ANALYSIS OF DISTRIBUTED SYSTEMS USING CONSTRAINED EXPRESSIONS

Laura K. Dillon

COINS Technical Report 84-18
September 1984

Computer and Information Science Department
University of Massachusetts, Amherst
Amherst, Massachusetts 01003

This work was supported in part by the International Business Machines Corporation under the Graduate Fellowship Program.

This research was funded in part by the International Business Machines Corporation under the Graduate Fellowship Program.

ANALYSIS OF DISTRIBUTED SYSTEMS USING CONSTRAINED EXPRESSIONS

A Dissertation Presented

by

Laura K. Dillon

Approved as to style and content by:

Jack C. Wileden, Chair of Committee

Lori A. Clarke, Member

George S. Avrunin, Outside Member

Edward M. Riseman, Department Chair
Computer and Information Science

ACKNOWLEDGEMENTS

I wish to thank my advisors, Jack C. Wileden, George S. Avrunin, and Lori A. Clarke, for their guidance and support during the preparation of this dissertation. Their direction, encouragement and friendship have always been generously given and are greatly appreciated.

I am also indebted to my husband, Ace, for his patience, emotional support and editorial comments, and to our son, Toby, for "helping" me keep my work in perspective.

Finally, I would like to thank my parents, Christopher and Eleanor Dillon, for planting the seeds from which this dissertation eventually grew.

ABSTRACT

ANALYSIS OF DISTRIBUTED SYSTEMS USING CONSTRAINED EXPRESSIONS

September 1984

Laura K. Dillon

B.A., University of Michigan

M.A., University of Michigan

M.S., University of Massachusetts

Ph.D., University of Massachusetts

Directed by: Associate Professor Jack C. Wileden

Due to the complexity of their tasks, developers of distributed systems would greatly benefit from automated tools to aid in the analysis of the systems they create. Ideally, these tools should be integrated in an environment providing tools to support all the activities required for developing and maintaining large, distributed software systems. Tools supporting analysis based on the constrained expression framework would make an important contribution to such an environment. The constrained expression framework is especially well-suited for use with system designs. It thus provides a basis for the development of preimplementation analysis tools. These tools would complement tools based on other distributed system analysis techniques. Due to the generality of the constrained expression framework, furthermore, analysis methods based on constrained expressions could be extended to provide common analysis methods across a number of phases in the software development process. This could be a source of valuable commonality and integration in a software development environment.

In this dissertation we describe the constrained expression framework and show how it can be used to help developers of distributed software systems analyze the systems they create. We explain how constrained expressions represent the possible behaviors of a concurrent system, illustrating the generality of constrained expressions by using them with systems expressed in three fundamentally different notations. A theory for manipulating constrained expressions is presented. Based on this theory, a procedure for simplifying the constrained expression representations of SDYMOL systems is formulated. This procedure results in a "reduced" constrained expression, which facilitates subsequent analysis. Algorithms for further simplifying SDYMOL reduced constrained expressions are also described. An example is presented to illustrate the simplification techniques. This example is also used to demonstrate analysis techniques, based on the constrained expression framework, that can be used to establish specific behavioral properties of a system.

TABLE OF CONTENTS

LIST OF FIGURES

Notation

$A, B, S, T, \ldots$	alphabets
$a, b, c, \ldots$	event symbols
$u, v, w, x, \ldots$	event strings
$\alpha, \beta, \gamma, \ldots$	event expressions
$\widehat{A}, \widehat{B}, \widehat{S}, \widehat{T}, \ldots$	collections of alphabets
$\widehat{C}$	collections of event expressions
$\mathbf{D}, \mathbf{F}$	tuples
$\mathcal{RE}(A)$	regular expressions over A
$\mathcal{EE}(A)$	event expressions over A
$\mathcal{L}(\alpha)$	language of α
$\mathcal{P}(\alpha)$	prefixes of α
$\rho_S(\alpha)$	projection on S (of α)
$L\|_{\widehat{C}}$	strings of L that satisfy $\widehat{C}$

Event and regular expression operators

$\alpha\beta$	concatenation
$\alpha \vee \beta$	alternation
α^*	Kleene star
$\alpha \Delta \beta$	interleave or shuffle operator
$\alpha^\dagger$	dagger or concurrent closure
α^n	the concatenation of n copies of α
$(\alpha \vee \beta)^{n \oplus m}$	the concatenation of n copies of α and m copies of β in all possible orders

General constrained expressions

$\mathbf{D} = (\mathbf{F}, \epsilon)$	constrained expression
$\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$	constraining context
A	augmented alphabet
E	terminal alphabet
$\widehat{S}$	constraining alphabets
$\widehat{C}$	constraints
ϵ	system expression
$I\mathcal{L}(\mathbf{D})$	interpreted language of $\mathbf{D}$
$\mathcal{WL}(\mathbf{D})$	Wileden interpreted language of $\mathbf{D}$
$\mathcal{P}(\epsilon)\|_{\widehat{C}}$	constrained prefixes
$\mathcal{L}(\epsilon)\|_{\widehat{C}}$	constrained language
$\mathbf{D} \sim \mathbf{D}'$	constrained expression equivalence
$\mathbf{D} \approx \mathbf{D}'$	strong equivalence of constrained expressions

SDYMOL system components

Σ	SDYMOL system
Q	set of processes
L	set of links (outbound ports)
P	set of (inbound) ports
M	set of message types
@	the undefined message type
$L(q)$	set of links of process q
$P(q)$	set of ports of process q
$F(p)$	set of links connected to port p

$T(l)$	set of ports connected to link l

SDYMOL constrained expression representations

$\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q\right)$	system expression
ι	initial expression
π_q	process expression
$\kappa_1, \kappa_2, \ldots, \kappa_6$	constraints
$S_1, S_2, \ldots, S_6$	constraint alphabets

SDYMOL event symbols

$r(l,p,m)$	reception of message of type m from link l through port p
$s(l,m)$	transmission of message of type m to link l
$ch(l,p)$	channel connecting link l to port p
$d(q,m)$	assignment of message type m to buffer of process q
$u(q,m)$	use of buffer of process q in which the value is assumed to be of type m
$w(p)$	process waits indefinitely for a communication through port p
$stop(q)$	normal termination of process q
$ne(q)$	non-event symbol for process q
$ill(q)$	illegal use of buffer of process q

CHAPTER I

INTRODUCTION

A distributed system is inherently concurrent and asynchronous. Its behavior often critically depends on the possibly unpredictable timing of its components. The large numbers of subtle interactions that can take place among the components of even a moderately-sized distributed system make it extremely difficult to reason about properties of the system's behavior. Powerful techniques are needed, therefore, for rigorously analyzing the possible behaviors of distributed software systems to assure that they exhibit all and only the properties intended.

Because of the complexity of the arguments involved, developers of distributed systems would greatly benefit from automated tools to aid in the analysis of the systems they create. Ideally, these tools should be integrated in an environment providing tools to support all the activities required for developing and maintaining large, distributed software systems. Such an environment should provide a diverse array of tools. In particular, it should offer tools supporting a wide variety of analysis techniques, since different techniques have complementary strengths and weaknesses. Just as importantly, it should offer tools with strong preimplementation analysis capabilities. By providing feedback on system descriptions that arise early in the development process (i.e., requirements, specifications, and designs), such tools can help developers detect errors early in that process, when they are least costly to correct.

The constrained expression framework provides a foundation for arguments concerning the order and number of events that occur in the behaviors of a distributed system. Such arguments play in important role in the analysis of distributed systems, and they have been widely, if informally, applied (see, for example, [HABE72], [HOAR82], [HOLZ82], [LAMP79], [MISR81]). Behavioral proper-

ties such as mutual exclusion, deadlock, and starvation, which involve interactions among the parts of a distributed system, are most naturally analyzed in terms of the order and number of event occurrences. The power of this type of analysis is illustrated in [AVRU83b], where a subtle flaw in a published algorithm for achieving mutual exclusion in distributed systems is uncovered using such analysis. Although this undesirable behavioral property is by no means obvious, it is uncovered by a routine application of analysis techniques based on the order and number of events that can occur in a behavior of the system. The constrained expression framework facilitates arguments of this nature by providing both a general purpose descriptive scheme that focuses on the representation of events and their ordering, and analysis techniques that can be used to reason about the number and order of event occurrences.

Tools supporting analysis based on the constrained expression framework would make an important contribution to an environment supporting the development of distributed software systems. They would complement tools based on other distributed system analysis techniques. As explained below, the constrained expression framework is especially well-suited for use with system designs. It thus provides a basis for the development of preimplementation analysis tools. Due to the generality of the constrained expression framework, furthermore, tools based on constrained expressions could be extended to provide common analysis methods across a number of phases in the software development process, including specification and implementation. This could be a source of valuable commonality and integration in a software development environment.

§1. The constrained expression framework

The constrained expression framework has two components. The first is a general purpose descriptive scheme that focuses on the representation of events and

their ordering. The second is a set of analysis techniques, based on the descriptive scheme, for reasoning about the number and order of event occurrences. We briefly describe below the approach to description and analysis used in the constrained expression framework. A thorough discussion of this material is the subject of this dissertation.

The descriptive scheme associated with the constrained expression framework is event-based. It originates from a technique developed by Riddle, called *message transfer expressions* [RIDD72] (later generalized and called *event expressions* [RIDD76]), for representing the behaviors of statically-structured concurrent systems, and closely resembles the *path expression* notations introduced by Campbell and Habermann [CAMP74]. Wileden further developed and generalized event expressions, producing *constrained expressions* [WILE78], which he then used to describe the behaviors of dynamically-structured parallel systems.

In the constrained expression framework, a system's activity is viewed as consisting of events, which can be of arbitrary duration and complexity, depending upon the level of detail at which the system is being considered. These events are assumed to be atomic (i.e., indivisible) and to occur serially. As a result of these assumptions, each possible behavior for a system can be described as a sequence of events, and represented by a string over an alphabet of event symbols defined for a given system and viewpoint level. The set of strings representing all the possible system behaviors thus constitutes a language over the event alphabet. A constrained expression representation of a distributed system provides a closed-form representation of this language, in much the same way as a regular expression provides a closed-form representation of the language describing the behavior of a finite state machine.

An important property of the constrained expression framework is its extreme generality. Although constrained expressions were originally formulated for representing the behaviors of systems expressed in DYMOL, they can be much more

widely applied. Examples of effective procedures for deriving constrained expression representations for systems expressed in three quite different development languages are presented in chapters III and IV. Constrained expressions can thus be used with systems expressed in a wide variety of software development languages.

The constrained expression representation of a distributed system is based on an *interleaved* model of concurrency [SHAW79]. The underlying idea is that a string representing a possible behavior of a system, which consists of a number of components (e.g., processes) executing concurrently, is obtained by shuffling or interleaving strings representing the behaviors of its components. A fundamental assumption of this model, then, is that the events in a behavior of a distributed system occur serially, and thus are totally ordered. It is often argued, however, that a partial ordering of system events more appropriately describes the behaviors of a distributed system (e.g., [GREI77], [CHEN82]). While it is true that the representation of a particular behavior of a distributed system as a string of event symbols imposes a total ordering on system events, a closed-form representation for all the possible behaviors of a system does not. As explained in chapter II, the constrained expression representation of a system is naturally interpreted as determining a partial ordering of system events. Events whose order is not determined by the constrained expression representation of the system can be (implicitly) considered concurrent. An activity of a component of a system that is to be explicitly represented as proceeding simultaneously with the activities of other components, furthermore, is easily modeled using an event symbol signifying the initiation of the activity and an event symbol signifying its termination. Thus, by simply varying the granularity of events, strings of event symbols can describe simultaneity within a distributed system.

A constrained expression representation of a distributed system is well-suited for use in reasoning about the order and number of occurrences of events in behaviors of the system. Like a regular expression, it explicitly encodes the order and number

of occurrences of event symbols in strings that represent behaviors of the system. Due to the interleaving of asynchronous behaviors that is a fundamental part of concurrency, the number of possible event orderings that must be considered when analyzing the behavior of a distributed system can be enormous. The constrained expression representation of a distributed system focuses on the data required for such reasoning. It provides a closed-form description of the possible behaviors of the system. Reasoning based on the constrained expression representation of a system thus deals with large classes of behaviors at once, reducing the combinatorial problems inherent in such analysis.

The approach to analysis employed in the constrained expression framework is to determine whether a particular symbol, or pattern of symbols, appears in a string representing a behavior of a system. The symbols in question may correspond to some desirable property of the system, such as mutually exclusive utilization of some shared resource, or graceful degradation and continued operation following the failure of one or more system components. Alternatively, they might represent pathological behaviors such as deadlocks. This directed approach to analysis (i.e., focusing on a particular question of interest) avoids the exhaustive analysis that is characteristic of most other techniques.

Some general techniques supporting this approach to reasoning about constrained expressions are based on the algebraic methods developed in [AVRU83a,b]. These techniques begin with the assumption that a certain pattern of events occurs in a string representing a behavior of the system. They then use the constrained expression to iteratively generate inequalities involving the numbers of occurrences of particular symbols appearing in various segments of the hypothesized string. If the assumption leads, at any stage of the iterative process, to an inconsistent system of inequalities, a contradiction has been reached. The assumption is thus incorrect and the given pattern does not occur in a behavior. Otherwise, inequalities are generated until enough information is obtained to construct a behavior contain-

ing the given pattern. Examples of this approach to reasoning about constrained expressions are presented in chapter VIII.

A number of techniques, described in chapters V–VIII, can be used to facilitate this analysis. One such technique involves "simplifying" the constrained expression representation of a distributed system. Constrained expressions are considered equivalent if they determine the same set of possible behaviors. Using this notion of constrained expression equivalence, a constrained expression representation for a distributed system can be formally manipulated and put into a normal form, much as an algebraic equality is simplified. The form of this "reduced" constrained expression facilitates arguments regarding the order and number of event occurrences. This simplification process is relatively straightforward, and could be easily automated.

