of view, primary/secondary obligations are not different, they merely describe two different situations in the game. As we have discussed

extensively in Chapter 1, it may sometimes be more profitable for an agent to deliberately violate a (primary) obligation in a contract, which suggests that primary/secondary obligations are just *alternative* obligations. (There may of course be other factors such as *reputation* at stake when considering which obligations to fulfill, which may be a reason for distinguishing between primary and secondary obligations. For this purpose the notion of (sub)ideal situations introduced in BCL in Section 2.2.6 may be useful.)

- Which kinds of events are needed for commercial contracts? In many of the approaches we have covered, events are encoded as propositional constants, but as we have argued earlier, a better solution is to have some fixed set of message types, in which parameters such as resources are included. CCML suggests to use the REA ontology for describing events, which may be a feasible solution.

- Are bilateral contracts sufficient? The appeal of bilateral contracts has been elaborated on in Chapter 1 (in particular assuming responsibility for success, which makes very good sense in a pure business context as well). Bilateral contracts do not necessarily mean that there are only two agents in the contract, it means that the *responsibility* for the normative contents are divided between the two signatories (principals). Example 1.4.7 from Chapter 1 illustrates this point: a client and a service provider have signed a contract, where a third agent (subcontractor) is used by the service provider to fulfill its obligations. But it is the responsibility of the provider that the subcontractor does what he is supposed to (for which reason the service provider has signed a separate contract with the subcontractor).

- Are infinite contracts needed? In CCML we saw possibility of expressing infinite contracts, but reasoning about potentially infinite contracts is much harder (if possible). In programming it is often the *intention* that a program should run forever (though in practice it never does), but are business contracts ever intended to run forever?

As mentioned above, the game-theoretic model will be used to capture the meaning (denotation) of contracts, but it is not a model in which contracts should be written directly. For this purpose we wish construct a DSL, and here many of the ideas we have encountered can be used:

- The isomorphism principle should be fulfilled to the extent that it is possible (we have already explained why in the introduction), which suggests a paragraph-based encoding of contracts (as seen in many approaches).

- A stateless encoding of contracts is desirable (as in CCML): avoiding contract state means that it is easier to see the control flow directly, and contract analyses can be performed directly on the syntax of the contract. Furthermore stateless contracts are immediately *portable* due to their self-containment.

- If a stateless encoding is possible, a reduction semantics should be provided. If c is a contract in the DSL and $[\![c]\!]$ is the denotation of c as a game, then the reduction semantics should be sound and complete (as in CCML): c steps to contract c' under event e exactly when the game $[\![c']\!]$ is equivalent to the game resulting from the transition of game $[\![c]\!]$ under event e. The reason why this property is interesting is that the residual obligations of contract c after event e has occurred, can be found by reducing c according to the reduction semantics. This enables efficient on-line monitoring of contracts and permits "what if" analysis, as discussed in Section 2.2.8.

- Which deontic modalities are needed in the DSL? It is quite obvious that the need for expressing obligations is present (whether they be expressed directly or as dualized permissions). We also claim that obligations should always be time-bounded; by this we mean that an obligation to perform some action should really be an obligation to perform that action before a certain *deadline*. Obligations without deadlines are useless, as the opponent can never prove that it has been violated (i.e., violations should happen in finite time, as discussed also in Chapter 1). This approach was also taken in the e-contracts framework.

  Are permissions needed? If a permission to perform some action does not imply an obligation for some other contract participant, is it then interesting? This suggests that only conditional obligations are needed (for instance "if buyer has informed seller of a claim for damage, then seller is obliged ..."). And if prohibitions are included and an action is not explicitly prohibited, then it should be permitted (closed world assumption; i.e., if buyer is not prohibited from claiming for damage, then he can do so).

- Are business contracts always concerned with ought-to-do statements, and never ought-to-be statements? At the moment we believe ought-to-do statements are sufficient for business contracts, since it is not even clear who is to blame if an ought-to-be obligation is not met. This is also the approach taken by almost all the formalisms we have considered.

The bullet points on the last couple of pages present many open questions, which are the subject of future research. A common requirement for be-

ing able to answer many of the points above, is a "representative" set of of example business contracts. In the even longer run, the goal is to utilize the formalisms above to be able to perform automatic analysis of contracts, and in particular to verify compliance of business processes to business contracts. Chapter 1 contains some of these aspects, which we seek to adopt to the pure business setting. Also in the even longer run, topics such as contract negotiation should be investigated: one possible interpretation of the negotiation phase is that contract negotiation itself occurs according to a contract (due to Lee, Section 2.2.3), and hence contract negotiation can (to some extent) be seen as a higher-order contract (whatever that exactly means).

With these open problems, ideas, and thoughts, we conclude the chapter.

Appendix A

Compset

This part of the appendix contains a short section on *compsets* (not a standard definition), which are used to prove equivalence of processes and automata in Section 1.3.1.

Definition A.0.1 (Compset)**.** *Let G be a set with partial binary operator $\cdot : G \times G \rightharpoonup G$ (we will write g_1g_2 for $g_1 \cdot g_2$), satisfying:*

1. *If g_1g_2 is defined then g_2g_1 is defined.*
2. *If g_1g_2 and g_1g_3 and g_2g_3 are all defined then $(g_1g_2)g_3$ is defined.*

We then call $(G, \cdot)$ a compset.

Lemma A.0.2. *If g_1g_2 and g_1g_3 and g_2g_3 are all defined then $g_1(g_2g_3)$ is defined.*

Proof. From 1. we get that g_2g_1 and g_3g_1 are defined and therefore by 2. we get that $(g_2g_3)g_1$ is defined, and the result follows from 1. □

Definition A.0.3 (Homomorphism/isomorphism)**.** *Let $(G, \cdot)$ and $(H, \cdot)$ be compsets. Then $\phi : G \to H$ is called a homomorphism whenever*

1. *g_1g_2 is defined if and only if $\phi(g_1)\phi(g_2)$ is defined.*
2. *$\phi(g_1g_2) = \phi(g_1)\phi(g_2)$.*

ϕ is called an isomorphism whenever ϕ is a bijective homomorphism.

Lemma A.0.4. *Let $\phi : G \to H$ be an isomorphism between compsets G and H. Then*

(a) $\phi^{-1} : H \to G$ *is an isomorphism.*

(b) If $\psi : H \to G$ *satisfies* $\phi(\psi(h)) = h$ *for all* $h \in H$ *then* $\psi = \phi^{-1}$ *(and hence* ψ *is an isomorphism).*

(c) If the composition in G *is associative then so is the composition in* H*. By associative we mean: if* g_1g_2 *and* g_1g_3 *and* g_2g_3 *are defined then* $(g_1g_2)g_3 = g_1(g_2g_3)$*.*

(d) If the composition in G *is commutative then so is the composition in* H*. By commutative we mean: if* g_1g_2 *is defined then* $g_1g_2 = g_2g_1$*.*

Proof.

(a) ϕ^{-1} is by definition bijective, so we need to show that it is an homomorphism. First: $h_1h_2 = \phi(g_1)\phi(g_2)$ is defined exactly when $g_1g_2 = \phi^{-1}(\phi(g_1))\phi^{-1}(\phi(g_2))$ is defined. Second:

$$\begin{aligned}
\phi^{-1}(h_1h_2) &= \phi^{-1}(\phi(g_1)\phi(g_2)) && (\phi \text{ surjective})\\
&= \phi^{-1}(\phi(g_1g_2)) && (\phi \text{ homomorphic})\\
&= g_1g_2\\
&= \phi^{-1}(h_1)\phi^{-1}(h_2) && (\phi \text{ injective})
\end{aligned}$$

(b) $\psi(h) = \phi^{-1}(\phi(\psi(h))) = \phi^{-1}(h)$

(c) If h_1h_2 and h_1h_3 and h_2h_3 are defined then with $\phi(g_i) = h_i$ we have that g_1g_2 and g_1g_3 and g_2g_3 are defined because ϕ is a homomorphism. Now:

$$\begin{aligned}
h_1(h_2h_3) &= \phi(g_1)\big(\phi(g_2)\phi(g_3)\big)\\
&= \phi(g_1)\phi(g_2g_3) && (\phi \text{ homomorphic})\\
&= \phi(g_1(g_2g_3)) && (\phi \text{ homomorphic})\\
&= \phi((g_1g_2)g_3) && (\text{associativity in } G)\\
&= \phi(g_1g_2)\phi(g_3) && (\phi \text{ homomorphic})\\
&= \big(\phi(g_1)\phi(g_2)\big)\phi(g_3) && (\phi \text{ homomorphic})\\
&= (h_1h_2)h_3
\end{aligned}$$

(d) If h_1h_2 is defined then with $\phi(g_i) = h_i$ we have that g_1g_2 is defined because ϕ is a homomorphism. Now:

$$\begin{aligned}
h_1h_2 &= \phi(g_1)\phi(g_2)\\
&= \phi(g_1g_2) && (\phi \text{ homomorphic})\\
&= \phi(g_2g_1) && (\text{commutativity in } G)\\
&= \phi(g_2)\phi(g_1) && (\phi \text{ homomorphic})\\
&= h_2h_1
\end{aligned}$$

□

Lemma A.0.5. *Let $(G, \cdot)$ be a compset and let $R \subseteq G \times G$ be a congruence relation, i.e.,*

$$\forall g \in G.(g, g) \in R \qquad \text{(reflexive)}$$
$$\forall g_1, g_2 \in G.(g_1, g_2) \in R \Rightarrow (g_2, g_1) \in R \qquad \text{(symmetric)}$$
$$\forall g_1, g_2, g_3 \in G.(g_1, g_2) \in R \wedge (g_2, g_3) \in R \Rightarrow (g_1, g_3) \in R \qquad \text{(transitive)}$$
$$\forall g_1, g_2, g_3, g_4 \in G.(g_1, g_2), (g_3, g_4) \in R \Rightarrow (g_1 g_3, g_2 g_4) \in R$$

where $(g_1 g_3, g_2 g_4) \in R$ means either both $g_1 g_3$ and $g_2 g_4$ undefined or both defined and related. Then $(G/R, \cdot)$ is a compset with

$$[g_1][g_2] \stackrel{\text{def}}{=} \begin{cases} \text{undefined} & \text{if } g_1 g_2 \text{ is undefined} \\ [g_1 g_2] & \text{otherwise} \end{cases}$$

where $[g] = \{g' \in G \mid (g, g') \in R\}$.

Proof. We must show that composition is well-defined, i.e., if $[g_1] = [g_2]$ and $[g_3] = [g_4]$ then $[g_1][g_3] = [g_2][g_4]$. But this follows by definition from R being a congruence relation. Next we must show that 1. and 2. are fulfilled but this follows from them being fulfilled for G. □

Lemma A.0.6. *Let $(G, \cdot)$ and $(H, \cdot)$ be compsets and R a congruence relation on G. If $\phi : G \to H$ is a surjective homomorphism satisfying $\phi(g_1) = \phi(g_2) \Leftrightarrow (g_1, g_2) \in R$, then the induced function $\phi_R : G/R \to H$ defined by*

$$\phi_R([g]) = \phi(g)$$

is an isomorphism.

Proof. We see first that ϕ_R is well-defined, as $(g_1, g_2) \in R \Rightarrow \phi(g_1) = \phi(g_2)$. Since ϕ is surjective, so is ϕ_R, and ϕ_R is injective since $\phi(g_1) = \phi(g_2) \Rightarrow (g_1, g_2) \in R$. Finally ϕ_R is homomorphic since $[g_1][g_2]$ is defined exactly when $g_1 g_2$ is defined which is exactly when $\phi(g_1)\phi(g_2) = \phi_R([g_1])\phi_R([g_2])$ is defined. And then we have that:

$$\phi_R([g_1][g_2]) = \phi_R([g_1 g_2]) = \phi(g_1 g_2) = \phi(g_1)\phi(g_2) = \phi_R([g_1])\phi_R([g_2])$$

□

Appendix B

Proofs

This part of the appendix contains the larger proofs of the report.