Other techniques that facilitate the analysis of constrained expressions involve "factoring" the expressions in ways that allow the analyses of individual "factors" to be combined to give results about the behavior of the composite system. Under certain conditions, for example, behavioral properties of a distributed system can be established by analyzing the constrained expression representations of subsystems of the system. Such techniques allow modularization of the analysis and should help reduce combinatorial problems in the analysis of large systems.

The constrained expression framework thus provides a foundation for tools to aid developers of distributed systems in analyzing the systems they create. Such tools would automatically generate and simplify a constrained expression representation for a system from a description of the system expressed in a suitable development language. They would then be applied in a directed manner, using this intermediate representation of the system and the techniques described in this dissertation, to establish specific behavioral properties of the system. To reduce problems of a combinatorial nature, the search for a proof of (or counterexample to) some property could be directed by the system developer. The use of constrained

expressions, however, would remain completely transparent to the user. Software developers would thus be able to use these tools while continuing to work with the languages most appropriate to their tasks.

§2. Related work

Numerous languages have been proposed for describing concurrent systems during different phases of the software development process. These include, for example, the programming languages, Ada [DOD83], CSP [HOAR78], Distributed Processes [BRIN78], and PLITS [FELD79], the design languages, DREAM [RIDD78, WILE79] and DPMS [WILE80], and the specification language, COSY [LAUE79]. As indicated by the disparities in these languages, there is little agreement as to what should be included in them. Existing languages differ in focus, in the types of objects they use to represent aspects of a distributed system's design, and in the assumptions they make about general features of distributed systems, such as the nature of communication between various parts of the system. Such fundamental issues as what types of interprocess communication and synchronization primitives should be provided in a development language have yet to be resolved. Indeed, it may well be that different design languages, based on different primitives and different underlying models of concurrent computation, may be best suited for use in different types of distributed systems and in different phases of the development process. Nevertheless, developers of distributed systems desperately need assistance in analyzing the behaviors of the systems they create, whatever languages they choose.

Because of its generality, the constrained expression framework can be used with systems expressed in a wide variety of languages. The semantics of a particular language are captured by the procedure for deriving constrained expression representations for systems expressed in the language. Tools to analyze the sys-

tem would extract information about events and their orderings directly from this intermediate representation of the system. They would not rely on any assumptions, therefore, regarding general features of distributed systems, and could be applied to any system once a constrained expression representation for the system was obtained, regardless of the language used for describing the system.

A number of techniques for analyzing distributed systems have been investigated. By axiomatizing schemes for describing systems of communicating processes (essentially variants of CSP), Hoare [HOAR81a–c] has developed a calculus for proving a process satisfies an assertion describing its intended behavior. Misra and Chandy [MISR81] have extended the Floyd-Hoare proof technique for sequential programming [HOAR69] to prove invariant and terminal properties of networks of sequential communicating processes. In Milner's calculus for expressing synchronous communicating systems (SCCS) [MILN82], rules of the calculus indicate how expressions can be manipulated to prove properties of the systems they describe. Taylor's static analysis algorithm [TAYL83] produces a tree containing the reachable "concurrency states" of a system of communicating processes in order to ascertain all possible rendezvous, actions that can occur in parallel, and infinite wait situations. Using a finite state model of a distributed system, finite state testing [STAV83] determines the possible states that a system can reach and the sequences of events that can occur during execution. Holzmann [HOLZ82] has developed an algebra for validating communication protocols in message passing systems. All of the above techniques are primarily concerned with demonstrating safety properties. Temporal logic [PNUE79] has been used to prove both liveness and safety properties of concurrent systems [OWIC80,RAMA81].

These analysis techniques, however, suffer from a number of shortcomings. Taylor's static analysis algorithm is guaranteed to detect all pertinent error phenomena, but because all program paths are assumed to be executable, also generates superfluous reports. While it can be effectively used to assure that specific phenom-

ena do not occur, additional techniques are required to determine which reports are spurious and which correspond to actual errors in the software.

Both static analysis and finite state testing generate potential behaviors in an exhaustive, rather than a directed, manner. This creates two problems for a software developer concerned with a specific behavioral property. First, these techniques generally report vast numbers of potential behaviors when applied to realistically large and complex designs. The developer is then left with the task of sorting through and interpreting the reported behaviors to determine which, if any, are relevant to the property under investigation. Second, the exhaustive nature of static analysis and finite state testing makes them particularly susceptible to the combinatorial problems inherent in analyzing distributed software systems. The value of tools based on these techniques would be significantly enhanced if they were supplemented by tools supporting analysis based on the constrained expression framework. Such tools could be used, in a directed fashion, to seek out problems that tools based on static analysis and finite state testing are not able to find and to determine if reported potential behaviors represent behaviors that could actually occur.

Holzmann's algebra uses a type of extended regular expression to describe the behaviors of processes in a message passing system. These expressions are "multiplied" and manipulated algebraically to validate intended communication protocols. The rules of this algebra rely on specific assumptions regarding the nature of communication (e.g., FIFO message queueing) that limit the applicability of the approach.

The remaining analysis techniques, while more general than Holzman's algebra and theoretically more powerful than static analysis and finite state testing, are of little practical value to the typical system developer. Verification continues to suffer from the lack of sufficiently powerful theorem proving technology and the impenetrability and limited expressive power of formal specifications. Of course,

both of these shortcomings may eventually be overcome. Recent work on theorem provers (e.g., [BOYE79]), human-engineered specification languages based on formal logic (e.g., [LANS83], [RAMA80]), and more expressive logic formulations (e.g., [SCHW83], [KOYM83]) provides encouraging indications. Another difficulty with verification is that it crucially depends upon the correctness of the specifications on which the verification proofs are based. Unfortunately, these specifications are usually stated in formalisms chosen for their rich mathematical structure, rather than ease of use or clarity of expression. The typical system developer would find these formalisms cumbersome, at best, and the associated specifications extremely obtuse. Tools supporting analysis based on the constrained expression framework could profitably be used to aid developers in determining that their specifications described exactly the desired properties for the system being designed, and thus enhance the value of verification-based analysis tools.

Techniques for analyzing distributed systems based on some particular formal model of distributed computation, such as temporal logic (e.g., [RAMA80]), Petri nets (e.g., [KELL76]), COSY [LAUE79], CSP [HOAR81a–c], or the CCS [MILN82] formalism (e.g., [DOEP83], [KANE83]), are generally equivalent to verification in their power and complexity, and so suffer from essentially the same shortcomings. As in the case of verification-based methods, tools supporting analysis based on the constrained expression framework would be compatible with approaches based on formalisms such as these and would add important analysis capabilities.

§3. Outline of the dissertation

In this dissertation we describe the constrained expression framework and the results of research on its representational power and analytic capability. Chapter II explains how constrained expressions are used to represent the possible behaviors of distributed systems. It defines constrained expressions and shows how they

are associated with languages. It then compares this formulation of constrained expressions with Wileden's original formulation of constrained expressions, in order to put the constrained expression framework, as presented in this dissertation, in the proper historical perspective.

Chapters III and IV explain how constrained expressions are used to represent the behaviors of systems expressed in appropriate distributed system development languages. Chapter III describes a particular distributed system design notation, called SDYMOL, and a procedure for generating constrained expression representations for SDYMOL systems. Procedures for generating constrained expression representations for systems described using two quite different notations, Petri net languages and CSP, are then sketched in chapter IV to illustrate the representational generality of the constrained expression framework.

Chapters V–VIII show how constrained expressions are manipulated and analyzed to determine behavioral properties of the systems they describe. Chapter V develops the theory required in later chapters. In particular, it introduces an appropriate notion of constrained expression equivalence, along with results for simplifying constrained expressions and for modularizing their analysis. To reasonably limit the scope of the research, the remaining chapters focus primarily on the manipulation and analysis of SDYMOL designs. Chapters VI and VII show how the constrained expressions derived from SDYMOL designs are simplified. The first of these, chapter VI, describes a procedure that puts the constrained expression derived from the SDYMOL design of a system into a normal form, which we call *reduced*, to facilitate the eventual analysis of the system. (The more technical details of a proof of correctness of this procedure are presented in the appendix.) The second, chapter VII, describes two algorithms that are used to further simplify a reduced constrained expression representation of an SDYMOL system. Techniques for analyzing the reduced constrained expression representation of an SDYMOL system are then presented in chapter VIII. These techniques are based on the algebraic tech-

niques outlined above and the results for modularizing the analysis of constrained expressions described in chapter V.

Finally, we summarize the dissertation and the conclusions that can be drawn from it, and suggest extensions and directions for future research in chapter IX.

CHAPTER II

CONSTRAINED EXPRESSION REPRESENTATIONS

Constrained expressions were originally developed by Wileden [WILE78] to describe the possible behaviors of dynamically-structured parallel systems expressed in DPMS. In [DILL83], Wileden's technique for describing the behaviors of DPMS systems was reformulated to facilitate the description of distributed systems expressed in different modeling schemes, as well as the description of certain pathological behaviors, such as deadlocks, or behaviors in which processes starve or system components fail. In this chapter, we explain precisely what constrained expressions are and how they are used to represent the potential behaviors of distributed systems. We also compare the formulation of constrained expressions presented here with Wileden's original formulation and show that the two are essentially equivalent.

A brief overview of the descriptive component of the constrained expression framework is presented first. The notation and theory required for a more formal treatment of this material is then introduced, after which the descriptive notation associated with the constrained expression framework is formally defined. Finally, the original formulation of constrained expressions is described and shown to be equivalent to the revised formulation presented here.

§1. Describing behavior using constrained expressions

The descriptive notation associated with the constrained expression framework is event-based, and so provides a foundation for analysis tools supporting arguments regarding order and number of event occurrences. As explained in the introduction, a system's activity is viewed as consisting of events, where the events can be of arbitrary duration and complexity, depending upon the level of detail

at which the system is being considered. The events, however, are assumed to be atomic (i.e., indivisible) and to occur serially. Each possible behavior for a system can thus be described as a sequence of events, which can be represented by a string over an alphabet of event symbols defined for the given system and viewpoint level. The set of legal *behavioral traces* of a system, i.e. of strings representing legitimate system behaviors, constitutes a language over the event alphabet, and the constrained expression representation of a system provides a closed-form representation of this language, in much the same way as a regular expression provides a closed-form representation of the regular language describing the behavior of a finite state machine. Because the language of strings representing a distributed system's behaviors is much more complicated than a regular language, the closed form representation is more complicated than a regular expression.

This representation of a distributed system's behaviors is based on an interleaved model of concurrency, in which every legal behavioral trace of a system consisting of a number of components (e.g., processes) executing concurrently is assumed to be obtained by suitably interleaving or shuffling behavioral traces of its components. The representation of a particular behavior of a distributed system as a string of event symbols, of course, imposes a total ordering on system events. The closed-form representation for all the possible behaviors of a distributed system, however, does not. As explained below, the constrained expression representation of a system is naturally interpreted as determining a partial ordering of system events. Events in a behavior of a system that are not ordered by the constrained expression representation of the system are thus (implicitly) considered concurrent. An activity of a component that is to be explicitly represented as proceeding simultaneously with activities of other components, furthermore, is easily modeled using an event signifying the initiation of the activity and an event signifying its termination. Thus, by varying the granularity of events, a behavioral trace can describe simultaneity within a distributed system.

To simplify the discussion, we assume that a constrained expression is derived from a design for a distributed system. In general, however, it could be derived from some other formal description of a distributed software system (e.g., a formal specification or an implementation).

The constrained expression representation of a distributed system consists of a *system expression* and a collection of *constraints*. The system expression is a regular expression over an alphabet of symbols representing events in the system. It can be derived from a formal design of the system through the use of a set of translation rules. The translation rules express part of the semantics of the design language. Every legal behavioral trace of the system is a prefix of a string from the regular language associated with the system expression. (We use prefixes, rather than complete strings, in the language of the system expression to facilitate the representation of behaviors in which parts of the system terminate abnormally.) Because the translation rules are not able to efficiently express that part of the semantics involving interaction between processes, however, some of the prefixes represent event sequences that cannot be behaviors. The prefixes of strings from the language of the system expression, therefore, can be viewed as "candidate" behavioral traces of the system. The constraints of the constrained expression representation are used to eliminate those that do not represent legitimate system behaviors.

The constraints are also expressions over the alphabet of event symbols. They are formed using the regular expression operators and one additional operator, denoted by $\dagger$. (A formal description is given below.) These expressions define legal patterns of event symbols and embody that part of the semantics of the design language not expressed by the translation rules. Essentially the same set of constraints is used with every constrained expression derived from a system expressed in a given design language. The constraints are used to "filter" the prefixes of strings from the language of the system expression to eliminate those prefixes that do not corre-

spond to possible system behaviors. Finally, event symbols introduced for technical reasons, but representing events that are not of interest when describing the final system behaviors, are erased from the remaining prefixes. The resulting language, called the *interpreted language* of the constrained expression, represents exactly the possible behaviors of the system. This construction is described more formally below.

§2. Preliminary concepts and notation

This section presents the theoretical background required for a formal definition of constrained expressions and also for the analysis of their interpreted languages, which is described later. A general familiarity with basic concepts from the theory of formal languages is assumed. We review relevant concepts and results from this theory here, explain our terminology, and establish some preliminary notation.

As mentioned above, every sequence of system events corresponds to a string of symbols over an alphabet representing these events. To describe specific subsequences of a sequence of system events, we introduce the following terminology.

(2.1) *Definition.* Given a string, u, of symbols over an alphabet, A, a string z is a *substring* of u if there are strings v and w such that $u = vzw$. If v is the null string then z is also called a *prefix* of u, and if w is the null string then z is also called a *tail* of u.

We use regular expressions to represent regular languages in the usual fashion. To facilitate the representation of concurrency, however, we add an additional operator to the traditional regular expression operators.

(2.2) *Notation.* $\mathcal{RE}(A)$ denotes the set of regular expressions over the alphabet A, where regular expressions over an alphabet are formed from the symbols in the alphabet, the null string (represented by λ) and the empty set (represented by $\emptyset$), and from finite applications of the usual regular expression operators, alternation (represented by $\vee$), concatenation (represented by juxtaposition), and transitive closure (represented by *), and of an additional operator, the shuffle or interleave operator (represented by Δ). (The relative precedence of the regular expression operators, from highest to lowest, is as follows: *, juxtaposition, Δ, and finally, $\vee$.)

The shuffle operator, a binary operator which signifies the shuffling or interleaving of the symbols from its arguments, has been shown to preserve regularity [GINS66] and is useful for representing concurrent activity. The regular expression $ab \Delta cd$, for example, represents the set $\{abcd, acbd, acdb, cabd, cadb, cdab\}$. If ab and cd represent the behaviors of two processes that do not interact, clearly $ab \Delta cd$ represents the possible behaviors of the two processes executing concurrently.

A broader class of languages is required to express certain constraints on the order and number of symbols that appear in legal behavioral traces of a system. To represent the languages in this class, we define the following set of expressions.

(2.3) *Definition.* The *event expressions* over an alphabet, A, are formed from the symbols in A, the null string and the empty set, and from finite applications of the regular expression operators (including Δ), along with the unary concurrent closure or dagger operator (represented by †), which signifies the shuffle of zero or more copies of its argument. (The † operator is at the same level of precedence as the * operator.) The set of event expressions over A is denoted by $\mathcal{EE}(A)$.

Event expressions were originally developed by Riddle to describe the message-passing behavior of PPML models [RIDD76]. The dagger operator greatly extends the descriptive power of event expressions, which can describe many non-regular languages. In the constrained expression representation of a system, it is typically

used for expressing constraints that assure processes are properly synchronized, as required by the semantics of the communication primitives.

Notations for referring to the language of an event expression or regular expression and to the language of prefixes of strings from the language of an event expression or regular expression are now introduced.

(2.4) *Notation.* If α is either an event expression or a regular expression, then $\mathcal{L}(\alpha)$ denotes the language of α and $\mathcal{P}(\alpha)$ denotes the language of prefixes of α.[1]

When analyzing the constrained expression representation of a system, we draw upon a wealth of theory regarding regular expressions and regular languages. For the most part, the results that we require are fairly elementary. We note a few of the less obvious ones here.

(2.5) **Proposition.** *If $\alpha \in \mathcal{RE}(A)$ does not involve the shuffle operator and if α' is obtained from α by replacing a subexpression, β, of α with the regular expression β', then*

(i) $u \in \mathcal{L}(\alpha) - \mathcal{L}(\alpha')$ implies that u contains a substring that belongs to $\mathcal{L}(\beta) - \mathcal{L}(\beta')$, and

(ii) $u \in \mathcal{P}(\alpha) - \mathcal{P}(\alpha')$ implies that u contains a substring that belongs to $\mathcal{L}(\beta) - \mathcal{L}(\beta')$ or a tail that belongs to $\mathcal{P}(\beta) - \mathcal{P}(\beta')$.

If the regular expression α in this proposition involves the shuffle operator, the results (i) and (ii) do not necessarily hold. Consider, for example, the regular expression $(ab \vee c) \Delta d$ and the subexpression ab. Replacing ab with the empty regular expression, we obtain the regular expression $(\emptyset \vee c) \Delta d = c \Delta d$. The string adb is then a counterexample to both (i) and (ii). It is a string (prefix)

[1] By a prefix of an expression, we obviously mean a prefix of a string from the language associated with the expression. We resort to similar abuses of terminology and notation when appropriate to avoid more cumbersome terminology (as above) and a proliferation of notational conventions (as in (2.9), where f is used to denote both a homomorphism on strings and a map on event symbols).

that is eliminated when ab is replaced by $\emptyset$, but it does not contain a substring that belongs to $\mathcal{L}(ab)$ (non-null tail that belongs to $\mathcal{P}(ab)$). It does, however, contain a non-null string that belongs to $\mathcal{L}(ab)$ ($\mathcal{P}(ab)$) interleaved within it, and in general, we have the following proposition.

(2.6) **Proposition.** *If α' is obtained from $\alpha \in \mathcal{RE}(A)$ by replacing a subexpression β of α with a regular expression β', then*

(i) $u \in \mathcal{L}(\alpha) - \mathcal{L}(\alpha')$ *implies that* $u \in \mathcal{L}(w \Delta v)$, *for some* $v, w \in A^*$ *such that* $v \in \mathcal{L}(\beta) - \mathcal{L}(\beta')$

(ii) $u \in \mathcal{P}(\alpha) - \mathcal{P}(\alpha')$ *implies that* $u \in \mathcal{L}(w \Delta v)$, *for some* $v, w \in A^*$ *such that* $v \in \mathcal{P}(\beta) - \mathcal{P}(\beta')$ *or* $v \in \mathcal{L}(\beta) - \mathcal{L}(\beta')$.

Certain maps, called *projections*, are used in defining the interpreted language of a constrained expression. These maps are more generally characterized as *homomorphisms* of the languages involved, as they distribute over concatenation. The next definition formalizes the notion of a homomorphism between languages.

(2.7) *Definition.* Given alphabets, A_1 and A_2, and languages over these alphabets, $L_1 \subseteq A_1^*$ and $L_2 \subseteq A_2^*$, a map, $f : L_1 \to A_2^*$, is a *homomorphism* of L_1 to L_2 if

(i) $f(L_1) \subseteq L_2$ and

(ii) $f(uv) = f(u)\,f(v)$, for every $u, v \in L_1$ such that $uv \in L_1$.

The following remarks pertain to homomorphisms in general, and so are also relevant when considering projections.

(2.8) *Remark.* Every map on symbols, $f : A_1 \to A_2^*$, extends uniquely to a homomorphism on strings, $f : A_1^* \to A_2^*$. This homomorphism is defined inductively by $f(\lambda) = \lambda$ and $f(aw) = f(a)\,f(w)$ for $a \in A_1$ and $w \in A_1^*$. Conversely, every homomorphism of A_1^* to A_2^* is obtained by extending the appropriate map from A_1 to A_2^*. In practice it is often convenient to define a homomorphism by specifying its behavior on symbols.

(2.9) *Remark.* A homomorphism, $f : A_1^* \to A_2^*$, induces a map on event (regular) expressions, $f : \mathcal{EE}(A_1) \to \mathcal{EE}(A_2)$ ($f : \mathcal{RE}(A_1) \to \mathcal{RE}(A_2)$), where for each event (regular) expression α, the expression $f(\alpha)$ is obtained by replacing each string from A_1^* in α by its image under f.

If α does not involve the shuffle or dagger operators and if f is a homomorphism, it is easy to see that $\mathcal{L}\Big(f(\alpha)\Big) = f\Big(\mathcal{L}(\alpha)\Big)$. In general, however, it can only be observed that $f\Big(\mathcal{L}(\alpha)\Big) \subseteq \mathcal{L}\Big(f(\alpha)\Big)$. If $f(a) = cd$, for example, $f\Big(\mathcal{L}(a \,\Delta\, a)\Big) = \{cdcd\}$ and $\mathcal{L}\Big(f(a \,\Delta\, a)\Big) = \{cdcd, ccdd\}$. In the special case where $f(A_1) \subseteq A_2 \cup \{\lambda\}$, however, $\mathcal{L}\Big(f(\alpha)\Big) = f\Big(\mathcal{L}(\alpha)\Big)$, for every $\alpha \in \mathcal{RE}(A_1)$ and also for every $\alpha \in \mathcal{EE}(A_1)$.

The projection maps, which are needed for constructing the interpreted language of a constrained expression, are now defined as follows.

(2.10) *Definition.* For each subset S of an alphabet A, we define a homomorphism, $\rho_S : A^* \to S^*$, called *projection* on S, by extending the map,

$$\rho_S(a) = \begin{cases} a & \text{if } a \in S; \\ \lambda & \text{otherwise,} \end{cases}$$

from a map on symbols to a homomorphism on strings.

The next observation, which is used repeatedly in the remainder of the discussion, follows from (2.9) above.

(2.11) $\qquad \mathcal{L}\Big(\rho_S(\alpha)\Big) = \rho_S\Big(\mathcal{L}(\alpha)\Big) \qquad$ and $\qquad \mathcal{P}\Big(\rho_S(\alpha)\Big) = \rho_S\Big(\mathcal{P}(\alpha)\Big),$

for every $S \subseteq A$, and $\alpha \in \mathcal{RE}(A)$ or $\alpha \in \mathcal{EE}(A)$. Thus, the projection maps respect the language structure of both event expressions and regular expressions.

§3. The descriptive component of the constrained expression framework

We now turn our attention to constrained expressions.

(3.1) *Definition.* A ***constrained expression*** is an ordered pair, $\mathbf{D} = (\mathbf{F}, \epsilon)$, where

(i) $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$ is a quadruple, called the ***constraining context***, with

(a) A an alphabet of event symbols, called the ***augmented alphabet***,

(b) $E \subseteq A$, called the ***terminal alphabet***,

(c) $\widehat{S} = \{ S_j \}_{j \in J}$, where J is a set of indices and, for each $j \in J$, $S_j \subseteq A$ is called a ***constraint alphabet***, and

(d) $\widehat{C} = \{ \kappa_j \}_{j \in J}$, where, for each $j \in J$, κ_j is an event expression, called a ***constraint***, over the constraint alphabet S_j, and

(ii) ϵ is a regular expression, called the ***system expression***, defined over the augmented alphabet A.

A constrained expression representing the possible behaviors of a system can be generated from any sufficiently formal design of the system. The system expression is derived from the design through the application of a set of translation rules,[1] so that the prefixes of the system expression represent event sequences which include all possible system behaviors. Because it is difficult to express certain semantics of a distributed system design language (the semantics of interprocess communication, for example) using translation rules, some of the prefixes of the system expression may represent event sequences which do not correspond to legitimate system behaviors. The constraints are used to eliminate those prefixes that are not legal behavioral traces. (We describe this process more formally below.) The terminal alphabet, which is determined by the problem being studied, identifies the symbols that represent events of interest and are to be retained in the interpreted language. Additional symbols are often needed to express certain semantic constraints. Together, these and the "interesting" symbols comprise the augmented alphabet.

By describing how symbols in the respective constraint alphabets appear in

[1] The particular set of translation rules used in deriving the system expression is determined by the design notation. The content of this set, and hence the precise procedure by which the system expression is obtained, is irrelevant to the ensuing discussion. Examples of different tranlation rules are presented in chapters III and IV.

legitimate system behaviors, the constraints embody those semantics of the design language that can be violated by prefixes of the system expression. The constraints are tailored to the particular system being described, but are essentially determined by the design language, and can be generated from the design of the system in a purely mechanical fashion. They constrain the prefixes of the system expression that survive the "filtering" step in the following sense: prefixes in which the symbols from the constraint alphabets appear as described by the various constraints are retained, and all other prefixes are eliminated. This notion is formalized in the next definition.

(3.2) *Definition.* A string $w \in A^*$ is a ***constrained prefix*** of the constrained expression $\mathbf{D}$, where $\mathbf{D}$ is as in (3.1), if

(i) $w \in \mathcal{P}(\epsilon)$, and

(ii) $\rho_{S_j}(w) \in \mathcal{L}(\kappa_j)$, for all $j \in J$.

A string $w \in A^*$ is said to *satisfy* the constraint κ_j if $\rho_{S_j}(w) \in \mathcal{L}(\kappa_j)$. Given a language $L \subseteq A^*$ and a set of constraints $\hat{B} \subseteq \hat{C}$, the subset of L satisfying all the constraints in $\hat{B}$ is written $L|_{\hat{B}}$. The set of constrained prefixes of $\mathbf{D}$ is thus written $\mathcal{P}(\epsilon)|_{\hat{C}}$. In the special case where $\hat{B} = \{\kappa_j\}$ is a singleton set, we write $L|_{\kappa_j}$ for $L|_{\hat{B}}$.

The constrained prefixes of a constrained expression are thus those prefixes from the language of the system expression that satisfy all the constraints, and so correspond to possible system behaviors.

Although every prefix of the system expression in the constrained expression representation of a system is, potentially, a behavioral trace of the system, analysis of the possible system behaviors is facilitated if every legal behavioral trace is represented by a full string from the language of the system expression. When this is the case, we need only consider full strings from the language of the system expression satisfying the constraints. We therefore make the following definition.

(3.3) *Definition.* The ***constrained language*** of the constrained expression $\mathbf{D}$, where $\mathbf{D}$ is defined in (3.1), consists of those strings from the language of its system expression that satisfy all the constraints. It is written $\mathcal{L}(\epsilon)|_{\widehat{C}}$.

The constrained expression $\mathbf{D}$ is said to be ***reduced*** if its constrained prefixes are all contained in its constrained language, i.e., if $\mathcal{L}(\epsilon)|_{\widehat{C}} = P(\epsilon)|_{\widehat{C}}$.

Certain techniques for analyzing constrained expressions can only be applied to reduced constrained expressions, and even techniques which can be used more generally are usually easier to apply to constrained expressions that are reduced. As shown in the next section, by simply replacing the system expression of a constrained expression with an appropriate regular expression, a reduced constrained expression with the same set of constrained prefixes as the original constrained expression can always be obtained. A more complex system expression is usually required, however, when passing to a reduced constrained expression in this manner. In chapter VI we show how the constraints in a constrained expression can be used to ameliorate this problem.

The interpreted language of a constrained expression is finally defined as follows.

(3.4) *Definition.* The ***interpreted language***, $I\mathcal{L}(\mathbf{D})$, of the constrained expression $\mathbf{D}$, where $\mathbf{D} = (\mathbf{F}, \epsilon)$ and $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, is obtained by projecting the constrained prefixes of $\mathbf{D}$ on the terminal alphabet E. Thus, $I\mathcal{L}(\mathbf{D}) = \rho_E\left(P(\epsilon)|_{\widehat{C}}\right)$.