Lemma 1.2.14. *Let* $\mathsf{p}_1 = (C_I^1, C_O^1, f^1)$ *and* $\mathsf{p}_2 = (C_I^2.C_O^2, f^2)$ *be given processes, and assume that* $\mathsf{p}_1 \parallel \mathsf{p}_2 = (C_I, C_O, f)$ *exists (Definition 1.2.11). Then* $\mathsf{p}_1 \parallel \mathsf{p}_2$ *is a process.*

Proof. We first need to show that $C_I \cap C_O = \emptyset$:

$$\begin{aligned}
C_I \cap C_O &= ((C_I^1 \cup C_I^2) \setminus C_{\text{int}}) \cap ((C_O^1 \cup C_O^2) \setminus C_{\text{int}}) \\
&= ((C_I^1 \cup C_I^2) \cap (C_O^1 \cup C_O^2)) \setminus C_{\text{int}} \\
&= ((C_I^1 \cap (C_O^1 \cup C_O^2)) \cup (C_I^2 \cap (C_O^1 \cup C_O^2))) \setminus C_{\text{int}} \\
&= ((C_I^1 \cap C_O^2) \cup (C_I^2 \cap C_O^1)) \setminus C_{\text{int}} \qquad (C_I^j \cap C_O^j = \emptyset, j = 1, 2) \\
&= C_{\text{int}} \setminus C_{\text{int}} \\
&= \emptyset
\end{aligned}$$

Next we must show that f is strictly monotone. Let $l_1, l_2 \in \mathcal{L}_{C_I}$ be given and assume that $l_{1|t} = l_{2|t}$, for $t < \min(\text{eol}(l_1), \text{eol}(l_2))$. Then $f(l_1) = (\mathcal{I}_N^1 \bowtie \mathcal{I}_N^2)_{|C_O}$ and $f(l_2) = (\mathcal{J}_M^1 \bowtie \mathcal{J}_M^2)_{|C_O}$ for some N and M. We show by induction on n that

$$\mathcal{I}_{n|t+1}^1 = \mathcal{J}_{n\,|t+1}^1 \wedge \mathcal{I}_{n|t+1}^2 = \mathcal{J}_{n\,|t+1}^2$$

$\underline{n = 0}$: OK, as $\mathcal{I}_0^1 = \mathcal{I}_0^2 = \mathcal{J}_0^1 = \mathcal{J}_0^1 = \emptyset$.

$\underline{n > 0}$: By the induction hypothesis it follows that $\mathcal{I}_{n-1|t+1}^2 = \mathcal{J}_{n-1|t+1}^2$ and therefore $((\mathcal{I}_{n-1}^2 \bowtie l_1)_{|C_I^1})_{|t} = ((\mathcal{J}_{n-1}^2 \bowtie l_2)_{|C_I^1})_{|t}$. But then strict monotonicity of f^1 yields:

$$\mathcal{I}_{n|t+1}^1 \stackrel{\text{def}}{=} f^1((\mathcal{I}_{n-1}^2 \bowtie l_1)_{|C_I^1})_{|t+1} = f^1((\mathcal{J}_{n-1}^2 \bowtie l_2)_{|C_I^1})_{|t+1} \stackrel{\text{def}}{=} \mathcal{J}_{n\,|t+1}^2$$

By a similar argument it follows that $\mathcal{I}^2_{n|t+1} = \mathcal{J}^2_{n\,|t+1}$ as required.

So for $n = \max(N, M)$ it follows that $(\mathcal{I}^1_n \bowtie \mathcal{I}^2_n)_{|t+1} = (\mathcal{J}^1_n \bowtie \mathcal{J}^2_n)_{|t+1}$ which implies that $f(l_1)_{|t+1} = f(l_2)_{|t+1}$ as required. □

Theorem 1.2.15. *Let* $\mathsf{p}_1 = (C^1_I, C^1_O, f^1)$ *and* $\mathsf{p}_2 = (C^2_I.C^2_O, f^2)$ *be given processes, such that* $C^1_I \cap C^2_I = C^1_O \cap C^2_O = \emptyset$. *Then parallel composition* $\mathsf{p}_1 \parallel \mathsf{p}_2$ *is always defined.*

Proof. Let $l \in \mathcal{L}_{C_I}$ be given. We need to show that there exists some $N \in \mathbb{N}$ such that $\mathcal{I}^1_N = \mathcal{I}^1_{N+1}$ and $\mathcal{I}^2_N = \mathcal{I}^2_{N+1}$ (c.f. Definition 1.2.11). We show by induction on n that

$$\mathcal{I}^i_{n|n} = \mathcal{I}^i_{n+1|n}$$

for $i = 1, 2$.

Case $n = 0$: $\mathcal{I}^i_{0|0} = \mathcal{I}^i_{1|0}$ holds trivially for $i = 1, 2$.

Case $n > 0$: It follows from the induction hypothesis that $\mathcal{I}^2_{n-1|n-1} = \mathcal{I}^2_{n|n-1}$ and thus $((\mathcal{I}^2_{n-1} \bowtie l)_{|C^1_I})_{|n-1} = ((\mathcal{I}^2_n \bowtie l)_{|C^1_I})_{|n-1}$. But then strict monotonicity of f^1 yields

$$\mathcal{I}^1_{n|n} \stackrel{\text{def}}{=} f^1((\mathcal{I}^2_{n-1} \bowtie l)_{|C^1_I})_{|n} = f^1((\mathcal{I}^2_n \bowtie l)_{|C^1_I})_{|n} \stackrel{\text{def}}{=} \mathcal{I}^1_{n+1|n}$$

By a similar argument it follows that $\mathcal{I}^2_{n|n} = \mathcal{I}^2_{n+1|n}$ as required.

The result now follows by choosing $N = \text{eol}(l)$. □

Lemma 1.3.5. *Below follows a series of results about OERs.*

(1) The identity relation $R_{\text{id}} \subseteq S \times S$ *is an OER for* a *and* a*:*

$$R_{\text{id}} = \{(s, s) \mid s \in S\}$$

(2) Let $R \subseteq S_1 \times S_2$ *be an OER for* a_1 *and* a_2. *Then the inverse relation* $R^{-1} \subseteq S_2 \times S_1$ *is an OER for* a_2 *and* a_1*:*

$$R^{-1} = \{(s_2, s_1) \mid (s_1, s_2) \in R\}$$

(3) Let $R_1 \subseteq S_1 \times S_2$ *and* $R_2 \subseteq S_2 \times S_3$ *be OERs for* a_1,a_2 *and* a_2,a_3. *Then the composed relation* $R_1 \circ R_2 \subseteq S_1 \times S_3$ *is an OER for* a_1 *and* a_3*:*

$$R_1 \circ R_2 = \{(s_1, s_3) \mid \exists s_2.\ (s_1, s_2) \in R_1 \wedge (s_2, s_3) \in R_2\}$$

Proof. We show that each of the relations are observational equivalence relations.

(1) Assume that $s_1 \; R_{\text{id}} \; s_2$. Then $s_1 = s_2$ so the conditions are trivially fulfilled.

(2) Assume that $s_2 \; R^{-1} \; s_1$. Then $s_1 \; R \; s_2$ which means that:

$$\begin{gathered} \delta^1_{\text{o}}(s_1) = \delta^2_{\text{o}}(s_2) \\ \forall m \in \mathcal{M}_{C_I}.\delta^1_{\text{t}}(s_1, m) \; R \; \delta^2_{\text{t}}(s_2, m) \end{gathered}$$

which gives:

$$\begin{gathered} \delta^2_{\text{o}}(s_2) = \delta^1_{\text{o}}(s_1) \\ \forall m \in \mathcal{M}_{C_I}.\delta^2_{\text{t}}(s_2, m) \; R^{-1} \; \delta^1_{\text{t}}(s_1, m) \end{gathered}$$

(3) Assume that $s_1(R_1 \circ R_2)s_3$. Then there exist s_2 such that $s_1 \; R_1 \; s_2$ and $s_2 \; R_2 \; s_3$ and then we get:

$$\delta^1_{\text{o}}(s_1) = \delta^2_{\text{o}}(s_2) = \delta^3_{\text{o}}(s_3)$$

and because $\delta^1_{\text{t}}(s_1, m) \; R_1 \; \delta^2_{\text{t}}(s_2, m)$ and $\delta^2_{\text{t}}(s_2, m) \; R_2 \; \delta^3_{\text{t}}(s_3, m)$ then by definition $\delta^1_{\text{t}}(s_1, m) \; (R_1 \circ R_2) \; \delta^3_{\text{t}}(s_3, m)$.

□

Lemma 1.3.6. *Observational equivalence is a congruence relation on automata, i.e., an equivalence relation (reflexive, transitive and symmetric) and*

$$\forall \mathsf{a}_1, \mathsf{a}_2, \mathsf{a}_3, \mathsf{a}_4 \in \mathfrak{A}.\mathsf{a}_1 \equiv \mathsf{a}_2 \wedge \mathsf{a}_3 \equiv \mathsf{a}_4 \Rightarrow \mathsf{a}_1 \parallel \mathsf{a}_3 \equiv \mathsf{a}_2 \parallel \mathsf{a}_4$$

Proof. The properties are proved one at a time below.

- (Reflexivity): We must show that for all a we have that $\mathsf{a} \equiv \mathsf{a}$. This corresponds to constructing an observational equivalence relation $R \subseteq S \times S$ such that $s_0 \; R \; s_0$. Lemma 1.3.5 (1) gives such a relation.
- (Symmetry): We must show that if $\mathsf{a}_1 \equiv \mathsf{a}_2$ then $\mathsf{a}_2 \equiv \mathsf{a}_1$. Similar to the reflexivity case we must take some arbitrary observational equivalence relation $R \subseteq S_1 \times S_2$ and then construct an observational equivalence relation on $S_2 \times S_1$. Lemma 1.3.5 (2) gives this relation.
- (Transitivity): We must show that if $\mathsf{a}_1 \equiv \mathsf{a}_2$ and $\mathsf{a}_2 \equiv \mathsf{a}_3$ then $\mathsf{a}_1 \equiv \mathsf{a}_3$. In this case we are given observational equivalence relations $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_2 \times S_3$ and we construct an observational equivalence relation using Lemma 1.3.5 (3).

- (Congruence): Now let

$$\begin{aligned}
\mathsf{a}_1 &= (C_I, C_O, S_1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)\\
\mathsf{a}_2 &= (C_I, C_O, S_2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)\\
\mathsf{a}_3 &= (C'_I, C'_O, S_3, s_0^3, \delta_{\mathrm{o}}^3, \delta_{\mathrm{t}}^3)\\
\mathsf{a}_4 &= (C'_I, C'_O, S_4, s_0^4, \delta_{\mathrm{o}}^4, \delta_{\mathrm{t}}^4)\\
\mathsf{a}_1 \parallel \mathsf{a}_3 &= (C''_I, C''_O, S_1 \times S_3, \langle s_0^1, s_0^3\rangle, \delta_{\mathrm{o}}^{1\|3}, \delta_{\mathrm{t}}^{1\|3})\\
\mathsf{a}_2 \parallel \mathsf{a}_4 &= (C''_I, C''_O, S_2 \times S_4, \langle s_0^2, s_0^4\rangle, \delta_{\mathrm{o}}^{2\|4}, \delta_{\mathrm{t}}^{2\|4})
\end{aligned}$$

We show that given observational equivalence relations $R_1 \subseteq S_1 \times S_2$ and $R_2 \subseteq S_3 \times S_4$ then the relation

$$\begin{aligned}
&R \subseteq (S_1 \times S_3) \times (S_2 \times S_4)\\
&R = \{(\langle s_1, s_3\rangle, \langle s_2, s_4\rangle) \mid (s_1, s_2) \in R_1 \wedge (s_3, s_4) \in R_2\}
\end{aligned}$$

is an observational equivalence relation (and clearly it relates the starting states of $\mathsf{a}_1 \parallel \mathsf{a}_3$ and $\mathsf{a}_2 \parallel \mathsf{a}_4$).