Of course, if $\mathbf{D}$ is reduced, then its interpreted language is given by $I\mathcal{L}(\mathbf{D}) = \rho_E\left(\mathcal{L}(\epsilon)|_{\widehat{C}}\right)$.

We project the language of constrained prefixes on a terminal alphabet when forming the interpreted language because, in general, certain symbols in the augmented alphabet of a constrained expression are used for expressing semantic constraints and do not represent actual system events. Moreover, a designer may find it easier to interpret the behavioral traces of a system when a restricted subset of the

event alphabet is used. For different applications, in fact, it may be appropriate to use different terminal alphabets in the constrained expression representations of the same system. For this reason, any subset of the augmented alphabet may compose the terminal alphabet of a constrained expression.

Although we are ultimately interested in the interpreted language of a constrained expression, we must consider its language of constrained prefixes to obtain general results for manipulating and analyzing the constrained expression. This is because the terminal alphabet and augmented alphabet of the constrained expression may coincide, and when this happens the interpreted language and the language of constrained prefixes are the same. When proving properties of constrained expressions, therefore, we usually reason about their languages of constrained prefixes (or, if the constrained expressions are reduced, about their constrained languages), rather than the corresponding interpreted languages.

A constrained expression can be viewed as determining a partial ordering of the event symbols in a string from its interpreted language (or from its set of constrained prefixes). The order of appearance of certain symbols relative to the appearance of certain other symbols in a string from the interpreted language of a constrained expression is usually determined by the forms of the system expression and various constraints. This implied ordering of event symbols in a string from the interpreted language of a constrained expression, however, is not, in general, total, since the relative order of the appearance of some of the symbols may not be determined by the form of the system expression (e.g., if the system expression contains Δ operators) or by any constraints (e.g., if the symbols do not belong to a constraint alphabet and are not ordered with respect to symbols that are ordered by constraints). When the constrained expression represents the possible behaviors of a distributed system, this partial ordering of event symbols in a legal behavioral trace of the system determines a partial ordering of system events. Events that are not related by this partial ordering can be viewed as concurrent.

§4. Comparison with Wileden's original formulation of constrained expressions

Constrained expressions were originally devised by Wileden for describing the behavior of distributed systems modeled in DPMS. As indicated by the next definition, this formulation of constrained expressions was slightly more restrictive than the one presented above.

(4.1) *Definition.* A ***Wileden constrained expression*** is a constrained expression in which the terminal and constraint alphabets partition the augmented alphabet.

In a Wileden constrained expression, therefore, the terminal alphabet and the constraint alphabets are all pairwise disjoint and the augmented alphabet is simply the union of these alphabets.

The next definition describes how these expressions were originally used to obtain a language representing the possible behaviors of a system.

(4.2) *Definition.* Given a Wileden constrained expression $\mathbf{D} = (\mathbf{F}, \epsilon)$, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\widehat{S} = \{ S_j \}_{j \in J}$, and $\widehat{C} = \{ \kappa_j \}_{j \in J}$, the ***Wileden interpreted language*** of $\mathbf{D}$ is obtained as follows. First, each constraint κ_j, for $j \in J$, is used to form a ***Wileden constraining language***, C_j, consisting of the language of κ_j shuffled with the transitive closures of all the alphabets except S_j,

$$C_j = \mathcal{L}(\kappa_j) \Delta \left(\underset{i \neq j}{\Delta} S_i^* \right).$$

Then the language of the system expression is intersected with the Wileden constraining languages to form the ***Wileden interpreted language***,

$$L = \mathcal{L}(\epsilon) \cap \left(\bigcap_{j \in J} C_j \right).$$

Finally, the Wileden constrained language is projected on the terminal alphabet to produce the Wileden interpreted language of $\mathbf{D}$, $\mathcal{WL}(\mathbf{D}) = \rho_E(L)$.

There are ostensibly three differences between the original formulation of constrained expressions and the revised formulation presented here.

(4.3) In the original formulation of constrained expressions, the terminal alphabet and the constraint alphabets are required to be pairwise disjoint, whereas any of these alphabets may intersect nontrivially in the revised formulation.

(4.4) The filtering step in the original formulation of constrained expressions is defined in terms of the intersection of Wileden constraining languages rather than images of constraint alphabet projection maps, as in the revised formulation.

(4.5) Finally, in the original formulation of constrained expressions, it is the language of the system expression, not the language of prefixes of the system expression, as in the revised formulation, that is used in the filtering step to produce a set of strings representing the actual system behaviors.

In spite of these apparent differences, the two formulations of constrained expressions can be shown to be equivalent, in the sense that, given an arbitrary constrained expression of either type there is an effective procedure for producing a constrained expression of the other type such that the Wileden interpreted language of the Wileden constrained expression and the interpreted language of the more general constrained expression are the same. At a theoretical level, therefore, the revised formulation of constrained expressions is no more powerful than the original one. At a practical level, however, it has been found to facilitate the description and analysis of systems expressed in different modeling schemes, as explained below.

To show that the frameworks are equivalent we examine each of the above noted differences separately. As the last two, (4.4) and (4.5), are more easily dismissed than the first, (4.3), we present the discussion of (4.3) after the discussion of the other two.

We first consider, therefore, the manner in which the constraints are used for filtering out strings representing event sequences which violate the semantics of the formal modeling scheme. Recall that a string survives the filtering step in the

original formulation of constrained expressions if it belongs to each of the Wileden constraining languages, C_j, for $j \in J$. The following observation shows that this is actually equivalent to the condition for retaining a string in the revised formulation.

Because the constraint and terminal alphabets in a Wileden constrained expression are pairwise disjoint, it is easily seen that

$$u \in C_j \Leftrightarrow \rho_{S_j}(u) \in \mathcal{L}(\kappa_j), \tag{4.6}$$

where $u \in A^*$ and κ_j, S_j, and C_j are as defined in (4.2). Thus, a string belongs to every Wileden constraining language if and only if it satisfies all the constraints (in the sense of (3.2)), which is precisely the condition under which a prefix is retained in the revised formulation of constrained expressions. The appropriate strings, therefore, are actually filtered in exactly the same manner in both frameworks. In fact, the constrained language (defined in (3.3)) and the Wileden constrained language of a Wileden constrained expression are identical.

In the revised formulation of constrained expressions, the original filtering process is simply restated using constraint alphabet projection maps instead of Wileden constraining languages. We chose to restate the filtering step in this manner because it is more natural, when analyzing the interpreted language of a constrained expression, to think in terms of projection maps than Wileden constraining languages. In fact, the typical way to determine if a string belongs to a Wileden constraining language is to project the string on the corresponding constraint alphabet and determine if the projected string belongs to the language of the constraint. Moreover, the projected string usually contains fewer symbols than the original string, making it easier to reason about.

This restatement of the filtering step suggest the following generalization of definition (4.2), which we use throughout the remainder of the discussion.

(4.7) *Definition.* The *Wileden interpreted language*, $\mathcal{WL}(\mathbf{D})$, of the constrained expression $\mathbf{D}$, where $\mathbf{D} = (\mathbf{F}, \epsilon)$ and $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, is obtained by project-

ing the constrained language of $\mathbf{D}$ on the terminal alphabet. Thus, $\mathcal{WL}(\mathbf{D}) = \rho_E\left(\mathcal{L}(\epsilon)|_{\widehat{C}}\right)$.

As noted above, this definition agrees with definition (4.2) in the special case where the constrained expression is a Wileden constrained expression. Definition (4.7), however, applies to arbitrary constrained expressions, whereas definition (4.2) applies only to Wileden constrained expressions. Every constrained expression, therefore, has both an interpreted language and a Wileden interpreted language associated with it. In general, these two languages are not the same, the essential difference being that the interpreted language is formed using the prefixes of the system expression and the Wileden interpreted language is formed using the language of the system expression.

We now consider the affect that using the prefixes of the system expression, instead of full strings from its language, has on the languages that are obtained. We thus compare the class of Wileden interpreted languages, which are obtained by filtering the language of the system expression, to the class of interpreted languages, which are obtained by filtering the prefixes of the system expression. The next two propositions show that these two classes are in fact identical. The first demonstrates an effective procedure for generating a constrained expression whose Wileden interpreted language is the same as the interpreted language of a given constrained expression. Conversely, the second demonstrates an effective procedure for generating a constrained expression whose interpreted language is the same as the Wileden interpreted language of a given constrained expression.

(4.8) **Proposition.** *Let the constrained expression* $\mathbf{D}$ *be given by,* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\widehat{S} = \{S_j\}_{j\in J}$, *and* $\widehat{C} = \{\kappa_j\}_{j\in J}$. *If* $\mathbf{D}' = (\mathbf{F}, \epsilon')$, *where* $\epsilon' = P(\epsilon)$, *and* P *is the function on* $\mathcal{RE}(A)$ *defined inductively by,*

$$P(\emptyset) = \emptyset,$$
$$P(\lambda) = \lambda,$$
$$P(a) = a \vee \lambda,$$

$$P(\alpha \vee \beta) = P(\alpha) \vee P(\beta),$$
$$P(\alpha\beta) = P(\alpha) \vee \alpha P(\beta),$$
$$P(\alpha^*) = \alpha^* P(\alpha), \text{ and}$$
$$P(\alpha \triangle \beta) = P(\alpha) \triangle P(\beta),$$

for $a \in A$ *and* $\alpha, \beta \in \mathcal{RE}(A)$, *then* $\mathcal{IL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}')$.

Proof. First observe that if α is a regular expression, then $\mathcal{L}\left(P(\alpha)\right) = \mathcal{P}(\alpha)$, a fact that follows easily from the definition of P. Hence, $\mathcal{P}(\epsilon) = \mathcal{L}\left(P(\epsilon)\right) = \mathcal{L}(\epsilon')$, from which it follows trivially that the constrained prefixes of $\mathbf{D}$ and the constrained language of $\mathbf{D}'$ are identical, and so $\mathcal{IL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}')$. ∎

Any language obtained by filtering the prefixes of a system expression through a set of constraints, therefore, can also be obtained by filtering the language of a different system expression through the same set of constraints. The class of interpreted languages is thus no richer than the class of Wileden interpreted languages. In general, however, a more complex system expression is required if the language of the system expression is used instead of the prefixes of the system expression.

The next proposition shows that the class of Wileden interpreted languages is no richer than the class of interpreted languages.

(4.9) **Proposition.** *Let* $\mathbf{D}$ *denote the Wileden constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\widehat{S} = \{S_j\}_{j \in J}$, *and* $\widehat{C} = \{\kappa_j\}_{j \in J}$. *If* $\mathbf{D}' = (\mathbf{F}', \epsilon')$, *where* $\mathbf{F}' = (A', E', \widehat{S}', \widehat{C}')$, $\widehat{S}' = \{S'_j\}_{j \in J'}$, $\widehat{C}' = \{\kappa'_j\}_{j \in J'}$, *and* ϵ' *are as follows,*

(i) $A' = A \cup \{a'\}$, *for some distinguished symbol* a' *that is not contained in the alphabet* A,

(ii) $E' = E$,

(iii) $J' = J \cup \{j'\}$, *for some distinguished indexing symbol* j' *that is not in the indexing set* J,

(iv) $S'_j = S_j$, *for* $j \neq j'$, *and* $S'_{j'} = \{a'\}$,

(v) $\kappa'_j = \kappa_j$, *for* $j \neq j'$, *and* $\kappa'_{j'} = a'$, *and*

(iv) $\epsilon' = \epsilon a'$,

then $\mathcal{WL}(\mathbf{D}) = \mathcal{IL}(\mathbf{D}')$.

Proof. The desired equality is established by proving that each of the languages is contained in the other.

$\mathcal{WL}(\mathbf{D}) \subseteq \mathcal{IL}(D')$:

For $x \in \mathcal{WL}(\mathbf{D})$, choose $w \in \mathcal{L}(\epsilon)$ such that w satisfies the constraints of $\widehat{C}$ and $\rho_E(w) = x$, and define $w' = wa'$. We show that w' is a constrained prefix of $\mathbf{D}'$ and that $\rho_E(w) = \rho_E(w')$, from which the desired inclusion follows immediately.

Clearly, $w' \in \mathcal{L}(\epsilon') \subseteq \mathcal{P}(\epsilon')$, and since $a' \notin S'_j$ implies that $\rho_{S'_j}(w') = \rho_{S_j}(w)$, for $j \neq j'$, and w satisfies the constraints $\kappa_j = \kappa'_j$, for $j \neq j'$, the string w' satisfies the constraints κ'_j, for $j \in J$. But w' also satisfies the constraint $\kappa'_{j'}$, and so w' is a constrained prefix for $\mathbf{D}'$. Since $a' \notin E'$, furthermore, we also have $\rho_E(w) = \rho_{E'}(w')$, which establishes the desired result.

$\mathcal{IL}(\mathbf{D}') \subseteq \mathcal{WL}(\mathbf{D})$:

For $x' \in \mathcal{IL}(\mathbf{D}')$, choose $w' \in \mathcal{P}(\epsilon')$ such that w' satisfies the constraints of $\widehat{C}'$ and $\rho_{E'}(w') = x'$. Now $w' \in \mathcal{P}(\epsilon')$ implies that there is some $v' \in A'^*$ such that $w'v' \in \mathcal{L}(\epsilon')$. We first show that $w' \in \mathcal{L}(\epsilon')$ by demonstrating that if $v' \neq \lambda$, then w' does not satisfy the constraint $\kappa'_{j'}$. Suppose, therefore that $v' \neq \lambda$. Then $w'v' \in \mathcal{L}(\epsilon') = \mathcal{L}(\epsilon a')$ implies that there is some $u \in A^*$ such that $w'v' = w'ua'$ and $w'u \in \mathcal{L}(\epsilon)$. But then since $\epsilon \in \mathcal{RE}(A)$ and $a' \notin A$, we reason that $\rho_{S'_{j'}}(w'u) \in \rho_{S'_{j'}}\Big(\mathcal{L}(\epsilon)\Big) = \mathcal{L}\Big(\rho_{S'_{j'}}(\epsilon)\Big) = \mathcal{L}(\lambda) = \lambda$. Hence, $\rho_{S'_{j'}}(w') = \lambda$, and so w' does not satisfy $\kappa'_{j'}$. We therefore conclude that $w' \in \mathcal{L}(\epsilon')$.

We now produce a string $y \in \mathcal{L}(\epsilon)|_{\widehat{C}}$ satisfying $\rho_E(y) = \rho_{E'}(w')$, to complete the proof. Clearly, $w' \in \mathcal{L}(\epsilon') = \mathcal{L}(\epsilon a')$ implies that there is some $y \in \mathcal{L}(\epsilon)$ with $w' = ya'$. For $j \in J$, furthermore, $\rho_{S'_j}(w') = \rho_{S_j}(y)$, because $S_j = S'_j$ and $a' \notin S_j$. As w' satisfies the constraints $\kappa'_j = \kappa_j$, for $j \in J$, we therefore conclude that y satisfies the constraints κ_j, for $j \in J$, and so lies in the con-

strained language of **D**. Finally, $\rho_{E'}(w') = \rho_{E}(y)$ follows from the observation that $a' \notin E'$. ∎

We use prefixes in the revised formulation of constrained expressions to make it easier to represent behaviors in which processes starve waiting for communications that cannot be realized. As originally conceived, the constrained expression representation generated from a DPMS model did not describe such behaviors. Strings representing event sequences that would result in the starvation of one or more processes were eliminated by the DPMS constraints. These are precisely the event sequences, however, that one must detect in order to identify certain undesirable system behaviors, such as those in which processes cannot progress due to system deadlock. We would like to be able to ascertain the conditions under which particular processes in a concurrent system can starve (or otherwise fail) by analyzing the constrained expression representation of the system. It is therefore important that strings representing these behaviors be included in the interpreted language of the constrained expression representation. We found that it is easier to devise translation rules for generating a system expression from a design, when event sequences that result in the failure of processes are considered legal behaviors, if every prefix of the system expression is viewed as representing a potential event sequence. The prefixes of the system expression are used instead of the language of the system expression in generating the interpreted language of a constrained expression description, therefore, in order to simplify the process for deriving an appropriate system expression from a design of the system.

Finally, we consider the less restrictive relationship among the various alphabets in the revised formulation of constrained expressions. By demonstrating an effective procedure that produces a Wileden constrained expression with the same Wileden interpreted language as any given constrained expression, the next proposition shows that permitting the terminal and constraint alphabets to intersect non-trivially does not increase the descriptive power of constrained expressions.

(4.10) **Proposition.** *Given a constrained expression,* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *there is a Wileden constrained expression,* $\mathbf{D}' = (\mathbf{F}', \epsilon')$, *with the same Wileden interpreted language as* $\mathbf{D}$.

Proof. Let $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\widehat{S} = \{S_j\}_{j \in J}$, and $\widehat{C} = \{\kappa_j\}_{j \in J}$. We describe how to construct ϵ' and $\mathbf{F}' = (A', E', \widehat{S}', \widehat{C}')$ from $\mathbf{D}$. As the example following this proof illustrates, the construction does not work if the shuffle operator appears in ϵ. If the shuffle operator does appear in ϵ, however, a regular expression that generates the same regular language as ϵ, but does not involve the shuffle operator, can be substituted for ϵ. (Generating such a regular expression is an effective procedure.) Without loss of generality, therefore, we assume that the shuffle operator does not appear in ϵ.

To construct $\mathbf{D}'$ we let S_j', for $j \in J$, denote the alphabet obtained from the constraint alphabet S_j by subscripting the symbols in S_j with a j. We then define $E' = E$, $A' = \left(\bigcup_{j \in J} S_j'\right) \cup E'$, and $\widehat{S}' = \{S_j'\}_{j \in J}$. Next we define subscripting homomorphisms, $s_j: S_j^* \to S_j'^*$, by extending the maps on symbols, $s_j(a) = a_j$, for $a \in S_j$, $j \in J$, to homomorphisms on strings. We use these homomorphisms to define constraints κ_j' over S_j', so that $\kappa_j' = s_j(\kappa_j)$, for $j \in J$, and we take $\widehat{C} = \{\kappa_j'\}_{j \in J}$. To obtain the regular expression ϵ', we first define the symbol a_I, for any $a \in A$ and subset of indices $I \subseteq J$, by $a_I = \prod_{i \in I} a_i$.[1] With every event symbol $a \in A$, we associate the subset of indices, $I \subseteq J$, defined by $a \in \bigcap_{i \in I} S_i$ and $a \notin \bigcup_{j \notin I} S_j$. We then define the homomorphism, $f: A^* \to A'^*$,

---

[1] The product symbol, $\prod$, signifies the concatenation of symbols (or expressions) indexed by a designated set (in this case, the set I). Of course, the order in which symbols (expressions) are concatenated is usually significant, and we assume that the order in which the symbols (expressions) in such a product are concatenated is determined by the indexing set, which must therefore by totally ordered. For this construction, however, the symbols a_i, for $i \in I$, can be concatenated together in any arbitrary order to produce a_I, and so no particular order is specified here.

by extending the map,

$$f(a) = \begin{cases} a_I a, & \text{if } a \in E; \\ a_I, & \text{otherwise,} \end{cases}$$

where I is the subset of indices associated with a described above, from a map on symbols to a homomorphism on strings. We finally define $\epsilon' = f(\epsilon)$. (This construction is illustrated below.)

We now show that $\mathcal{WL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}')$, by proving that each of the sets $\rho_E(L)$ and $\rho_{E'}(L')$ is contained in the other, where L and L' denote the constrained languages of $\mathbf{D}$ and $\mathbf{D}'$, respectively, i.e., $L = \mathcal{L}(\epsilon)|_{\widehat{C}}$ and $L' = \mathcal{L}(\epsilon')|_{\widehat{C}'}$.

$\rho_E(L) \subseteq \rho_{E'}(L')$:

We first show that $f(L) \subseteq L'$. Now $L \subseteq \mathcal{L}(\epsilon)$ implies that $f(L) \subseteq f\Big(\mathcal{L}(\epsilon)\Big) \subseteq \mathcal{L}\Big(f(\epsilon)\Big) = \mathcal{L}(\epsilon')$, and so we need only show that every string in $f(L)$ satisfies the constraints of $\widehat{C}'$ to conclude that $f(L) \subseteq L'$.

For $j \in J$, we know that $\rho_{S_j}(L) \subseteq \mathcal{L}(\kappa_j)$, since L satisfies the constraints of $\widehat{C}$. As s_j respects the language structure of event expressions (see (2.9)) and $\kappa'_j = s_j(\kappa_j)$, this last inclusion implies that $s_j \circ \rho_{S_j}(L) \subseteq s_j\Big(\mathcal{L}(\kappa_j)\Big) = \mathcal{L}\Big(s_j(\kappa_j)\Big) = \mathcal{L}\Big(\kappa'_j\Big)$. Observe, however, that for $a \in A$,

$$s_j \circ \rho_{S_j}(a) = \rho_{S'_j} \circ f(a) = \begin{cases} a_j, & \text{if } a \in S_j; \\ \lambda, & \text{otherwise.} \end{cases}$$

Since the extension of a map on symbols to a homomorphism on strings is unique, we conclude that $s_j \circ \rho_{S_j} = \rho_{S'_j} \circ f : A^* \to S'^*_j$. From this and the previous inclusion, we have $\rho_{S'_j}\Big(f(L)\Big) = s_j \circ \rho_{S_j}(L) \subseteq \mathcal{L}\Big(\kappa'_j\Big)$, and so every string in $f(L)$ satisfies κ'_j, for $j \in J$, as desired.

Now $f(L) \subseteq L'$ and $E = E'$ implies that $\rho_E\Big(f(L)\Big) \subseteq \rho_{E'}(L')$. But for $a \in A$, we have

$$\rho_E(a) = \rho_E \circ f(a) = \begin{cases} a, & \text{if } a \in E; \\ \lambda, & \text{otherwise,} \end{cases}$$

and so $\rho_E = \rho_E \circ f: A^* \to E^*$. We therefore conclude that $\rho_E(L) = \rho_E\Big(f(L)\Big) \subseteq \rho_{E'}(L')$, as desired.

$\rho_{E'}(L') \subseteq \rho_E(L)$:

We first show that there is a set $L'' \subseteq L$, with $f(L'') = L'$. For this, observe that $\mathcal{L}(\epsilon') = \mathcal{L}\Big(f(\epsilon)\Big) = f\Big(\mathcal{L}(\epsilon)\Big)$, since the shuffle operator does not appear in ϵ (see (2.9)), and so f maps $\mathcal{L}(\epsilon)$ onto $\mathcal{L}(\epsilon')$. As $L' \subseteq \mathcal{L}(\epsilon')$, therefore, we conclude that there is a set $L'' \subseteq \mathcal{L}(\epsilon)$ with $f(L'') = L'$. (We could take $L'' = f^{-1}(L') \cap \mathcal{L}(\epsilon)$, for instance.)

To show that $L'' \subseteq L$, we must show that every string in L'' satisfies the constraints of $\widehat{C}$. Now, since $f(L'') = L'$ and the strings in L' satisfy the constraints of $\widehat{C}'$, we conclude that $\rho_{S'_j} \circ f(L'') \subseteq \mathcal{L}\Big(\kappa'_j\Big)$, for $j \in J$. But $\rho_{S'_j} \circ f = s_j \circ \rho_{S_j}$ (see above), and as s_j respects the language structure of event expressions and $\kappa'_j = s_j(\kappa_j)$, we therefore argue that $\rho_{S'_j}\Big(f(L'')\Big) \subseteq \mathcal{L}\Big(\kappa'_j\Big)$ implies $s_j\Big(\rho_{S_j}(L'')\Big) \subseteq \mathcal{L}\Big(s_j(\kappa_j)\Big)$, which implies $s_j\Big(\rho_{S_j}(L'')\Big) \subseteq s_j\Big(\mathcal{L}(\kappa_j)\Big)$. As s_j is injective, we conclude that $\rho_{S_j}(L'') \subseteq \mathcal{L}(\kappa_j)$, and so L'' satisfies the constraint κ_j, for $j \in J$.

Now, as $L'' \subseteq L$, $f(L'') = L'$, and $\rho_E \circ f = \rho_E$, we conclude that $\rho_E(L') = \rho_E\Big(f(L'')\Big) \subseteq \rho_E\Big(f(L)\Big) = \rho_E(L)$, as desired. ∎

The construction described above is illustrated by the following example.

(4.11) *Example.* Define $\mathbf{D} = (\mathbf{F}, \epsilon)$, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, as follows. Take

(i) $A = \{a, b, c\}$,

(ii) $E = A$,

(iii) $\epsilon = cc \triangle a^*b^*$,

(iv) $\widehat{S} = \{S_1, S_2\}$, where $S_1 = \{a, b\}$ and $S_2 = \{c, a\}$, and

(v) $\widehat{C} = \{\kappa_1, \kappa_2\}$, where $\kappa_1 = (ab)^*$ and $\kappa_2 = ca^*c$.

Observe that $\mathcal{WL}(\mathbf{D}) = \{cc, cacb, cabc\}$.

Before the construction described in proposition (4.10) can be applied, ϵ must

be transformed into an expression that does not involve the shuffle operator. We therefore write,

$$\epsilon = a^*ca^*ca^*b^* \vee a^*b^*cb^*cb^* \vee a^*ca^*b^*cb^*.$$

The construction then yields,

$$A' = \{ a, b, c, a_1, a_2, b_1, c_2 \},$$
$$E' = \{ a, b, c \}$$
$$S_1' = \{ a_1, b_1 \}, S_2' = \{ c_2, a_2 \}, \text{and } \widehat{S}' = \{ S_1', S_2' \},$$
$$\kappa_1' = (a_1b_1)^*, \kappa_2' = c_2a_2^*c_2, \text{and } \widehat{C}' = \{ \kappa_1', \kappa_2' \}, \text{and}$$
$$\epsilon' = (a_1a_2a)^*(c_2c)(a_1a_2a)^*(c_2c)(a_1a_2a)^*(b_1b)^*$$
$$\vee (a_1a_2a)^*(b_1b)^*(c_2c)(b_1b)^*(c_2c)(b_1b)^*$$
$$\vee (a_1a_2a)^*(c_2c)(a_1a_2a)^*(b_1b)^*(c_2c)(b_1b)^*.$$

The constrained language of $\mathbf{D}'$ consists of three strings, c_2cc_2c, $c_2ca_1a_2ac_2cb_1b$, and $c_2ca_1a_2ab_1bc_2c$. When projected on the terminal alphabet E', it produces the Wileden interpreted language, $\mathcal{WL}(\mathbf{D}') = \{ cc, cacb, cabc \}$.

To see that the assumption that the shuffle operator does not appear in ϵ is required, note that if A', E', $\widehat{S}'$ and $\widehat{C}'$ are as defined above and $\epsilon' = (c_2c)(c_2c) \Delta (a_1a_2a)^*(b_1b)^*$, then the string $c_2ca_1a_2c_2cab_1b$ lies in the constrained language of $\mathbf{D}'$, and so $ccab$ belongs to $\mathcal{WL}(\mathbf{D}')$, but not to $\mathcal{WL}(\mathbf{D})$.