So assume that $\langle s_1, s_3\rangle\ R\ \langle s_2, s_4\rangle$ this means that $s_1\ R_1\ s_2$ and $s_3\ R_2\ s_4$. We now have:

$$\begin{aligned}
\delta_{\mathrm{o}}^{1\|3}(\langle s_1, s_3\rangle) &= (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^3(s_3))_{|C''_O} && \text{(by definition)}\\
&= (\delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^4(s_4))_{|C''_O} && (R_1,\ R_2 \text{ OERs})\\
&= \delta_{\mathrm{o}}^{2\|4}(\langle s_2, s_4\rangle) && \text{(by definition)}
\end{aligned}$$

And if $m \in \mathcal{M}_{C''_I}$ then because R_1 and R_2 are observational equivalence relations we have that:

$$\begin{aligned}
&\delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I})\ R_1\ \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I})\\
&\delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C'_I})\ R_2\ \delta_{\mathrm{t}}^4(s_4, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C'_I})
\end{aligned}$$

But then we have:

$$\begin{aligned}
\delta_{\mathrm{t}}^{1\|3}(\langle s_1, s_3\rangle, m) &= \langle \delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I}), \delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C'_I})\rangle\\
&\quad R\ \langle \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I}), \delta_{\mathrm{t}}^4(s_4, (m \cup \delta_{\mathrm{o}}^1(s_1))_{|C'_I})\rangle\\
&= \langle \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^4(s_4))_{|C_I}), \delta_{\mathrm{t}}^4(s_4, (m \cup \delta_{\mathrm{o}}^2(s_2))_{|C'_I})\rangle\\
&= \delta_{\mathrm{t}}^{2\|4}(\langle s_2, s_4\rangle, m)
\end{aligned}$$

where the second last equality follows from R_1 and R_2 being observational equivalence relations. Hence the result follows as required.

□

Lemma B.0.7. *For sets with $A \cap B = \emptyset$ the following equality holds:*

$$(A \cup (B \setminus C)) \setminus (D \cup (C \setminus B)) = (A \cup B) \setminus (C \cup D)$$

Proof.

$$\begin{aligned}
&(A \cup (B \setminus C)) \setminus (D \cup (C \setminus B)) \\
&= \big(A \setminus (D \cup (C \setminus B))\big) \cup \big((B \setminus C) \setminus (D \cup (C \setminus B))\big) \\
&\overset{*}{=} \big(A \setminus (D \cup C)\big) \cup \big((B \setminus C) \setminus D) \cap ((B \setminus C) \setminus (C \setminus B))\big) \\
&= \big(A \setminus (C \cup D)\big) \cup \big((B \setminus (C \cup D)) \cap (B \setminus C)\big) \\
&= \big(A \setminus (C \cup D)\big) \cup \big(B \setminus (C \cup D)\big) \\
&= (A \cup B) \setminus (C \cup D)
\end{aligned}$$

where the equation marked with star follows from $A \cap B = \emptyset$. □

Theorem 1.3.7 (Parallel composition is associative)**.** *Consider three automata, $\mathsf{a}_i = (C_I^i, C_O^i, S^i, s_0^i, \delta_{\mathrm{o}}^i, \delta_{\mathrm{t}}^i) \in \mathfrak{A}$, for $i = 1, 2, 3$, where*

$$C_I^1 \cap C_I^2 = C_I^1 \cap C_I^3 = C_I^2 \cap C_I^3 = \emptyset$$
$$C_O^1 \cap C_O^2 = C_O^1 \cap C_O^3 = C_O^2 \cap C_O^3 = \emptyset$$

then

$$\mathsf{a}_1 \parallel (\mathsf{a}_2 \parallel \mathsf{a}_3) \equiv (\mathsf{a}_1 \parallel \mathsf{a}_2) \parallel \mathsf{a}_3$$

Proof. First part of the proof is to show that the input and output channels of the composed automata are well-defined, we show that

$$C_I^{1\|(2\|3)} = (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3) = C_I^{(1\|2)\|3}$$
$$C_O^{1\|(2\|3)} = (C_O^1 \cup C_O^2 \cup C_O^3) \setminus (C_I^1 \cup C_I^2 \cup C_I^3) = C_O^{(1\|2)\|3}$$

We only show the case for the input channels, the output case is symmetric. We start with the left equality, and first note that

$$\begin{aligned}
C_I^{1\|(2\|3)} = \ & \big(C_I^1 \cup \big((C_I^2 \cup C_I^3) \setminus (C_O^2 \cup C_O^3)\big)\big) \setminus \\
& \big(C_O^1 \cup \big((C_O^2 \cup C_O^3) \setminus (C_I^2 \cup C_I^3)\big)\big)
\end{aligned}$$

From the assumptions we get that $C_I^1 \cap (C_I^2 \cup C_I^3) = \emptyset$ this means the we can use Lemma B.0.7 with $A = C_I^1$, $B = C_I^2 \cup C_I^3$, $C = C_O^2 \cup C_O^3$ and $D = C_O^1$ and then we get:

$$C_I^{1\|(2\|3)} = (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3)$$

For the right equality we note that:

$$\begin{aligned} C_I^{(1\|2)\|3} = \ & \big(\big((C_I^1 \cup C_I^2) \setminus (C_O^1 \cup C_O^2)\big) \cup C_I^3\big) \setminus \\ & \big(\big((C_O^1 \cup C_O^2) \setminus (C_I^1 \cup C_I^2)\big) \cup C_O^3\big) \end{aligned}$$

But then we have from assumptions that $C_I^3 \cap (C_I^1 \cup C_I^2) = \emptyset$ and we can use Lemma B.0.7 with $A = C_I^3$, $B = C_I^1 \cup C_I^2$, $C = C_O^1 \cup C_O^2$ and $D = C_O^3$, and get:

$$C_I^{(1\|2)\|3} = (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3)$$

We abbreviate:

$$\begin{aligned} C_I &= (C_I^1 \cup C_I^2 \cup C_I^3) \setminus (C_O^1 \cup C_O^2 \cup C_O^3) \\ C_O &= (C_O^1 \cup C_O^2 \cup C_O^3) \setminus (C_I^1 \cup C_I^2 \cup C_I^3) \end{aligned}$$

The second part of the proof is finding an observational equivalence relation which relates the start states. We use the following (which clearly relates the start states):

$$\begin{aligned} &R \subseteq \big(S_1 \times (S_2 \times S_3)\big) \times \big((S_1 \times S_2) \times S_3\big) \\ &R = \{(\langle s_1, \langle s_2, s_3\rangle\rangle, \langle\langle s_1, s_2\rangle, s_3\rangle) \mid s_1 \in S_1, s_2 \in S_2, s_3 \in S_3\} \end{aligned}$$

We prove that this is an observational equivalence relation, so assume that $\langle s_1, \langle s_2, s_3\rangle\rangle \; R \; \langle\langle s_1, s_2\rangle, s_3\rangle$. First we need to compare the outputs:

$$\begin{aligned} \delta_{\mathrm{o}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle) &= (\delta_{\mathrm{o}}^1(s_1) \cup (\delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O^{2\|3}})_{|C_O} \\ &\overset{*}{=} (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O} \\ &\overset{*}{=} ((\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_O^{1\|2}} \cup \delta_{\mathrm{o}}^3(s_3))_{|C_O} \\ &= \delta_{\mathrm{o}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle) \end{aligned}$$

where the first starred equation follows from $C_O^1 \cap (C_O^2 \cup C_O^3) = \emptyset$, and the second from $(C_O^1 \cup C_O^2) \cap C_O^3 = \emptyset$. Finally we need to show that for any $m \in \mathcal{M}_{C_I}$ the updated states are related. To show this we introduce shorthand notation:

$$\begin{aligned} s_1' &= \delta_{\mathrm{t}}^1(s_1, (m \cup \delta_{\mathrm{o}}^2(s_2) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^1}) \\ s_2' &= \delta_{\mathrm{t}}^2(s_2, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^3(s_3))_{|C_I^2}) \\ s_3' &= \delta_{\mathrm{t}}^3(s_3, (m \cup \delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^3}) \end{aligned}$$

And we calculate:

$$
\begin{aligned}
&\delta_{\mathrm{t}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle, m)\\
&= \langle \delta_{\mathrm{t}}^{1}(s_1, (m \cup (\delta_{\mathrm{o}}^{2}(s_2) \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_O^{2\|3}})_{|C_I^1}), \delta_{\mathrm{t}}^{2\|3}(\langle s_2, s_3\rangle, (m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I^{2\|3}})\rangle\\
&\overset{*}{=} \langle \delta_{\mathrm{t}}^{1}(s_1, (m \cup \delta_{\mathrm{o}}^{2}(s_2) \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I^1}), \delta_{\mathrm{t}}^{2\|3}(\langle s_2, s_3\rangle, (m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I^{2\|3}})\rangle\\
&= \langle s_1', \langle \delta_{\mathrm{t}}^{2}(s_2, ((m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I^{2\|3}} \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I^2}),\\
&\qquad\qquad \delta_{\mathrm{t}}^{3}(s_3, ((m \cup \delta_{\mathrm{o}}^{1}(s_1))_{|C_I^{2\|3}} \cup \delta_{\mathrm{o}}^{2}(s_2))_{|C_I^3})\rangle\rangle\\
&\overset{*}{=} \langle s_1', \langle \delta_{\mathrm{t}}^{2}(s_2, (m \cup \delta_{\mathrm{o}}^{1}(s_1) \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I^2}), \langle \delta_{\mathrm{t}}^{3}(s_3, (m \cup \delta_{\mathrm{o}}^{1}(s_1) \cup \delta_{\mathrm{o}}^{2}(s_2))_{|C_I^3})\rangle\rangle\\
&= \langle s_1', \langle s_2', s_3'\rangle\rangle
\end{aligned}
$$

where the first starred equation follows from $C_I^1 \cap (C_I^2 \cup C_I^3) = \emptyset$, and the second from $C_O^1 \cap (C_O^2 \cup C_O^3) = \emptyset$.

In a similar way we can calculate:

$$
\begin{aligned}
&\delta_{\mathrm{t}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle, m)\\
&= \langle \delta_{\mathrm{t}}^{1\|2}(\langle s_1, s_2\rangle, (m \cup \delta_{\mathrm{o}}^{3}(s_3))_{|C_I^{1\|2}}), \delta_{\mathrm{t}}^{3}(s_3, (m \cup (\delta_{\mathrm{o}}^{1}(s_1) \cup \delta_{\mathrm{o}}^{2}(s_2))_{|C_O^{1\|2}})_{|C_I^3})\rangle\\
&= \langle\langle s_1', s_2'\rangle, s_3'\rangle
\end{aligned}
$$

And therefore by definition we have that:

$$\delta_{\mathrm{t}}^{1\|(2\|3)}(\langle s_1, \langle s_2, s_3\rangle\rangle, m)\ R\ \delta_{\mathrm{t}}^{(1\|2)\|3}(\langle\langle s_1, s_2\rangle, s_3\rangle, m)$$

concluding the proof. □

Theorem 1.3.13 ($\ulcorner\cdot\urcorner$ is well-defined)**.** *The denotation of an automaton* $\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$ *,* $\ulcorner\mathsf{a}\urcorner = (C_I, C_O, f)$*, is a process.*

Proof. The conditions that C_I and C_O are finite and disjoint follow directly from a being an automaton. So we must show that f is a log transformer from $\mathcal{L}_{C_I}$ to $\mathcal{L}_{C_O}$.

As stated in Lemma 1.3.11, the big step relation is total and deterministic, thus f denotes a function from $\mathcal{L}_{C_I}$ to $\mathcal{L}_{C_O}$. So we need to show:

1. $\forall l \in \mathcal{L}_{C_I}.\mathrm{eol}(l) = \mathrm{eol}(f(l))$
2. $\forall l_1, l_2 \in \mathcal{L}_{C_I}.\forall t < \min(\mathrm{eol}(l_1), \mathrm{eol}(l_2)).l_{1|t} = l_{2|t} \Rightarrow f(l_1)_{|t+1} = f(l_2)_{|t+1}$

ad. 1) Follows from Lemma 1.3.11.

ad. 2) We show the following generalization:

$$\begin{array}{l}\text{If } \tau \vdash s, l_1 \Downarrow l_1' \text{ and } \tau \vdash s, l_2 \Downarrow l_2' \text{ and} \\ \quad l_{1|t} = l_{2|t} \text{ for } t < \min(\text{eol}(l_1), \text{eol}(l_2)) \\ \text{then } \forall t' \in \mathbb{N}.\tau \leq t' \leq t \Rightarrow l_1'(t') = l_2'(t')\end{array} \tag{B.1}$$

The proof is by induction on the derivation of $\tau \vdash s, l_1 \Downarrow l_1'$:

Case e-end: Now $\tau \leq t < \min(\text{eol}(l_1), \text{eol}(l_2))$ and $\tau \geq \text{eol}(l_1)$, making the case trivial.

Case e-step:

$$\frac{\delta_{\text{o}}(s) = m \qquad \delta_{\text{t}}(s, l_1(\tau)) = s' \qquad \overbrace{\tau + 1 \vdash s', l_1 \Downarrow l_1''}^{(*)}}{\tau \vdash s, l_1 \Downarrow l_1''[\tau \mapsto m]}\,(\tau < \text{eol}(l_1))$$

If $\tau \geq \text{eol}(l_2)$ then since $\tau \leq t < \min(\text{eol}(l_1), \text{eol}(l_2))$ the case is trivial (as before). So assume that $\tau < \text{eol}(l_2)$. Then the derivation of $\tau \vdash s, l_2 \Downarrow l_2'$ must have used the e-step rule as well:

$$\frac{\delta_{\text{o}}(s) = m \qquad \delta_{\text{t}}(s, l_2(\tau)) = s'' \qquad \overbrace{\tau + 1 \vdash s'', l_2 \Downarrow l_2''}^{(**)}}{\tau \vdash s, l_2 \Downarrow l_2''[\tau \mapsto m]}\,(\tau < \text{eol}(l_2))$$

Now $l_1'(\tau) = l_2'(\tau) = m$ so in order to show (B.1) it suffices to show that

$$\forall t' \in \mathbb{N}.\tau + 1 \leq t' \leq t \Rightarrow l_1'(t') = l_2'(t')$$

If $\tau = t$ the result follows trivially, so assume that $\tau < t$. Now $l_1' = l_1''[\tau \mapsto m]$ and $l_2' = l_2''[\tau \mapsto m]$ so the result will follow by the generalized induction hypothesis applied to $(*)$ and $(**)$ if we can show that $s' = s''$. But this is the case since $l_1(\tau) = l_2(\tau)$ as we assumed that $\tau < t$. This concludes the proof of (B.1).

We can now apply the generalized induction hypothesis (with $\tau = 0$) to get

$$\begin{array}{l}\forall l_1, l_2 \in \mathcal{L}_{C_I}.\forall t < \min(\text{eol}(l_1), \text{eol}(l_2)). \\ \qquad\qquad l_{1|t} = l_{2|t} \Rightarrow \forall t'.0 \leq t' \leq t.f(l_1)(t') = f(l_2)(t')\end{array}$$

which is equivalent to strict monotonicity, as required.

□

Lemma 1.3.14. *If* $\mathsf{a}_1 \equiv \mathsf{a}_2$ *then* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$.

Proof. Let $\mathsf{a}_1 = (C_I, C_O, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ be given and assume that $\mathsf{a}_1 \equiv \mathsf{a}_2$. We then need to show that

$$\text{If } 0 \vdash s_0^1, l \Downarrow l_1 \text{ and } 0 \vdash s_0^2, l \Downarrow l_2 \text{ then } l_1 = l_2$$

We show the more general result (which entails the above):

$$\begin{array}{l}\text{If } t \vdash s_1, l \Downarrow l_1 \text{ and } t \vdash s_2, l \Downarrow l_2 \text{ and } s_1 \equiv s_2 \\ \text{then } \forall t' \in \mathbb{N}. t \leq t' < \mathrm{eol}(l) \Rightarrow l_1(t') = l_2(t')\end{array}$$

The proof is by induction on the first derivation.

Case e-end: Now $t > \mathrm{eol}(l)$ so the result follows trivially.

Case e-step: Now both derivations must have used the e-step rule:

$$\frac{\delta_{\mathrm{o}}^1(s_1) = m_1 \qquad \delta_{\mathrm{t}}(s_1, l(t)) = s_1' \qquad \overbrace{t+1 \vdash s_1', l \Downarrow l_1}^{(*)}}{t \vdash s_1, l \Downarrow l_1[t \mapsto m_1]}\,(t < \mathrm{eol}(l))$$

$$\frac{\delta_{\mathrm{o}}^2(s_2) = m_2 \qquad \delta_{\mathrm{t}}(s_2, l(t)) = s_2' \qquad \overbrace{t+1 \vdash s_2', l \Downarrow l_2}^{(**)}}{t \vdash s_2, l \Downarrow l_2[t \mapsto m_2]}\,(t < \mathrm{eol}(l))$$

By assumption $s_1 \equiv s_2$ so $m_1 = m_2$, hence it suffices to show that

$$\forall t' \in \mathbb{N}. t+1 \leq t' < \mathrm{eol}(l) \Rightarrow l_1(t') = l_2(t')$$

But this follows by induction on $(*)$ and $(**)$ since $s_1 \equiv s_2$ implies that $s_1' \equiv s_2'$. □

Lemma 1.3.15. *If* $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$ *then* $\mathsf{a}_1 \equiv \mathsf{a}_2$.

Proof. Let $\mathsf{a}_1 = (C_I, C_O, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I, C_O, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ be given and assume that $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$. We then need to find an observational equivalence relation $R \subseteq S^1 \times S^2$ such that $(s_0^1, s_0^2) \in R$. So consider the set

$$R \stackrel{\text{def}}{=} \left\{(s_1, s_2) \mid \exists t \in \mathbb{N}. \forall l, l' \in \mathcal{L}_{C_I}. \left(t \vdash s_1, l \Downarrow l' \Leftrightarrow t \vdash s_2, l \Downarrow l'\right)\right\}$$

We show first that R is an observational equivalence relation. So assume $(s_1, s_2) \in R$ with some witness $t \in \mathbb{N}$. We then need to show

$$\delta_{\mathrm{o}}^1(s_1) = \delta_{\mathrm{o}}^2(s_2) \wedge \forall m \in \mathcal{M}_{C_I}. (\delta_{\mathrm{t}}^1(s_1, m), \delta_{\mathrm{t}}^2(s_2, m)) \in R$$

Let $l \in \mathcal{L}_{C_I}$ be some log with $\mathrm{eol}(l) = t+1$. Then $t \vdash s_1, l \Downarrow l'$ and $t \vdash s_2, l \Downarrow l'$ both using the e-step rule, and hence $\delta^1_{\mathrm{o}}(s_1) = l'(t) = \delta^2_{\mathrm{o}}(s_2)$ as needed.

We now need to show that $\forall m \in \mathcal{M}_{C_I}.(\delta^1_{\mathrm{t}}(s_1, m), \delta^2_{\mathrm{t}}(s_2, m)) \in R$. So let $m \in \mathcal{M}_{C_I}$ be given. We then show

$$\forall l, l' \in \mathcal{L}_{C_I}.\left(t+1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow l' \Leftrightarrow t+1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow l'\right) \tag{B.2}$$

So let $l \in \mathcal{L}_{C_I}$ be given and assume that $t+1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow l'$ and $t+1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow l''$. We then need to show that $l' = l''$.

If $t+1 \geq \mathrm{eol}(l)$ then $l' = l'' = \emptyset$ so assume that $t+1 < \mathrm{eol}(l)$. Now $(s_1, s_2) \in R$ with witness t, so for $l_m = l[t \mapsto m]$ we have that

$$t \vdash s_1, l_m \Downarrow l''' \Leftrightarrow t \vdash s_2, l_m \Downarrow l'''$$

Since $t+1 < \mathrm{eol}(l)$ these derivations must use the e-step rule:

$$\frac{\delta^1_{\mathrm{o}}(s_1) = m_1 \qquad \delta^1_{\mathrm{t}}(s_1, l_m(t)) = s_1' \qquad \overbrace{t+1 \vdash s_1', l_m \Downarrow \hat{l}}^{(*)}}{t \vdash s_1, l_m \Downarrow \hat{l}[t \mapsto m_1]}\,(t < \mathrm{eol}(l_m))$$

$$\frac{\delta^2_{\mathrm{o}}(s_2) = m_2 \qquad \delta^2_{\mathrm{t}}(s_2, l_m(t)) = s_2' \qquad \overbrace{t+1 \vdash s_2', l_m \Downarrow \hat{l}}^{(**)}}{t \vdash s_2, l_m \Downarrow \hat{l}[t \mapsto m_2]}\,(t < \mathrm{eol}(l_m))$$

But $l_m(t) = m$ so it follows that $\delta^1_{\mathrm{t}}(s_1, m) = s_1'$ and $\delta^2_{\mathrm{t}}(s_2, m) = s_2'$. We thus have that

$$t+1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l_m \Downarrow \hat{l}$$
$$t+1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l_m \Downarrow \hat{l}$$

But $l(t') = l_m(t')$ for all $t' \geq t+1$, hence we can replace l_m by l in the relations above (formally a proof by induction on the derivation)

$$t+1 \vdash \delta^1_{\mathrm{t}}(s_1, m), l \Downarrow \hat{l}$$
$$t+1 \vdash \delta^2_{\mathrm{t}}(s_2, m), l \Downarrow \hat{l}$$

But then by the determinism of the big-step relation (Lemma 1.3.11) it follows that $l' = \hat{l} = l''$ which concludes the proof of (B.2)

Now in order to conclude the lemma it suffices to show that $(s^1_0, s^2_0) \in R$. But this is the case since $\ulcorner \mathsf{a}_1 \urcorner = \ulcorner \mathsf{a}_2 \urcorner$ (i.e., witness 0). $\square$

Lemma 1.3.16 ($\ulcorner\cdot\urcorner$ is homomorphic)**.** $\ulcorner\cdot\urcorner : (\mathfrak{A}, \|) \to (\mathfrak{P}, \|)$ *is a (compset) homomorphism.*

Proof. Let $\mathsf{a}_1 = (C_I^1, C_O^1, S^1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ and $\mathsf{a}_2 = (C_I^2, C_O^2, S^2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$ be given. Since $\ulcorner\cdot\urcorner$ preserves input/output channels, $\mathsf{a}_1 \parallel \mathsf{a}_2$ is defined exactly when $\ulcorner\mathsf{a}_1\urcorner \parallel \ulcorner\mathsf{a}_2\urcorner$ is defined – so assume both are defined, and let C_I and C_O denote input channels and output channels respectively for the two compositions (which are the same c.f. Definition 1.2.11 and Definition 1.3.2).