We permit the terminal and constraint alphabets to intersect non-trivially in the revised formulation of constrained expressions in order to minimize the number of symbols required in the augmented alphabet of a constrained expression and to keep the constraints from becoming unnecessarily complex. The appearance of symbols representing a particular event in a legal behavioral trace of a system must often be restricted because of a number of independent semantic concerns. In strings representing behaviors of a system of communicating processes, for example, symbols associated with the starvation of a particular process must be constrained

so that at least one starvation symbol appears in traces that do not contain a symbol associated with the completion of the process, but no starvation symbols appear in traces that do contain the completion symbol, and so that no starvation symbols appear in traces in which some communication can be realized that would permit the process to proceed. A Wileden constrained expression for such a system would require either a single, relatively complicated constraint to assure that each of these restrictions on the number of starvation symbols appearing in legal behavioral traces are satisfied, or distinct symbols representing the same starvation event, to be used in separate simpler constraints. Simple constraints over relatively small constraint alphabets are preferable to a single monolithic constraint over a large alphabet, as each constraint helps modularize the analysis of strings in the interpreted language of the constrained expression, by highlighting restrictions imposed on the symbols of the corresponding constraint alphabet. A proliferation of event symbols, many of which represent essentially the same system event, however, complicates the translation rules for deriving system expressions and produces system expressions that are unnecessarily long and hard to interpret. In the revised formulation of constrained expressions, there is no need to express all the restrictions imposed on a particular event in a single constraint or to introduce redundant event symbols, since the same symbol can be used to represent an event in as many constraints as necessary and, if the event is of interest, also in the terminal alphabet.

The above results are summarized in the following theorem, which shows that the original and revised formulations of constrained expressions are, in fact, equivalent.

(4.12) **Theorem.** *Given a Wileden constrained expression,* $\mathbf{D}'$, *there is an effective procedure for obtaining a constrained expression,* $\mathbf{D}$, *such that* $\mathcal{IL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}')$, *and conversely, given a constrained expression,* $\mathbf{D}$, *there is an effective procedure for obtaining a Wileden constrained expression,* $\mathbf{D}'$, *such that* $\mathcal{WL}(\mathbf{D}') = \mathcal{IL}(\mathbf{D})$.

Proof. Given a Wileden constrained expression, $\mathbf{D}'$, proposition (4.9) describes

an effective procedure for obtaining a constrained expression, $\mathbf{D}$, with the desired property.

Given the constrained expression, $\mathbf{D}$, first use the construction of proposition (4.8) to obtain a constrained expression, $\mathbf{D}''$, such that $\mathcal{IL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}'')$. Then use the construction of proposition (4.10) to obtain a Wileden constrained expression, $\mathbf{D}'$, such that $\mathcal{WL}(D'') = \mathcal{WL}(\mathbf{D}')$. We then have $\mathcal{IL}(\mathbf{D}) = \mathcal{WL}(\mathbf{D}')$, as desired. ∎

CHAPTER III

SDYMOL CONSTRAINED EXPRESSION REPRESENTATIONS

Having formally described what a constrained expression is and how it is associated with an interpreted language, we now turn our attention to the generation of constrained expressions. To use constrained expressions for analyzing the behavior of concurrent systems, we must first devise a technique for associating a design of a system with a constrained expression, so that the interpreted language of the constrained expression represents exactly the possible behaviors of the system. Our approach is to formulate a procedure for generating constrained expression representations for systems expressed in a specific design language, based on the semantics of the language.

Of course, constrained expressions are only useful for analyzing the behavior of systems expressed in a particular design language if the procedure for generating constrained expression representations for such systems is correct, i.e., if the semantics of the design language are completely and accurately modeled, so that the interpreted language of the constrained expression generated from a system design represents all and only the possible behaviors of the system. The formulation of a procedure for generating constrained expressions to represent systems expressed in a specific design language is thus a critical prerequisite to constrained expression-based analysis of the behaviors of systems written in the design language. This step of tailoring the constrained expression framework to a particular design language need only be performed once, and is analogous to steps required by other techniques for analyzing the behavior of systems. In most axiomatic techniques for proving properties of concurrent systems, for example, logical predicates are attached to specific points within each process' code and the programs' statements are used to direct a formal reasoning process involving these predicates. A necessary prereq-

uisite to such techniques is thus the formulation of rules for reasoning about these predicates, based on the semantics of the programming language.

In this chapter we describe a technique for generating a constrained expression representation of a system from an SDYMOL design for the system. The SDYMOL design language is introduced first. The technique is then described and informally justified. (A formal proof of correctness cannot be given without a formal definition of the SDYMOL design language semantics and is tangential to the purpose of this chapter, which is simply to illustrate that constrained expressions can be used to represent the behavior of concurrent systems, and to provide a vehicle for exploring the analytical potential of the constrained expression framework.) An example is presented to illustrate the technique. The constrained expression generated in this example is used in subsequent chapters to show how constrained expressions can be simplified and analyzed to validate specific behavioral properties of the systems they describe.

§1. SDYMOL designs

SDYMOL is a simplified version of the Dynamic Modelling Language (DYMOL), which was designed to be used with the Dynamic Process Modelling Scheme (DPMS). It is a high-level design language that focuses on interprocess communication and synchronization. Language features that are not essential for describing concurrency are kept to a minimum. SDYMOL thus possesses a relatively simple semantics when compared to other languages for describing concurrent systems, and yet it facilitates the representation of the potential interactions between the processes in a concurrent system. For this reason, we have found it to be a useful research tool for exploring the capability of analysis techniques based on constrained expressions. We summarize the relevant features of SDYMOL below.

A concurrent system in SDYMOL is considered to be composed of individual sequential processes, communicating with each other by means of message transmissions. The SDYMOL program for a process precisely specifies the ways in which the process may interact with other processes through message transmission, but only abstractly describes the local, internal activities of the process itself.

Message transmission as modeled in SDYMOL is both a communication and a synchronization mechanism. A process may send a message through an outbound *port* into a *link* associated with that port. The link is essentially an unbounded, unordered repository that is used to mediate the asynchronous message transmission activity of SDYMOL processes. Sending a message may be viewed as copying the current contents of the process' *buffer* (a distinguished memory location within the process) into the designated link leaving the buffer's contents unchanged. Having sent a message, the sending process may continue with subsequent activities as described by its SDYMOL program.

A process may also request receipt of a message through one of its inbound ports. Such a request can be fulfilled whenever at least one link containing one or more messages is connected to the designated inbound port by an interprocess communication *channel*. When the request is fulfilled a link is nondeterministically selected from among those that contain one or more messages and are connected to the designated port, and a message is nondeterministically chosen from those residing in the selected link. The message is then removed from the link and placed into the buffer of the requesting process. If no messages are residing in any of the links connected to the designated inbound port when a receive request is lodged, the requesting process waits. The wait continues at least until a message becomes available in a link connected to the designated inbound port. The appearance of a message in such a link, however, does not necessarily end a wait, since competing requests might be lodged in the interim and requests need not be serviced in the order in which they were made.

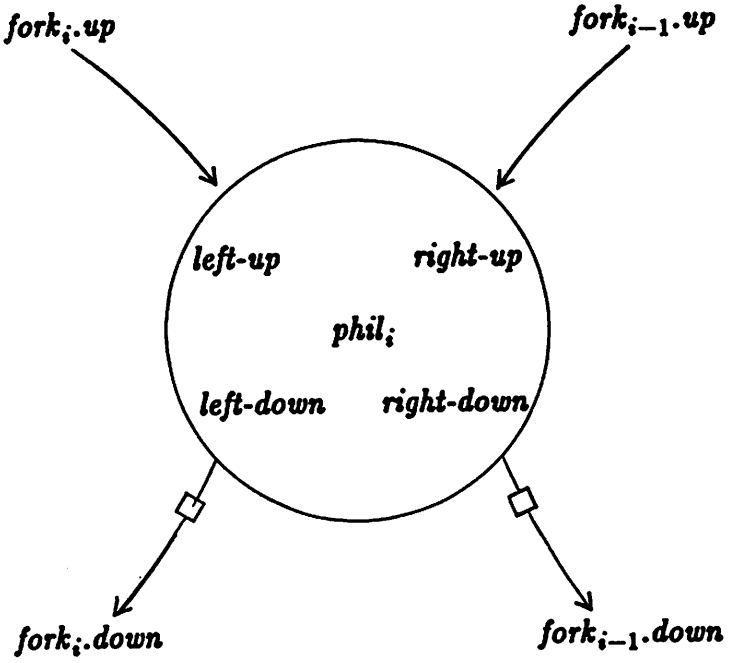
The SDYMOL language is a simple programming-like language whose syntax is based on Algol 60. Among its features are instructions for message transmission (**send** and **receive**), and a standard set of control flow constructs. Decisions based upon internal process computation are modeled as nondeterministic choices (e.g., **if internal test** ..., or **while internal test do** ...).

The main distinction between SDYMOL and DYMOL is that DYMOL provides constructs for changing the overall structure of a system and SDYMOL does not, so that the structure of an SDYMOL system is static, but processes and communication channels can be dynamically created and destroyed in a DYMOL system. Additionally, internal process computation can be modeled in SDYMOL by a primitive statement that does not contain any SDYMOL keywords, but consists of a single identifier. A developer would presumably choose a particular identifier for its connotative value. Such statements, however, are essentially null statements, serving simply as placeholders for activities that may be elaborated in subsequent system descriptions.

We illustrate the essential features of the SDYMOL design language using an SDYMOL design for the solution presented in [HOAR78] to the dining philosophers problem. For this problem, five philosophers are assumed to share a common dining room. The dining room contains a circular table, surrounded by five chairs and set with five forks. Each philosopher spends the productive years of his/her life thinking and eating. Upon becoming hungry, a philosopher enters the dining room and sits in his/her chair. Dinner is always spaghetti, which requires two forks. When the forks to the philosopher's immediate right and left become available, therefore, the philosopher picks them up, eats, and then replaces the forks and leaves the dining room to continue thinking. The problem is to design a system that exhibits the behavior of the five philosophers. (The dining philosophers problem is due to E. W. Dijkstra.)

The SDYMOL programs for the processes in a design of a solution to this

```
phil_i :
  while internal test do
    begin
      think_i ;
      receive right-up;
      receive left-up;
      eat_i ;
      send left-down;
      send right-down
    end.
```




```
fork_i :
  set buffer := ok;
  do forever
    begin
      send up;
      receive down;
    end.
```

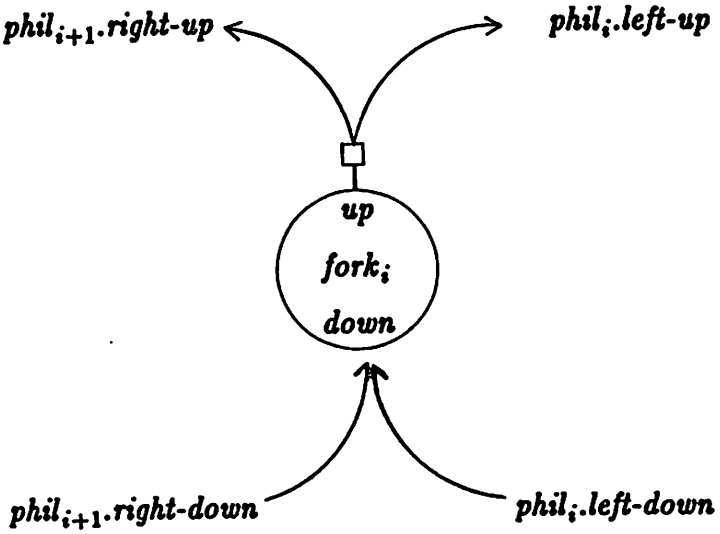


Figure 1

SDYMOL programs for a solution to the dining philosophers problem

Process/port name	Abbreviation
phil	*p*
fork	*f*
down	*d*
up	*u*
left-down	*ld*
right-down	*rd*
left-up	*lu*
right-up	*ru*

Figure 2

Abbreviations for process and port names

problem are given in figure 1. Using these programs and the abbreviations of figure 2, the concurrent system is defined in figure 3 to consist of ten processes, represented by the circles labeled p_i and f_i, $0 \leq i \leq 4$. Boxes represent links, which correspond to outbound ports and, in this example, are all initially empty. Arrows connecting links and inbound ports represent interprocess communication channels. The arithmetic operations in the subscripts in these and all other figures and in the discussion of this example are performed modulo 5.