Now $\ulcorner\mathsf{a}_1 \parallel \mathsf{a}_2\urcorner(l) = l' \Leftrightarrow 0 \vdash \langle s_0^1, s_0^2\rangle, l \Downarrow l'$ using the two rules

$$\textsf{e-end}_{1\|2}\ \frac{\forall t' < \mathrm{eol}(l).\, l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash \langle s_1, s_2\rangle, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_{1\|2}\ \frac{(\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_O} = m \qquad \begin{array}{l}\delta_{\mathrm{t}}^1(s_1, (l(t) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^1}) = s_1' \\ \delta_{\mathrm{t}}^2(s_2, (l(t) \cup \delta_{\mathrm{o}}^1(s_1))_{|C_I^2}) = s_2'\end{array} \qquad t+1 \vdash \langle s_1', s_2'\rangle, l \Downarrow l'}{t \vdash \langle s_1, s_2\rangle, l \Downarrow l'[t \mapsto m]}\ (t < \mathrm{eol}(l))$$

The translations of a_1 and a_2 use the rules $\textsf{e-end}_1$, $\textsf{e-step}_1$ and $\textsf{e-end}_2$, $\textsf{e-step}_2$ respectively:

$$\textsf{e-end}_1\ \frac{\forall t' < \mathrm{eol}(l).\, l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash s_1, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_1\ \frac{\delta_{\mathrm{o}}^1(s_1) = m_1 \qquad \delta_{\mathrm{t}}^1(s_1, l(t)) = s_1' \qquad t+1 \vdash s_1', l \Downarrow l'}{t \vdash s_1, l \Downarrow l'[t \mapsto m_1]}\ (t < \mathrm{eol}(l))$$

$$\textsf{e-end}_2\ \frac{\forall t' < \mathrm{eol}(l).\, l_\varepsilon(t') = \varepsilon \qquad l_\varepsilon \in \mathcal{L}_{C_O}^{\mathrm{eol}(l)}}{t \vdash s_2, l \Downarrow l_\varepsilon}\ (t \geq \mathrm{eol}(l))$$

$$\textsf{e-step}_2\ \frac{\delta_{\mathrm{o}}^2(s_2) = m_2 \qquad \delta_{\mathrm{t}}^2(s_2, l(t)) = s_2' \qquad t+1 \vdash s_2', l \Downarrow l'}{t \vdash s_2, l \Downarrow l'[t \mapsto m_2]}\ (t < \mathrm{eol}(l))$$

So for $l \in \mathcal{L}_{C_I}$ we have that $(\ulcorner\mathsf{a}_1\urcorner \parallel \ulcorner\mathsf{a}_2\urcorner)(l) = (\mathcal{I}_N^1 \bowtie \mathcal{I}_N^2)_{|C_O}$ where

$$\begin{aligned}
\mathcal{I}_0^1 &= \emptyset \\
\mathcal{I}_0^2 &= \emptyset \\
\mathcal{I}_{n+1}^1 &= l' \text{ where } 0 \vdash s_0^1, (\mathcal{I}_n^2 \bowtie l)_{|C_I^1} \Downarrow l' \text{ (using } \textsf{e-end}_1 \text{ and } \textsf{e-step}_1) \\
\mathcal{I}_{n+1}^2 &= l'' \text{ where } 0 \vdash s_0^2, (\mathcal{I}_n^1 \bowtie l)_{|C_I^2} \Downarrow l'' \text{ (using } \textsf{e-end}_2 \text{ and } \textsf{e-step}_2)
\end{aligned}$$

and N is such that $\mathcal{I}_N^1 = \mathcal{I}_{N+1}^1$ and $\mathcal{I}_N^2 = \mathcal{I}_{N+1}^2$.

We show the following generalization:

If	$t \vdash s_1, (l \bowtie A)_{\mid C_I^1} \Downarrow B$	(using $\mathsf{e\text{-}end}_1$ and $\mathsf{e\text{-}step}_1$)
and	$t \vdash s_2, (l \bowtie C)_{\mid C_I^2} \Downarrow D$	(using $\mathsf{e\text{-}end}_2$ and $\mathsf{e\text{-}step}_2$)
and	$t \vdash \langle s_1, s_2 \rangle, l \Downarrow E$	(using $\mathsf{e\text{-}end}_{1\|2}$ and $\mathsf{e\text{-}step}_{1\|2}$)
and	$\mathrm{eol}(A) = \mathrm{eol}(C) = \mathrm{eol}(l)$	
and	$\forall t' \in \mathbb{N}.t \leq t' < \mathrm{eol}(l) \Rightarrow B(t') = C(t') \wedge D(t') = A(t')$	
then	$\forall t' \in \mathbb{N}.t \leq t' < \mathrm{eol}(l) \Rightarrow E(t') = (B \bowtie D)_{\mid C_O}(t')$	

The proof is by induction on $n = \mathrm{eol}(l) - t$.

Case $n = 0$: In this case the result follows trivially.

Case $n > 0$: Now $\mathrm{eol}(l) = \mathrm{eol}((l \bowtie A)_{\mid C_I^1}) = \mathrm{eol}((l \bowtie C)_{\mid C_I^2}) > t$, so the three derivations must have used the rules $\mathsf{e\text{-}step}_1$, $\mathsf{e\text{-}step}_2$ and $\mathsf{e\text{-}step}_{1\|2}$ respectively:

$$\frac{\delta_{\mathrm{o}}^1(s_1) = m_1 \qquad \delta_{\mathrm{t}}^1(s_1, (l \bowtie A)_{\mid C_I^1}(t)) = s_1' \qquad \overbrace{t+1 \vdash s_1', (l \bowtie A)_{\mid C_I^1} \Downarrow B'}^{(*)}}{t \vdash s_1, (l \bowtie A)_{\mid C_I^1} \Downarrow B'[t \mapsto m_1]}\ (t < \mathrm{eol}(l))$$

$$\frac{\delta_{\mathrm{o}}^2(s_2) = m_2 \qquad \delta_{\mathrm{t}}^2(s_2, (l \bowtie C)_{\mid C_I^2}(t)) = s_2' \qquad \overbrace{t+1 \vdash s_2', (l \bowtie C)_{\mid C_I^2} \Downarrow D'}^{(**)}}{t \vdash s_2, (l \bowtie C)_{\mid C_I^2} \Downarrow D'[t \mapsto m_2]}\ (t < \mathrm{eol}(l))$$

$$\frac{(\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{\mid C_O} = m \qquad \begin{array}{l}\delta_{\mathrm{t}}^1(s_1, (l(t) \cup \delta_{\mathrm{o}}^2(s_2))_{\mid C_I^1}) = s_1'' \\ \delta_{\mathrm{t}}^2(s_2, (l(t) \cup \delta_{\mathrm{o}}^1(s_1))_{\mid C_I^2}) = s_2''\end{array} \qquad \overbrace{t+1 \vdash \langle s_1'', s_2'' \rangle, l \Downarrow E'}^{(***)}}{t \vdash \langle s_1, s_2 \rangle, l \Downarrow E'[t \mapsto m]}\ (t < \mathrm{eol}(l))$$

We first show that $E(t) = (B \bowtie D)_{\mid C_O}(t)$:

$$\begin{aligned}(B \bowtie D)_{\mid C_O}(t) &= (B(t) \cup D(t))_{\mid C_O} \\ &= (\delta_{\mathrm{o}}^1(s_1) \cup \delta_{\mathrm{o}}^2(s_2))_{\mid C_O} \\ &= E(t)\end{aligned}$$

So in order to show the generalized induction hypothesis it suffices to show that

$$\forall t' \in \mathbb{N}.t+1 \leq t' < \mathrm{eol}(l) \Rightarrow E(t') = (B \bowtie D)_{\mid C_O}(t')$$

Which will follow if we can show that

$$\forall t' \in \mathbb{N}.t+1 \leq t' < \mathrm{eol}(l) \Rightarrow E'(t') = (B' \bowtie D')_{\mid C_O}(t')$$

But this follows from the induction hypothesis applied to $(*)$, $(**)$ and $(***)$ if we can show that (i) $\forall t'.\, t+1 \leq t' < \mathrm{eol}(l) \Rightarrow B'(t') = C(t') \wedge D'(t') = A(t')$ and (ii) $\langle s_1', s_2' \rangle = \langle s_1'', s_2'' \rangle$.

(i) Follows from the assumption because $B'[t \mapsto m_1] = B$ and $D'[t \mapsto m_2] = A$.

(ii) We have that

$$\begin{aligned}
s_1' &= \delta_{\mathrm{t}}^1(s_1, (l \bowtie A)_{|C_I^1}(t)) && \text{(by definition)}\\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup A(t))_{|C_I^1}) \\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup D(t))_{|C_I^1}) && \text{(by assumption)}\\
&= \delta_{\mathrm{t}}^1(s_1, (l(t) \cup \delta_{\mathrm{o}}^2(s_2))_{|C_I^1}) && \text{(by definition)}\\
&= s_1'' && \text{(by definition)}
\end{aligned}$$

By an analogue argument it can be shown that $s_2' = s_2''$ hence the generalized lemma follows.

The result of the lemma now follows from the generalized induction hypothesis with $A = \mathcal{I}_N^2$, $B = \mathcal{I}_{N+1}^1$, $C = \mathcal{I}_N^1$ and $D = \mathcal{I}_{N+1}^2$. □

Lemma 1.3.18 ($\ulcorner \cdot \urcorner$ is the left inverse of $\llcorner \cdot \lrcorner$)**.** $\forall \mathsf{p} \in \mathfrak{P}.\ulcorner \llcorner \mathsf{p} \lrcorner \urcorner = \mathsf{p}$

Proof. Let $\mathsf{p} = (C_I, C_O, f) \in \mathfrak{P}$ be given. Then $\llcorner \mathsf{p} \lrcorner = (C_I, C_O, \mathcal{L}_{C_I}, \emptyset, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$, where

$$\begin{aligned}
\delta_{\mathrm{o}}(l) &= f(l \mathbin{@} m_d)(\mathrm{eol}(l)) \\
\delta_{\mathrm{t}}(l, m) &= l \mathbin{@} m
\end{aligned}$$

Hence $\ulcorner \llcorner \mathsf{p} \lrcorner \urcorner(l) = l' \Leftrightarrow 0 \vdash \emptyset, l \Downarrow l'$, so we need to show that $f(l) = l' \Leftrightarrow 0 \vdash \emptyset, l \Downarrow l'$. We show the following generalization:

$$\begin{aligned}
&\text{If } t \vdash l_1, l_2 \Downarrow l_3 \text{ and } l_1 = l_{2|t} \\
&\text{then } \forall t' \in \mathbb{N}.t \leq t' < \mathrm{eol}(l_2) \Rightarrow l_3(t') = f(l_2)(t')
\end{aligned}$$

The proof is by induction on the derivation of $t \vdash l_1, l_2 \Downarrow l_3$.

Case e-end: Now $t \geq \mathrm{eol}(l_2)$ so the result follows trivially.

Case e-step:

$$\frac{\delta_{\mathrm{o}}(l_1) = m_1 \qquad \delta_{\mathrm{t}}(l_1, l_2(t)) = l_1' \qquad \overbrace{t + 1 \vdash l_1', l_2 \Downarrow l_3'}^{(*)}}{t \vdash l_1, l_2 \Downarrow l_3'[t \mapsto m_1]}\ (t < \mathrm{eol}(l_2))$$

By assumption $l_1 = l_{2|t}$ hence $\mathrm{eol}(l_1) = t$. Now it follows from the definition of δ_{o} that $m_1 = f(l_1 \mathbin{@} m_d)(t)$. Furthermore $(l_1 \mathbin{@} m_d)_{|t} = l_{2|t}$ so by strict monotonicity it follows that $f(l_1 \mathbin{@} m_d)(t) = f(l_2)(t)$. So $m_1 = f(l_2)(t)$ and hence

$$l_3'[t \mapsto m_1](t) = f(l_2)(t)$$

So it suffices to show that

$$\forall t' \in \mathbb{N}. t+1 \leq t' < \mathrm{eol}(l_2) \Rightarrow l_3'(t') = f(l_2)(t')$$

But this follows from the induction hypothesis applied to $(*)$ if we can show that $l_1' = l_{2|t+1}$. By definition of δ_{t} we have that

$$l_1' = l_1 \mathbin{@} l_2(t)$$

so since $l_1 = l_{2|t}$ the result follows.

The lemma now follows from the generalization since from $0 \vdash \emptyset, l \Downarrow l'$ we get that $\emptyset = l_{|0}$ and hence

$$\forall t' \in \mathbb{N}. 0 \leq t' < \mathrm{eol}(l) \Rightarrow l'(t') = f(l)(t')$$

meaning exactly $l' = f(l)$ as desired. □

Theorem 1.5.13 (Soundness of projection)**.** *Consider a contract portfolio $C = \{c_1, \ldots, c_n\}$ for P and routing $r = (r_1, r_2, r_3)$ for C and input/output channels C_I/C_O. If*

$$C \rightsquigarrow^{\theta} \mathsf{C}$$

where θ is a renaming map for r, and

$$\models \mathsf{a} : \mathsf{C}$$

for an automaton $\mathsf{a} = (C_I, C_O, S, s_0, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$, then

$$\models (\mathsf{a}, r) : C$$

Proof. We will use the following definitions and abbreviations throughout the proof:

$$\begin{aligned}
c_i &= (\Lambda_{\mathsf{PA}_i}, \Lambda_{\mathsf{A}_i\mathsf{P}}, G_i, g_i, \rho_i) \\
\mathsf{c}_i &= (C_i, G_i, g_i, \rho_i') \\
\Lambda_I &= \bigcup_{i=1}^{n} \Lambda_{\mathsf{A}_i\mathsf{P}} \\
\Lambda_O &= \bigcup_{i=1}^{n} \Lambda_{\mathsf{PA}_i} \\
\Lambda &= \Lambda_I \cup \Lambda_O \\
C_E &= (\bigcup_{i=1}^{n} C_i) \setminus C_O
\end{aligned}$$

From the definition of automaton contract conformance we get that there exists a conformance relation R:

$$R \subseteq \mathbb{Q} \times S \times G_1 \times \ldots \times G_n$$

for a and $\mathsf{C} = \{\mathsf{c}_1, \ldots, \mathsf{c}_n\}$, which relates the start states. We must find a conformance relation R' for $i = (\mathsf{a}, r)$ and $C = \{c_1, \ldots, c_n\}$, so we claim that $R' = R$ is such a conformance relation. Clearly it relates the starting states, so if we can show that R' is a conformance relation for i and C, then the lemma follows.

So assume $(k, s, g_1, \ldots, g_n) \in R'$ and let $m \in \mathcal{M}_{\Lambda_I}$ be given. Now let

$$(g_i', k_i) = \rho_i(g_i, \overline{r}_i(\delta_{\mathrm{o}}(s), m), m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}) \tag{B.3}$$

for $i = 1, \ldots, n$. We must then show:

$$\sum_{i=1}^{n} k_i \geq k \tag{B.4}$$

$$(k - \sum_{i=1}^{n} k_i, \delta_{\mathrm{t}}(s, m \circ r_1), g_1', \ldots, g_n') \in R' \tag{B.5}$$

In order to show this, we construct a move $\widehat{m} \in \mathcal{M}_{C_E}$, which can be used in the automaton conformance relation R:

$$\widehat{m}(\alpha) \stackrel{\text{def}}{=} m(\lambda), \text{ where } \lambda \in \Lambda_I \wedge \theta(\lambda) = \alpha$$

First we must show that this is well-defined, i.e., that (1) there always exists such a λ, and (2) the function value is unique.

1. Let $\alpha \in C_E$: then there exists $\lambda \in \Lambda$ such that $\theta(\lambda) = \alpha$ by construction of the projection. If $\lambda \in \Lambda_I$ we are done, so assume $\lambda \in \Lambda_O$. If $r_2(\alpha') = \lambda$ then $\alpha = \theta(\lambda) = \theta(r_2(\alpha')) = \alpha'$ in contradiction with $\alpha \notin C_O$. Therefore $r_3(\lambda') = \lambda$, and then $\theta(\lambda') = \theta(\lambda) = \alpha$ with $\lambda' \in \Lambda_I$.
2. Let $\theta(\lambda) = \theta(\lambda')$ with $\lambda, \lambda' \in \Lambda_I$ then by the fourth condition for renaming we have either $\lambda = \lambda'$ or $r_3(\lambda) = \lambda'$ or $r_3(\lambda') = \lambda$. But the last two cannot happen, as that would imply that either $\lambda \in \Lambda_O$ or $\lambda' \in \Lambda_O$, hence $\lambda = \lambda'$.

We can now use $\widehat{m}$ in relation R because $C_I \subseteq C_E$ (Observation 1.5.11), so let

$$\begin{aligned}(g_i'', k_i'') &= \rho_i'(g_i, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i}) \\ &= \rho_i(g_i, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{P}\mathsf{A}_i}}, (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}})\end{aligned} \tag{B.6}$$

We now show that

$$\overline{r}_i(\delta_{\mathrm{o}}(s), m) = (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{PA}_i}} \tag{B.7}$$

$$m_{|\Lambda_{\mathsf{A}_i\mathsf{P}}} = (\widehat{m} \cup \delta_{\mathrm{o}}(s))_{|C_i} \circ \theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}} \tag{B.8}$$

ad. B.7) Now

$$\overline{r}_i(\delta_{\mathrm{o}}(s), m)(\lambda) = \begin{cases} \delta_{\mathrm{o}}(s)(\alpha) & \text{, if } r_2(\alpha) = \lambda \\ m(\lambda') & \text{, if } r_3(\lambda') = \lambda \end{cases}$$

So assume that $r_2(\alpha) = \lambda$. Then LHS $= \delta_{\mathrm{o}}(s)(\alpha)$. Now by the second requirement for renaming maps it follows that $\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda) = \alpha$, so also RHS $= \delta_{\mathrm{o}}(s)(\alpha)$.

Now assume that $r_3(\lambda') = \lambda$. Then LHS $= m(\lambda')$. Now by the third requirement for renaming maps it follows that $\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda) = \theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda') \notin C_O$, hence RHS $= \widehat{m}(\theta_{|\Lambda_{\mathsf{PA}_i}}(\lambda')) = m(\lambda')$ as required.

ad. B.8) Now let $\lambda \in \Lambda_{\mathsf{A}_i\mathsf{P}}$ be given. Then $\theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}(\lambda) \notin C_O$ so RHS $= \widehat{m}(\theta_{|\Lambda_{\mathsf{A}_i\mathsf{P}}}(\lambda)) = m(\lambda) =$ LHS as required.

We have now established (B.7) and (B.8), from which it follows that $g_i' = g_i''$ and $k_i = k_i''$ for $i = 1, \ldots, n$, cf. (B.3) and (B.6). Hence (B.4) follows from the fact that $\sum_{i=1}^{n} k_i'' \geq k$ (the conformance relation R). Now in order to show (B.5), it suffices to show that

$$\delta_{\mathrm{t}}(s, m \circ r_1) = \delta_{\mathrm{t}}(s, \widehat{m}_{|C_I})$$

which means that we must show that

$$(m \circ r_1)(\alpha) = \widehat{m}_{|C_I}(\alpha)$$

for all $\alpha \in C_I$. But this follows from the first condition of renaming maps, and hence the lemma follows. □

Theorem 1.5.17 (Contract conformance is compositional)**.** *Consider two automata* $\mathsf{a}_1 = (C_I^1, C_O^1, S_1, s_0^1, \delta_{\mathrm{o}}^1, \delta_{\mathrm{t}}^1)$ *and* $\mathsf{a}_2 = (C_I^2, C_O^2, S_2, s_0^2, \delta_{\mathrm{o}}^2, \delta_{\mathrm{t}}^2)$*, where parallel composition*

$$\mathsf{a}_1 \parallel \mathsf{a}_2 = (C_I, C_O, S_1 \times S_2, \langle s_1, s_2 \rangle, \delta_{\mathrm{o}}, \delta_{\mathrm{t}})$$

is defined (Definition 1.3.2). If

$$\begin{gathered} \models \mathsf{a}_1 : \mathsf{c}_1, \ldots, \mathsf{c}_n, \mathsf{c}_1', \ldots, \mathsf{c}_{n_1}' \\ \models \mathsf{a}_2 : \overline{\mathsf{c}_1}, \ldots, \overline{\mathsf{c}_n}, \mathsf{c}_1'', \ldots, \mathsf{c}_{n_2}'' \\ \mathsf{chan}(\{\mathsf{c}_1', \ldots, \mathsf{c}_{n_1}', \mathsf{c}_1'', \ldots, \mathsf{c}_{n_2}''\}) \cap C_{\mathrm{int}} = \emptyset \end{gathered}$$

(where C_{int} *is the internal channels of* $\mathsf{a}_1 \parallel \mathsf{a}_2$*), then*

$$\models \mathsf{a}_1 \parallel \mathsf{a}_2 : \mathsf{c}_1', \ldots, \mathsf{c}_{n_1}', \mathsf{c}_1'', \ldots, \mathsf{c}_{n_2}''$$

Proof. We are given two conformance relations:

$$R_1 \subseteq \mathbb{Q} \times S_1 \times G_1 \times \ldots \times G_n \times G'_1 \times \ldots \times G'_{n_1}$$
$$R_2 \subseteq \mathbb{Q} \times S_2 \times G_1 \times \ldots \times G_n \times G''_1 \times \ldots \times G''_{n_2}$$

which relates the start states. We wish to construct a new conformance relation:

$$R \subseteq \mathbb{Q} \times (S_1 \times S_2) \times G'_1 \times \ldots \times G'_{n_1} \times G''_1 \times \ldots \times G''_{n_2}$$

which relates the start states in the parallel composition. We define R in the following way:

$$\begin{aligned}(k, \langle s_1, s_2\rangle, g'_1, \ldots, g'_{n_1}, g''_1, \ldots, g''_{n_2}) \in R \stackrel{\text{def}}{\Longleftrightarrow}\\ \exists (g_1, \ldots, g_n) \in G_1 \times \ldots \times G_n.\ \exists k_1, k_2 \in \mathbb{Q}.\\ k = k_1 + k_2 \wedge\\ (k_1, s_1, g_1, \ldots, g_n, g'_1, \ldots, g'_{n_1}) \in R_1 \wedge\\ (k_2, s_2, g_1, \ldots, g_n, g''_1, \ldots, g''_{n_2}) \in R_2\end{aligned}$$

It is clear that R relates the start states, because R_1 and R_2 do so. So we need to show that R is a conformance relation.

So assume that $(k, \langle s_1, s_2\rangle, g'_1, \ldots, g'_{n_1}, g''_1, \ldots, g''_{n_2}) \in R$. This means that there exists $k_1, k_2, g_1, \ldots, g_n$ such that:

$$k = k_1 + k_2 \tag{B.9}$$
$$(k_1, s_1, g_1, \ldots, g_n, g'_1, \ldots, g'_{n_1}) \in R_1 \tag{B.10}$$
$$(k_2, s_2, g_1, \ldots, g_n, g''_1, \ldots, g''_{n_2}) \in R_2 \tag{B.11}$$

Now put for easy reference:

$$C = \bigcup_{i=1}^{n} C_i$$
$$C' = \bigcup_{i=1}^{n_1} C'_i$$
$$C'' = \bigcup_{i=1}^{n_2} C''_i$$

So let $m \in \mathcal{M}_{(C_I \cup C' \cup C'') \setminus C_O}$ be given, and let

$$(\underline{g'_i}, \underline{k'_i}) = \rho'_i(g'_i, (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2\rangle))_{|C'_i}), \text{ for } i = 1 \ldots n_1 \tag{B.12}$$
$$(\underline{g''_i}, \underline{k''_i}) = \rho''_i(g''_i, (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2\rangle))_{|C''_i}), \text{ for } i = 1 \ldots n_2 \tag{B.13}$$

We must then show:

$$\sum_{i=1}^{n_1} \underline{k'_i} + \sum_{i=1}^{n_2} \underline{k''_i} \geq k \tag{B.14}$$

$$(k - \sum_{i=1}^{n_1} \underline{k'_i} + \sum_{i=1}^{n_2} \underline{k''_i}, \delta_{\mathrm{t}}(\langle s_1, s_2 \rangle, m_{|C_I}), \underline{g'_1}, \ldots, \underline{g'_{n_1}}, \underline{g''_1}, \ldots, \underline{g''_{n_2}}) \in R \tag{B.15}$$

We now want to construct $m_1 \in \mathcal{M}_{(C_I^1 \cup C \cup C') \setminus C_O^1}$ and $m_2 \in \mathcal{M}_{(C_I^2 \cup C \cup C'') \setminus C_O^2}$ to use in R_1 and R_2. The idea is to define m_1 to behave like (1) m where this is possible, (2) to the value of a_2's output on C_O^2, and (3) to some arbitrary value on the rest. m_2 will be defined in a similar way.