The "philosopher processes", p_i, for $0 \leq i \leq 4$, model the activity of the five philosophers. The "fork processes", f_i, for $0 \leq i \leq 4$, synchronize the activities of the philosopher processes. The ith fork process is intended to assure that the ith fork is used in a mutually exclusive fashion. The availability of the ith fork is represented by an *ok* message residing in the link $f_i.u$. This link is connected to ports of the two philosopher processes representing the philosophers that may use the fork (i.e., to $p_{i+1}.ru$, since the fork lies to the immediate right of the $(i+1)$th philosopher, and to $p_i.lu$, since the fork lies to the immediate left of the ith philosopher). By attempting to receive a message through the appropriate ports, therefore, these processes can determine if the ith fork is available. Note

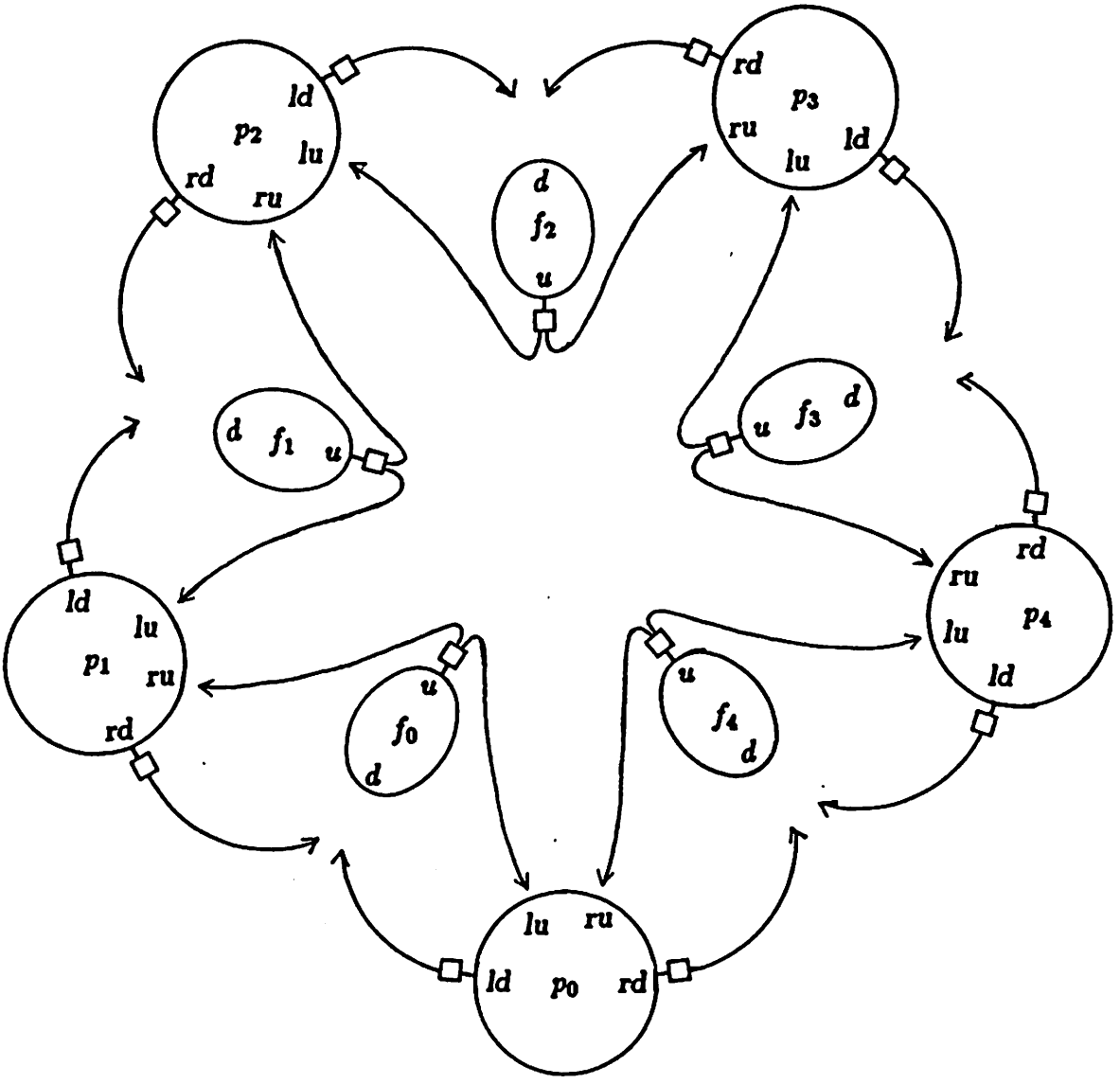


Figure 3

A concurrent system to solve the dining philosophers problem

that if both philosopher processes check for the i th fork simultaneously (modeling the situation where both philosophers try to pick up the fork at the same time), the fact that there is only a single *ok* message residing in the link assures that one of the processes receives the message and the other process waits until a message becomes available for it to receive. (Thus, one philosopher actually picks up the fork, while the other one waits until the fork becomes available.) The links associated with the ports $p_{i+1}.rd$ and $p_i.ld$ are connected to the inbound port $f_i.d$. When the philosopher associated with the process that receives the *ok* message is finished with the fork, therefore, the process can signal this fact to the fork process f_i by depositing an *ok* message in the appropriate link.

The **while internal test** statement, which constitutes the body of a philosopher process, models some internal computation performed by the philosopher to determine whether to repeat another cycle of thinking and eating. This cycle is represented by the six statements that constitute the scope of the **while** statement. The first of these statements, the *think*$_i$ statement, is an example of an identifier statement. It is meant to model an internal activity (thinking) performed by the philosopher. As mentioned above, however, it is a semantically null statement. The remaining five statements represent the activity of the philosopher while in the dining room. Before the philosopher eats (modeled by the *eat*$_i$ statement), the philosopher process receives messages indicating that the forks to the philosopher's immediate right and left are available (thus, the **receive** statements). When the philosopher finishes eating, the philosopher process signals the appropriate fork processes by depositing messages in the links connected to their inbound ports.

A fork process begins by initializing its buffer. It then cycles forever, depositing a message in the link connected to its *up* port to indicate that the associated fork is available, and then waiting to receive a message through its *down* port, indicating that one of the philosophers has put the fork down, so that it is once again available for use.

Clearly, this design focuses on the interaction between processes, rather than the internal activities of the processes themselves. A designer would do well to analyze a high level design like this first, before proceeding to elaborate the processes. We have argued informally that the fork processes assure the forks are used in a mutually exclusive fashion. When informally reasoning about the behavior of a concurrent system, however, it is easy to overlook unexpected event sequences that could invalidate an argument. We would like, therefore, to rigorously prove this fact. A designer might also be concerned with whether a philosopher, as modeled in this system, could be denied access to a fork forever, and if so, under what conditions this could happen. These are the types of questions that analysis of the constrained expression representation of this system can address.

§2. The derived constrained expression

In this section we show how to derive a constrained expression representation of a distributed system from an SDYMOL design for the system. We first describe the augmented and terminal alphabets, then the system expression, and finally the constraints and constraint alphabets that compose this constrained expression, which we refer to as the *derived constrained expression representation* of the system.

In this description we use the following notational conventions to refer to a system and its components.

(2.1) *Notation.* The system described by an SDYMOL design is denoted by Σ. The set of processes of Σ is represented by Q, the set of inbound ports by P, the set of links, where each link is identified with its associated outbound port, by L and the set of message types by M. The symbol $@ \in M$ is reserved to represent undefined messages. The SDYMOL code for a process $q \in Q$ is denoted by $\mathbf{code}_q$, the set of inbound ports of the process q by $P(q)$, and the set of links (outbound ports) of the process q by $L(q)$.

In the example of the previous section, for instance, Σ represents the distributed system described by figures 1–3, while, using the abbreviations of figure 2,

$$
\begin{aligned}
&Q = \{ f_i, p_i \}_{0 \le i \le 4}, \\
&P = \{ f_i.d, p_i.lu, p_i.ru \}_{0 \le i \le 4}, \\
&L = \{ f_i.u, p_i.rd, p_i.ld \}_{0 \le i \le 4}, \\
&M = \{ \mathbf{Q}, ok \}, \\
&P(p_i) = \{ p_i.lu, p_i.ru \}, \\
&P(f_i) = \{ f_i.d \}, \\
&L(p_i) = \{ p_i.rd, p_i.ld \}, \text{ and} \\
&L(f_i) = \{ f_i.u \}, \text{ for } 0 \le i \le 4.
\end{aligned}
\tag{2.2}
$$

The $\mathbf{code}_{p_i}$ and $\mathbf{code}_{f_i}$, for $0 \le i \le 4$, are given in figure 1.

The augmented and terminal alphabets

In order to capture the semantics of SDYMOL, certain symbols are needed in the augmented alphabet of the derived constrained expression representation of an SDYMOL system Σ. These are shown in figure 4. Conceptually, it is helpful to associate these symbols with events that can occur in the behaviors of Σ, as indicated in this figure. Strictly speaking, however, these symbols are needed in constraints for describing permissible event sequences and may or may not correspond to actual system events. This figure contains symbols representing channels (which are established when the system is started up), the use and modification of buffer contents, and the transmission and receipt of messages. Additionally, a *stop* symbol is associated with the completion of a process, which occurs when the process has finished the execution of the sequence of statements that appear in its design, while a w symbol represents an attempt to receive that results in starvation (the process waiting forever). The *ne* (non-event) symbol for a process is an example of a symbol that does not represent an actual system event. In fact it can never occur in a string

Symbol	*Associated Event*
$r(l,p,m)$	A message of type m is transmitted from link l through port p ($m \neq @$).
$s(l,m)$	A message of type m is sent to link l ($m \neq @$).
$ch(l,p)$	Link l and port p are connected by a channel.
$d(q,m)$	The buffer of process q is modified so that it contains a message of type m ($m = @$ represents an undefined buffer).
$u(q,m)$	The buffer of process q is presumed to contain a message of type m ($m = @$ represents an undefined buffer).
$w(p)$	The process containing inbound port p starves waiting for a message on a channel connected to p.
$stop(q)$	Process q terminates normally.
$ne(q)$	A non-event involving process q (marks positions in the process expression for q that cannot be reached in legitimate system behaviors).

Figure 4

SDYMOL event symbols

representing a legitimate system behavior. It is used in a constraint that assures certain patterns of event symbols do not occur in constrained prefixes, as required by the SDYMOL design notation semantics, so that constrained prefixes correspond to legitimate system behaviors. We discuss the role of this symbol in more detail below.

In addition to the symbols in figure 4, the augmented alphabet contains

symbols representing internal process computations that are abstractly modeled in the SDYMOL design for Σ by statements consisting of single identifiers. No meaning is attached to these symbols, which serve only to remind the developer of activities that may be elaborated in more detailed system designs. For these symbols we usually use the identifier itself (or an abbreviated form of the identifier).

The augmented alphabet for the derived constrained expression representation of Σ is thus given by

$$A = \{\, r(l,p,m),\ s(l,m),\ ch(l,p),\ d(q,m'),\ u(q,m'),\ w(p),\ stop(q),\ ne(q) \,\} \cup F,$$

where F is a (possibly empty) set of event symbols, q ranges over Q, l ranges over L, p ranges over P, m and m' range over M, and $m \neq \mathbb{Q}$. The set F contains any symbols required, in addition to the symbols defined in figure 4, to represent activities that have been abstractly modeled in the SDYMOL design using identifiers.

Taking $F = \{\, think_i,\ eat_i \,\}_{0 \le i \le 4}$ to represent the internal computations modeled by the $think_i$ and eat_i statements in figure 1, for example, the augmented alphabet for the derived constrained expression representation of the solution to the dining philosophers problem (see figures 1 and 3) is obtained from A using the values for Q, P, L and M given in (2.2).

The decision as to what symbols represent events of interest and should be retained in the interpreted language of the derived constrained expression representation of an SDYMOL system Σ depends on the particular question that analysis of the representation is intended to address. For this reason the only requirement that we impose on the terminal alphabet of the derived constrained expression representation of Σ is that it is a subset of the augmented alphabet.

Ideally, the terminal alphabet should contain the symbols required to analyze some aspect of the modeled system's behavior, determined by the designer's interests, and as few other symbols as possible. Suppose, for example, a designer

wishes to determine if any of the philosopher processes in the SDYMOL design in figures 1 and 3 can starve, and if the desired mutually exclusive utilization of the forks is realized. In terms of sequences of event symbols, the former question can be interpreted as asking if there are any legal behavioral traces of Σ that contain a $w(p_i.lu)$ or $w(p_i.ru)$ symbol, for any $0 \leq i \leq 4$, and the latter as asking if there are any legal behavioral traces of Σ that contain an $r(f_i.u, p_i.lu, ok)$ symbol (representing the ith philosopher picking up the fork on his/her left) followed by an $r(f_i.u, p_{i+1}.ru, ok)$ symbol (representing the $(i+1)$th philosopher picking up the fork on his/her right) with no intervening $r(p_i.ld, f_i.d, ok)$ symbols (representing the ith fork becoming available for use again), or an $r(f_{i-1}.u, p_i.ru, ok)$ symbol followed by an $r(f_{i-1}.u, p_{i-1}.lu, ok)$ symbol with no intervening $r(p_i.rd, f_{i-1}.d, ok)$ symbols. In this case, therefore, we might choose

$$E = \{ w(p_i.ru),\ w(p_i.lu) \}_{0 \leq i \leq 4} \cup \{ r(l, f_i.d, ok),\ r(l, p_i.lu, ok),\ r(l, p_i.ru, ok) \}_{l \in L}$$

as the terminal alphabet for the derived constrained expression representation of Σ.

The system expression

The system expression, ϵ, for the derived constrained expression representation of an SDYMOL system Σ consists of an *initial expression*, ι, followed by the interleave of *process expressions*, π_q, one for each process q in Q:

$$\iota \left(\underset{q \in Q}{\triangle} \pi_q \right).$$

The initial expression represents the overall structure of Σ (constituent processes and interprocess connectivities) and the initial status of its links and buffers. Each process expression represents the sequential activity of one of the processes in Σ.

The initial expression is simply the concatenation of certain symbols from the constraint alphabets of Σ, determined by inspection of the design for Σ. It begins

with a sequence of $ch(l,p)$ symbols representing the interprocess communication pathways in Σ. This sequence contains a single $ch(l,p)$ symbol for each link l and port p that are connected by a channel. This initial string of $ch(l,p)$ symbols is followed by a string of $d(q,\Omega)$ symbols, one for each process $q \in Q$, indicating the contents of the processes' buffers are initially undefined. Finally, $s(l,m)$ symbols are concatenated to the end of the initial expression to represent each message of type m that initially resides in a link l. The SDYMOL system expression for Σ thus begins with a series of constraint alphabet symbols which encode the initial status of links, buffers and interprocess communication pathways in Σ.

The process expressions, whose interleave constitutes the remainder of the SDYMOL system expression for Σ, are obtained from the SDYMOL code for the processes in Σ through the statement-by-statement application of the set of translation rules given in figure 5. Each rule indicates that an SDYMOL statement of the given form is to be transformed into the symbol sequence that appears to the right of the arrow. For each process $q \in Q$, the rules are applied to the code corresponding to q (i.e., to $\mathbf{code}_q$).