To make this precise we see that because $C' \cap C_{\mathrm{int}} = \emptyset$ and $C'' \cap C_{\mathrm{int}} = \emptyset$ then (we call the domain of m for D):

$$\begin{aligned} D &= (C_I \cup C' \cup C'') \setminus C_O \\ &= ((C_I^1 \cup C_I^2) \setminus C_{\mathrm{int}} \cup C' \cup C'') \setminus (C_O^1 \cup C_O^2 \setminus C_{\mathrm{int}}) \\ &= ((C_I^1 \cup C_I^2 \cup C' \cup C'') \setminus C_{\mathrm{int}}) \setminus (C_O^1 \cup C_O^2 \setminus C_{\mathrm{int}}) \\ &= (C_I^1 \cup C_I^2 \cup C' \cup C'') \setminus (C_O^1 \cup C_O^2 \cup C_{\mathrm{int}}) \end{aligned}$$

Hence the elements $\alpha \in (C_I^1 \cup C \cup C') \setminus C_O^1$ which are not in D (i.e. those α on which m_1 cannot agree with m), can have two possible forms:

1. $\alpha \in C_O^2$.
2. $\alpha \notin C_O^2 \wedge \alpha \notin C_{\mathrm{int}} \wedge \alpha \in C$.

(The case $\alpha \notin C_O^2 \wedge \alpha \in C_{\mathrm{int}}$ cannot occur because $\alpha \notin C_O^1$.) Now for each channel α in the second case we choose some (fixed) value $x_\alpha \in \mathcal{A}_\alpha$ and set the value of m_1 to that value. So in summary we get the following definition of m_1:

$$m_1(\alpha) = \begin{cases} m(\alpha) & \text{if } \alpha \in D \\ \delta_{\mathrm{o}}^2(s_2)(\alpha) & \text{if } \alpha \in C_O^2 \\ x_\alpha & \text{otherwise} \end{cases}$$

(The three cases correspond to (1), (2), and (3) mentioned above). By a similar analysis m_2 is defined by:

$$m_2(\alpha) = \begin{cases} m(\alpha) & \text{if } \alpha \in D \\ \delta_{\mathrm{o}}^1(s_1)(\alpha) & \text{if } \alpha \in C_O^1 \\ x_\alpha & \text{otherwise} \end{cases}$$

We now apply m_1 in conformance relation R_1 (B.10):

$$(\underline{\underline{g_i}}, \underline{\underline{k_i}}) = \rho_i(g_i, (m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C_i}), \text{ for } i = 1 \ldots n \tag{B.16}$$

$$(\underline{\underline{g'_i}}, \underline{\underline{k'_i}}) = \rho'_i(g'_i, (m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C'_i}), \text{ for } i = 1 \ldots n_1 \tag{B.17}$$

and m_2 in conformance relation R_2 (B.11):

$$(\underline{\underline{g_i}}, \underline{\underline{k_i}}) = \overline{\rho_i}(g_i, (m_2 \cup \delta^2_{\mathrm{o}}(s_2))_{|C_i}), \text{ for } i = 1 \ldots n \tag{B.18}$$

$$(\underline{\underline{g''_i}}, \underline{\underline{k''_i}}) = \rho''_i(g''_i, (m_2 \cup \delta^2_{\mathrm{o}}(s_2))_{|C''_i}), \text{ for } i = 1 \ldots n_2 \tag{B.19}$$

In order to show (B.14) and (B.15), we first show three equalities:

(I) $(m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C'_i} = (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C'_i}$, for $i = 1, \ldots, n_1$

(II) $(m_2 \cup \delta^2_{\mathrm{o}}(s_2))_{|C''_i} = (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C''_i}$, for $i = 1, \ldots, n_2$

(III) $(m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C_i} = (m_2 \cup \delta^2_{\mathrm{o}}(s_2))_{|C_i}$, for $i = 1, \ldots, n$

ad. I) Let $\alpha \in C'_i$, and consider the possible cases:

- $\alpha \in C^1_O$: Here we must have that $\alpha \in C_O$ because $C_{\mathrm{int}} \cap C'_i = \emptyset$, and then:

$$\begin{aligned}(m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C'_i}(\alpha) &= \delta^1_{\mathrm{o}}(s_1)(\alpha) \\ &= \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle)(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C'_i}(\alpha)\end{aligned}$$

- $\alpha \in C^2_O$: Here we must again have that $\alpha \in C_O$, and then:

$$\begin{aligned}(m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C'_i}(\alpha) &= m_1(\alpha) \\ &= \delta^2_{\mathrm{o}}(s_2)(\alpha) \\ &= \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle)(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C'_i}(\alpha)\end{aligned}$$

- $\alpha \in D$: Here we have:

$$\begin{aligned}(m_1 \cup \delta^1_{\mathrm{o}}(s_1))_{|C'_i}(\alpha) &= m_1(\alpha) \\ &= m(\alpha) \\ &= (m \cup \delta_{\mathrm{o}}(\langle s_1, s_2 \rangle))_{|C'_i}(\alpha)\end{aligned}$$

ad. II) Similar to the proof of (I).

ad. III) Follows directly by construction of m_1 and m_2.

Based on the equalities above, we can conclude the following:

(A) By (I) and definitions (B.12) and (B.17): $\underline{g'_i} = \underline{\underline{g'_i}}$ and $\underline{k'_i} = \underline{\underline{k'_i}}$ for $i = 1, \ldots, n_1$.

(B) By (II) and definitions (B.13) and (B.19): $\underline{g''_i} = \underline{\underline{\underline{g''_i}}}$ and $\underline{k''_i} = \underline{\underline{\underline{k''_i}}}$ for $i = 1, \ldots, n_2$.

(C) By (III) and definitions (B.16) and (B.18): $\underline{\underline{g_i}} = \underline{\underline{g_i}}$ and $\underline{\underline{k_i}} = -\underline{\underline{k_i}}$ for $i = 1, \ldots, n$.

And now finally the proof of (B.14):

$$\begin{aligned}
\sum_{i=1}^{n_1} \underline{k'_i} + \sum_{i=1}^{n_2} \underline{k''_i} &= \sum_{i=1}^{n_1} \underline{\underline{k'_i}} + \sum_{i=1}^{n_2} \underline{\underline{\underline{k''_i}}} && \text{(by A and B)} \\
&= \sum_{i=1}^{n_1} \underline{\underline{k'_i}} + \left(\sum_{i=1}^{n} \underline{\underline{k_i}} + \sum_{i=1}^{n} \underline{\underline{k_i}} \right) + \sum_{i=1}^{n_2} \underline{\underline{\underline{k''_i}}} && \text{(by C)} \\
&= \left(\sum_{i=1}^{n_1} \underline{\underline{k'_i}} + \sum_{i=1}^{n} \underline{\underline{k_i}} \right) + \left(\sum_{i=1}^{n} \underline{\underline{k_i}} + \sum_{i=1}^{n_2} \underline{\underline{\underline{k''_i}}} \right) \\
&\geq k_1 + k_2 && \text{(by B.10 and B.11)} \\
&= k && \text{(by B.9)}
\end{aligned}$$

And in order to prove (B.15), it follows from the equalities in (A), (B), and (C) that it suffices to show that

$$\delta_{\mathrm{t}}(\langle s_1, s_2 \rangle, m_{|C_I}) = \langle \delta^1_{\mathrm{t}}(s_1, m_{1|C^1_I}), \delta^2_{\mathrm{t}}(s_2, m_{2|C^2_I}) \rangle$$

which follows by definition of m_1 and m_2.

This concludes the proof of the theorem. □

Appendix C

Sales Contract Template

The following is a sample sales contract template downloaded from FindLaw[1] (see copyright disclaimer on the next page).

[1] http://www.findlaw.com.

MAKING BUSINESS CONTRACTS

SAMPLE SALES CONTRACT

Although all contracts may—in fact *should*—vary in order accurately to reflect the intent of the parties in particular circumstances, the following sales contract is a sample of what such contracts may look like. It is intended to be a starting point and a guide to help you and your attorney create a contract that includes all of the terms relevant to your business interactions.

CONTRACT FOR THE SALE OF GOODS

Paragraph 1. _____, hereinafter referred to as Seller, and _____, hereinafter referred to as Buyer, hereby agree on this ____ day of _____, in the year _____, to the following terms.

A. Identities of the Parties

Paragraph 2. Seller, whose business address is _____, in the city of _____, state of _____, is in the business of _____. Buyer, whose business address is _____, in the city of _____, state of _____, is in the business of _____.

B. Description of the Goods

Paragraph 3. Seller agrees to transfer and deliver to Buyer, on or before _____ [date], the below-described goods:
_____
_____
_____
_____
____

C. Buyer's Rights and Obligations

Paragraph 4. Buyer agrees to accept the goods and pay for them according to the terms further set out below.

Paragraph 5. Buyer agrees to pay for the goods:

In full upon receipt

In installments, as billed by Seller, and subject to the separate installment sale contract of _____[date] between Seller and Buyer.

Half upon receipt, with the remainder due within 30 days of delivery.

Paragraph 6. Goods are deemed received by Buyer upon delivery to Buyer's address as set forth above.

Paragraph 7. Buyer has the right to examine the goods upon receipt and has ____ days in which to notify seller of any claim for damages based on the condition, grade, quality or quality of the goods. Such notice must specify in detail the particulars of the claim. Failure to provide such notice within the requisite time period constitutes irrevocable acceptance of the goods.

D. Seller's Obligations

Paragraph 8. Until received by Buyer, all risk of loss to the above-described goods is borne by Seller.

Paragraph 9. Seller warrants that the goods are free from any and all security interests, liens, and encumbrances.

E. Attestation

Paragraph 10. Agreed to this _____ day of _____, in the year _____.

By: _____ Official Title: _____

On behalf of _____, Seller

I certify that I am authorized to act and sign on behalf of Seller and that Seller is bound by my actions. _____ [initial]

By: _____ Official Title: _____

On behalf of _____, Buyer

I certify that I am authorized to act and sign on behalf of Buyer and that Buyer is bound by my actions. _____ [initial]

[NOTARY STAMP HERE]

Appendix D

Sales Contract

The contract below is an instance of the template contract from Appendix C, extended with a contrary-to-duty obligation (Paragraph 11).

Paragraph 1. YourBooks.com, hereinafter referred to as Seller, and Tom Hvitved, hereinafter referred to as Buyer, hereby agree on this Monday of August 31st, in the year 2009, to the following terms.

Paragraph 2. Seller, whose business address is Universitetsparken 1, 2300 Copenhagen E, in the city of Copenhagen, is in the business of book selling. Buyer, whose business address is Rektorparken 16, 2450 Copenhagen SV, in the city of Copenhagen, is in the business of researching.

Paragraph 3. Seller agrees to transfer and deliver to Buyer, on or before 1st of September 2009, the below-described goods:
1 book titled: "Introducing Game Theory and Its Applications"

Paragraph 4. Buyer agrees to accept the goods and pay for them according to the terms further set out below.

Paragraph 5. Buyer agrees to pay for the goods: Half upon receipt (150 DKK), with the remainder (150 DKK) due within 30 days of delivery.

Paragraph 6. Goods are deemed received by Buyer upon delivery to Buyer's address as set forth above.

Paragraph 7. Buyer has the right to examine the goods upon receipt and has 14 days in which to notify seller of any claim for damages based on the condition, grade, or quality of the goods. Such notice must specify in detail the particulars of the claim. Failure to provide such notice within the requisite time period constitutes irrevocable acceptance of the goods.

Paragraph 8. Until received by Buyer, all risk of loss to the above-described goods is borne by Seller.

Paragraph 9. Seller warrants that the goods are free from any and all security interests, liens, and encumbrances.

Paragraph 10. Agreed to this Monday of August 31st, in the year 2009.

Paragraph 11. If Seller fails to deliver to Buyer, as described in Paragraph 3, then seller has to pay a penalty of 10 percent of the total order price to Buyer (30 DKK), on or before 2nd of September 2009. The contract is hereafter terminated.

Bibliography

[1] 3gERP. 3gERP – Application to Højteknologifonden. `http://www.3gerp.org/Documents/HTF-application.pdf`, 2007.

[2] Samson Abramsky and Radha Jagadeesan. Games and Full Completeness for Multiplicative Linear Logic (Extended Abstract). In *Proceedings of the 12th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 291–301, London, UK, 1992. Springer-Verlag. ISBN 3-540-56287-7.

[3] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994. ISSN 0304-3975.

[4] Jesper Andersen, Ebbe Elsborg, Fritz Henglein, Jakob Grue Simonsen, and Christian Stefansen. Compositional specification of commercial contracts. *International Journal on Software Tools for Technology Transfer (STTT)*, 8(6):485–516, 2006. ISSN 1433-2779.

[5] Ralph-Johan Back and Joakim von Wright. *Refinement Calculus: A Systematic Introduction*. Texts in Computer Science. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998. ISBN 0387984178.

[6] Ralph-Johan Back and Joakim von Wright. Contracts, Games and Refinement. *Inf. Comput.*, 156(1-2):25–45, 2000. ISSN 0890-5401.

[7] T. J. M. Bench-Capon and F. P. Coenen. Isomorphism and Legal Knowledge Based Systems. *Artificial Intelligence and Law*, 1(1):65–86, 1992.

[8] Arthur J. Bernstein and Michael Kifer. *Databases and Transaction Processing: An Application-Oriented Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001. ISBN 0201708728.

[9] Azer Bestavros. The Input Output Timed Automaton: A model for real-time parallel computation. In *Proceedings of 1990 ACM Intl. Workshop on Timing Issues in the Specification and Synthesis of Digital Systems (Tau '90), Vancouver, Canada, August 1990*, 1990.

[10] Abdel Boulmakoul and Mathias Sallé. Integrated contract management. In *Proceedings of the 9th workshop of HP Openview University Association*, 2002.

[11] Giuseppe Castagna, Nils Gesbert, and Luca Padovani. A theory of contracts for web services. In *POPL '08: Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 261–272. ACM, 2008.

[12] Pierre-Louis Curien. Notes on game semantics. `http://www.pps.jussieu.fr/~curien/Game-semantics.pdf`, 2006.

[13] Chris Exton and Jian Chen. Programming by Contract in a Distributed Object Environment. In *Proceedings of the International Symposium on Future Software Technology (ISFST-96)*, pages 272 – 278. Software Engineers Association (Japan), October 1996.

[14] Stephen Fenech, Gordon J. Pace, and Gerardo Schneider. Automatic Conflict Detection on Contracts. In *6th International Colloquium on Theoretical Aspects of Computing (ICTAC'09)*, LNCS, Kuala Lumpur, Malaysia, August 2009. Springer.

[15] Content Reference Forum. Contract Expression Language (CEL) – An UN/CEFACT BCF Compliant Technology. Technical report, Content Reference Forum, 2004.

[16] Andrew Goodchild, Charles Herring, and Zoran Milosevic. Business Contracts for B2B. In *Proceedings of the CAISE '00 Workshop on Infrastructure for Dynamic Business-to-Business Service Outsourcing (ISDO 2000)*, 2000.

[17] Guido Governatori. Representing Business Contracts in RuleML. *International Journal of Cooperative Information Systems*, 14:181–216, 2005.

[18] Guido Governatori and Zoran Milosevic. Dealing with contract violations: formalism and domain specific language. In *Ninth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2005)*, pages 46–57, 2005.

[19] Guido Governatori and Zoran Milosevic. A Formal Analysis of a Business Contract Language. *Int. J. Cooperative Inf. Syst.*, 15(4):659–685, 2006.

[20] Guido Governatori and Duy Hoang Pham. DR-CONTRACT: An Architecture for e-Contracts in Defeasible Logic. *International Journal of Business Process Integration and Management*, 5(4), 2009.

[21] Guido Governatori and Antonio Rotolo. Logic of Violations: A Gentzen System for Reasoning with Contrary-To-Duty Obligations. *The Australasian Journal of Logic*, 4:193–215, 2006.

[22] Guido Governatori, Zoran Milosevic, and Shazia Wasim Sadiq. Compliance checking between business processes and business contracts. In *Tenth IEEE International Enterprise Distributed Object Computing Conference (EDOC 2006)*, pages 221–232, 2006.

[23] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 169 – 180. ACM, 1990.

[24] Fritz Henglein, Ken Friis Larsen, Jakob Grue Simonsen, and Christian Stefansen. Poets: Process-oriented event-driven transaction systems. *Journal of Logic and Algebraic Programming (JLAP). Special Issue on Contract-Oriented Software*, 2009.

[25] Anders Starcke Henriksen, Tom Hvitved, and Andrzej Filinski. A Game-Theoretic Model for Distributed Programming by Contract. In *Workshop on Games, Business Processes, and Models of Interactions*, Lübeck, September 2009. Gesellschaft für Informatik. To appear.

[26] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969. ISSN 0001-0782.

[27] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, 1978. ISSN 0001-0782.

[28] Kohei Honda, Vasco T. Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *In ESOP'98, volume 1381 of LNCS*, pages 122–138. Springer-Verlag, 1998.

[29] Jozef Hooman. Extending Hoare Logic to Real-Time. *Formal Aspects of Computing*, 6:6–801, 1994.

[30] Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, August 2004. ISBN 052154310X.

[31] Tom Hvitved. Session Types and Deadlockfree Communication. Student project, 2007.

[32] Tom Hvitved. Contracts in Programming and Enterprise Systems. `http://www.diku.dk/hjemmesider/ansatte/hvitved/master/projectdescription.pdf`, 2009. Project description ("Specialekontrakt").

[33] Tom Hvitved. Architectural Analysis of Microsoft Dynamics NAV. Technical report, Department of Computer Science, University of Copenhagen, 2009.

[34] Rune Højsgaard. Valuation of financial contracts in CCML. Master's thesis, Department of Computer Science, University of Copenhagen, 2009.

[35] C. B. Jones. *Wanted: a compositional approach to concurrency*, pages 1–15. Springer-Verlag New York, Inc., New York, NY, USA, 2000.

[36] Dexter Kozen. Results on the propositional mu-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[37] Gary T. Leavens and Yoonsik Cheon. Design by Contract with JML. `ftp://ftp.cs.iastate.edu/pub/leavens/JML/jmldbc.pdf`, 2005.

[38] Ronald M. Lee. A Logic Model for Electronic Contracting. *Decision Support Systems*, 4(1):27–44, 1988. ISSN 0167-9236.

[39] Peter F. Linington, Zoran Milosevic, James B. Cole, Simon Gibson, Sachin Kulkarni, and Stephen W. Neal. A unified behavioural model and a contract language for extended enterprise. *Data Knowl. Eng.*, 51 (1):5–29, 2004.

[40] William E. McCarthy. The REA Accounting Model: A Generalized Framework for Accounting Systems in a Shared Data Environment. *The Accounting Review*, LVII(3):554–578, 1982.

[41] Paul McNamara. Deontic Logic (Stanford Encyclopedia of Philosophy). `http://plato.stanford.edu/entries/logic-deontic/`, 2006.

[42] Elliot Mendelson. *Introducing Game Theory and its Applications*. Chapman & Hall/CRC, 2004. ISBN 1584883006.

[43] Bertrand Meyer. Applying "Design by Contract". *Computer*, 25(10):40–51, 1992. ISSN 0018-9162.

[44] R. Milner. *Communication and concurrency*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. ISBN 0-13-115007-3.

[45] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Information and Compututation*, 100(1):1–40, 1992. ISSN 0890-5401.

[46] Robing Milner. *Communicating and Mobile Systems: the Pi-Calculus*. Cambridge University Press, June 1999. ISBN 0521658691.

[47] Z. Milosevic, S. Gibson, P. F. Linington, J. Cole, and S. Kulkarni. On design and implementation of a contract monitoring facility. In Boualem Benatallah, Claude Godart, and Ming-Chien Shan, editors, *Proceedings of WEC, First IEEE International Workshop on Electronic*, pages 62–70. IEEE Computer Society, July 2004. ISBN 0-7695-2184-3.

[48] George C. Necula and Peter Lee. Safe Kernel Extensions Without Run-Time Checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996. USENIX.

[49] Morten Ib Nielsen. Logical Models for Value-Added Tax Legislation. Master's thesis, Department of Computer Science, University of Copenhagen, 2008.

[50] Michael Nissen. Contract Analysis. Technical report, Department of Computer Science, University of Copenhagen, 2007.

[51] Nir Oren, Sofia Panagiotidi, Javier Vázquez-Salceda, Sanjay Modgil, Michael Luck, and Simon Miles. Towards a Formalisation of Electronic Contracting Environments. In *Coordination, Organizations, Institutions and Norms in Agent Systems IV*, pages 156–171. Springer-Verlag, Berlin, Heidelberg, 2009. ISBN 978-3-642-00442-1.

[52] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[53] Vishal Patel. The Contract Management Benchmark Report: Procurement Contracts. Technical report, Aberdeen Group, 2006.

[54] Vishal Patel and Christopher J. Dwyer. Contract Lifecycle Management and the CFO: Optimizing Revenues and Capturing Savings. Technical report, Aberdeen Group, 2007.

[55] Simon Peyton-Jones and Jean-Marc Eber. How to write a financial contract. In Jeremy Gibbons and Oege de Moor, editors, *The Fun of Programming*. Cambridge University Press, New York, NY, USA, 2003.

[56] Henry Prakken and Marek Sergot. Contrary-to-duty obligations. *Studia Logica*, 57:91–115, 1996.

[57] Cristian Prisacariu and Gerardo Schneider. A Formal Language for Electronic Contracts . In *Formal Methods for Open Object-Based Distributed Systems*, pages 174–189. Springer-Verlag, Berlin, Heidelberg, 2007. ISBN 978-3-540-72919-8.

[58] Cristian Prisacariu and Gerardo Schneider. Towards Model-Checking Contracts. NWPT'07/FLACOS'07 Workshop Proceedings, Oslo, Norway, October 9–10 2007. Extended Abstract.

[59] Sriram K. Rajamani and Jakob Rehof. A behavioral module system for the pi-calculus. In *SAS '01: Proceedings of the 8th International Symposium on Static Analysis*, pages 375–394, London, UK, 2001. Springer-Verlag. ISBN 3-540-42314-1.

[60] RuleML. The Rule Markup Initiative. `http://www.ruleml.org`, 2009.

[61] Beat F. Schmid and Markus A. Lindemann. Elements of a reference model for electronic markets. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 4*, page 193, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 1060-3425.

[62] John R. Searle. *Speech acts: an essay in the philosophy of language.* Cambridge University Press, London, 1969.

[63] Perdita Stevens and Rob Pooley. *Using UML: Software Engineering with Objects and Components.* Addison-Wesley, 2000. ISBN 0201648601. Updated edition for UML1.4: first published 1998 (as Pooley and Stevens).

[64] Yao-Hua Tan and Walter Thoen. INCAS: a legal expert system for contract terms in electronic commerce. *Decision Support Systems*, 29 (4):389–411, 2000. ISSN 0167-9236.

[65] Yao-Hua Tan, Walter Thoen, and Somasundaram Ramanathan. A Survey of Electronic Contracting Related Developments. In *BLED 2001 Proceedings*, 2001.

[66] W3C. Web Services Choreography Description Language. `http://www.w3.org/TR/2005/CR-ws-cdl-10-20051109/`, 2005.

[67] Philip Wadler and Robert Bruce Findler. Well-Typed Programs Can't Be Blamed. In *ESOP*, pages 1–16, 2009.

[68] Glynn Winskel. *The formal semantics of programming languages: an introduction.* MIT Press, Cambridge, MA, USA, 1993. ISBN 0-262-23169-7.

[69] J. Woleński. Deontic Logic and Possible Worlds Semantics: A Historical Sketch. *Studia Logica*, 49(2):273–282, 1990.