The application of the rules pertaining to the SDYMOL basic statements, the rules $T_1 - T_4$ of figure 5, is a simple matter of replacing the statement by the indicated sequence of symbols from the augmented alphabet. (In T_4, *identifier-symbol* denotes the event symbol associated with the internal process computation modeled by the SDYMOL statement consisting of the single identifier *identifier*.) The remaining rules are applied recursively, that is, the indicated action is performed on the expression resulting from application of the appropriate rule to the embedded statement(s) of the statement being translated. Thus, for instance, the statement

while internal test do set buffer := m

appearing in the code for a process q would be transformed into $d(q,m)^*$ by the application of T_3 and T_6.

Statement	*Translation*	*Rule*
send l	$\rightarrow \left(\bigvee_{m \neq \Omega} u(q,m)\, s(l,m) \right) \vee u(q,\Omega)$	T_1
receive p	$\rightarrow \left(\bigvee_{l,m \neq \Omega} r(l,p,m)\, d(q,m) \right) \vee w(p)\, ne(q)$	T_2
set buffer $:= m$	$\rightarrow d(q,m)$	T_3
identifier	$\rightarrow$ *identifier-symbol*	T_4
do forever ⟨s⟩	$\rightarrow \{\langle s \rangle\}^*\, ne(q)$	T_5
while internal test do ⟨s⟩	$\rightarrow \{\langle s \rangle\}^*$	T_6
while buffer $= m$ **do** ⟨s⟩	$\rightarrow (u(q,m)\, \{\langle s \rangle\})^* \left(\bigvee_{n \neq m} u(q,n) \right)$	T_7
if internal test then ⟨ss⟩ **else** ⟨s⟩	$\rightarrow \{\langle ss \rangle\} \vee \{\langle s \rangle\}$	T_8
if internal test then ⟨s⟩	$\rightarrow \{\langle s \rangle\} \vee \lambda$	T_9
if buffer $= m$ **then** ⟨ss⟩ **else** ⟨s⟩	$\rightarrow u(q,m)\, \{\langle ss \rangle\} \vee \left(\bigvee_{n \neq m} u(q,n)\, \{\langle s \rangle\} \right)$	T_{10}
if buffer $= m$ **then** ⟨s⟩	$\rightarrow u(q,m)\, \{\langle s \rangle\} \vee \left(\bigvee_{n \neq m} u(q,n) \right)$	T_{11}
⟨sl⟩ ; ⟨s⟩	$\rightarrow \{\langle sl \rangle\}\, \{\langle s \rangle\}$	T_{12}
q : ⟨s⟩	$\rightarrow \{\langle s \rangle\}\, stop(q)$	T_{13}

Where q denotes the process defined by the SDYMOL program, curly brackets ({ }) signify the recursive application of these rules, l ranges over L, and m and n range over M.

Figure 5

SDYMOL translation rules

When interpreted using the associations established in figure 4, most of these rules are easily understood. The sequence of events produced by a process $q \in Q$ executing a **send** l statement, for example, depends on the contents of the process' buffer, and so this statement translates into the disjunction of a number of different alternatives. If the buffer contains a message, that message is placed in l, hence the disjunction over all defined message types to the right of the arrow in T_1. Otherwise the buffer is undefined, as indicated by the final alternative in the translation column of this rule. As we show below, the constraining context of the derived constrained expression representation of Σ contains a constraint to assure that the appropriate alternative from the translation of this statement is selected in a particular behavioral trace.

With the possible exception of T_2 and T_5, in which the role of the $ne(q)$ symbols may not be obvious, the rest of the translation rules can be similarly interpreted. Clearly the $w(p)\,ne(q)$ alternative in the translation of a **receive** p statement in the code for process q represents the possibility that the request for a message through the port p might never be serviced, so that the process q waits forever. If this happens, however, the SDYMOL language semantics require that q does not participate in future system events, and so we need to assure that if a $w(p)$ symbol appears in a constrained prefix of the derived constrained expression representation for Σ, where p is an inbound port of process q (i.e., $p \in P(q)$), no symbols associated with q appear in the prefix anywhere to the right of this $w(p)$ symbol. We do this by placing the non-event symbol, $ne(q)$, immediately after the $w(p)$ symbol in T_2 and then including a constraint in the constraining context of the derived constrained expression representation for Σ that filters out prefixes containing the non-event symbol for q. By placing the $ne(q)$ symbol immediately after the $w(p)$ symbol in T_2, we are assured that if a prefix of the system expression contains a $w(p)$ symbol, where $p \in P(q)$, and if any other symbols associated with the process q appear anywhere in the prefix to the right of the $w(p)$ symbol, at least

one of these symbols (namely the first one) is an $ne(q)$ symbol. As we show below, a constraint in the derived constrained expression representation of Σ filters out prefixes containing $ne(q)$ symbols. This permits us to conclude that, if $p \in P(q)$, no symbols associated with the process q can appear to the right of a $w(p)$ symbol in a constrained prefix of the derived constrained expression representation of Σ, as required by the SDYMOL design notation semantics.

The $ne(q)$ symbol is used in a similar fashion in T_5. According to the SDYMOL semantics, a process can never "exit" a **do forever** loop, that is, when a process executes a statement of the form **do forever** ⟨s⟩, it repeats the embedded statement ⟨s⟩ indefinitely, never executing any statements that logically follow the **do forever** statement in the process' code. The Kleene star to the right of the arrow in T_5 is intended to represent this indefinite iteration. By itself, however, the Kleene star does not accurately describe the semantics of "doing forever". It expresses the fact that the code corresponding to the embedded statement is executed some finite number of times, but not the fact that no other statements in the process' code can ever be reached. When translating a **do forever** statement in the code for a process q, therefore, we put an $ne(q)$ symbol immediately after the Kleene star that represents the indefinite iteration of the embedded statement. This assures that prefixes from the derived system expression that represent event sequences in which the process q exits a **do forever** loop contain an $ne(q)$ symbol, so that the constraint in the derived constrained expression representation of Σ that eliminates prefixes containing $ne(q)$ symbols filters these prefixes out. The $ne(q)$ symbol is thus required in T_5 to accurately express the semantics of the SDYMOL **do forever** statement.

The translation rules in figure 5 provide a means for associating the SDYMOL code for any process with a regular expression over the augmented alphabet. It is convenient for future reference to give a name to this mapping from SDYMOL process designs to regular expressions.

(2.3) *Notation.* We let t denote the map that associates the SDYMOL code for a process with a regular expression using the translation rules of figure 5. The process expressions π_q used in forming the system expression for the derived constrained expression representation of an SDYMOL system Σ are therefore given by $\pi_q = t(\mathbf{code}_q)$, for $q \in Q$.

To generate the system expression for the derived constrained expression representation of the SDYMOL solution to the dining philosophers problem, for example, we first derive an initial expression from figure 3, and then the process expressions from figure 1. Examination of figure 3 shows that there are channels connecting the link $p_i.ld$ with the port $f_i.d$, the link $p_i.rd$ with the port $f_{i-1}.d$, and the link $f_i.u$ with the ports $p_i.lu$ and $p_{i+1}.ru$, for $0 \leq i \leq 4$, and so the initial expression begins with a string of $ch(l,p)$ symbols to represent each of these channels. A $d(q,@)$ symbol is then required for each $q \in Q$, to represent the fact that the buffers of all processes are initially undefined. Finally, as all the boxes representing links in this figure are empty, there are no messages residing in any of the links when the system is started up. No $s(l,m)$ symbols are therefore required in the initial expression in this example. Hence we obtain the initial expression of figure 6. The process expressions for the processes in this system are obtained by applying the translation rules of figure 5 to the $\mathbf{code}_{p_i}$ and $\mathbf{code}_{f_i}$, for $0 \leq i \leq 4$, of figure 1. This produces the process expressions and system expression of figure 6.

Clearly, many prefixes of the system expression of figure 6 do not represent possible behaviors of the system. Consider, for example, the prefix

$$\iota\ think_1\ r(f_1.u, p_1.ru, ok)\ d(p_1, ok)\ w(p_1.lu)\ ne(p_1)\ eat_1, \tag{2.4}$$

where ι denotes the initial expression of figure 6. As there is no $s(f_1.u, ok)$ symbol preceding the $r(f_1.u, p_1.ru, ok)$ symbol, this string represents an event sequence in which an ok message is received from the link $f_1.u$ before any messages of that type have been deposited in the link. Even if an ok message were available in the

$$\iota = \prod_{0 \le i \le 4} ch(p_i.ld, f_i.d)\, ch(p_i.rd, f_{i-1}.d)\, ch(f_i.u, p_i.lu)\, ch(f_i.u, p_{i+1}.ru)$$
$$\prod_{0 \le i \le 4} d(p_i, \mathbb{Q})\, d(f_i, \mathbb{Q}).$$

$$t(\mathbf{code}_{p_i}) =$$
$$\Big[\ think_i$$
$$\Big(\bigvee_{l \in L} r(l, p_i.ru, ok)\, d(p_i, ok) \vee w(p_i.ru)\, ne(p_i)\Big)$$
$$\Big(\bigvee_{l \in L} r(l, p_i.lu, ok)\, d(p_i, ok) \vee w(p_i.lu)\, ne(p_i)\Big)\ eat_i$$
$$\Big(u(p_i, ok)\, s(p_i.ld, ok) \vee u(p_i, \mathbb{Q})\Big)$$
$$\Big(u(p_i, ok)\, s(p_i.rd, ok) \vee u(p_i, \mathbb{Q})\Big)\Big]^*\ stop(p_i)$$

$$t(\mathbf{code}_{f_i}) =$$
$$d(f_i, ok)$$
$$\Big[\ \Big(u(f_i, ok)\, s(f_i.u, ok) \vee u(f_i, \mathbb{Q})\Big)$$
$$\Big(\bigvee_{l \in L} r(l, f_i.d, ok)\, d(f_i, ok) \vee w(f_i.d)\, ne(f_i)\Big)\Big]^*\ ne(f_i)\ stop(f_i)$$

$$\epsilon = \iota\Big(\Big(\mathop{\triangle}_{0 \le i \le 4} t(\mathbf{code}_{p_i})\Big) \triangle \Big(\mathop{\triangle}_{0 \le i \le 4} t(\mathbf{code}_{f_i})\Big)\Big)$$

Figure 6

Initial expression, process expressions and corresponding system expression derived from a solution to the dining philosophers problem

link $f_1.u$, however, an event sequence representing a possible behavior of this system could not contain a $r(f_1.u, p_1.ru, ok)$ symbol, as the link $f_1.u$ and port $p_1.ru$ are not connected by a channel, and so, according to the SDYMOL design notation semantics, messages cannot be transmitted from $f_1.u$ through $p_1.ru$. Furthermore, if a message never becomes available for p_1 to receive through its *left-up* port, so that p_1 is forced to wait forever (as indicated by the $w(p_1.lu)$ symbol in this prefix), the process cannot engage in any subsequent system activities. The symbols $ne(p_1)$ and eat_1, therefore, cannot follow a $w(p_1.lu)$ symbol in a legal behavioral trace of this system. As we show below, a number of constraints in the derived constrained expression representation of the system filter out this prefix.

The constraints and constraint alphabets

Constraints are required in the derived constrained expression representation for an SDYMOL system Σ because some of the SDYMOL language semantics are not expressed by the translation rules of figure 5. Each constraint describes the patterns of event symbols from the associated constraint alphabet that can appear in strings representing legal SDYMOL system behaviors. They are used to restrict the order and number of symbols in prefixes of the system expression that are retained as constrained prefixes, so that the constrained prefixes represent possible system behaviors. We use six different types of constraints for this purpose.

The first type of constraint relates to the flow of messages in Σ. In any legal SDYMOL system behavior, messages can only be received through inbound port p from link (i.e., outbound port) l if there is a communication channel connecting l and p. Hence, a constraint is needed to assure that symbols representing the transmission of messages from l to p appear in a particular behavioral trace only if l and p are connected by a channel, in which case a symbol representing this channel also appears. The constraint alphabet $S_1(l,p) = \{ ch(l,p), r(l,p,m) \}_{m \neq \Theta}$,

is used in stating this behavioral requirement. The corresponding constraint is

$$\kappa_1(l,p) = ch(l,p)\Big(\bigvee_{m \neq @} r(l,p,m)\Big)^* \vee \lambda.$$

Using the associations of figure 4, this constraint is easily understood. If there is a channel connecting l and p, a single $ch(l,p)$ symbol appears in the initial expression of the derived system expression, and, as the process expressions $t(\mathbf{code}_q)$, for $q \in Q$, do not contain any $ch(l,p)$ symbols, every prefix of the system expression satisfies the first alternative of this constraint. Otherwise, no $ch(l,p)$ symbols appear in the system expression, and a prefix satisfies $\kappa_1(l,p)$ only if it does not contain any $r(l,p,m)$ symbols. This assures that a constrained prefix represents an event sequence in which messages are transmitted from l through p only if l and p are connected by a channel, which is precisely the intended constraint on the flow of messages within Σ. As the initial expression of figure 6 does not contain a $ch(f_1.u, p_1.ru)$ symbol, for example, the constraint $\kappa_1(f_1.u, p_1.ru)$ in the constraining context for the derived constrained expression representation of the solution to the dining philosophers problem filters out the prefix of (2.4), which represents an event sequence in which a message is transmitted from $f_1.u$ through $p_1.ru$. The constraining context for the derived constrained expression representation of an SDYMOL system Σ contains a constraint $\kappa_1(l,p)$ for each $p \in P$ and $l \in L$.

A second type of constraint on SDYMOL system behavior pertains to the use and modification of buffer contents during a given process' operation. In order to accurately represent the dependency of message sending and certain iterative and conditional statement executions on the contents of the buffer of a process q, we use the constraint alphabet $S_2(q) = \{ d(q,m), u(q,m) \}_{m \in M}$. The distinguished value $@ \in M$ allows for representing an empty buffer. The constraint corresponding to

$S_2(q)$ is

$$\kappa_2(q) = \Big(\bigvee_{m \in M} d(q,m)u(q,m)^* \Big)^* .$$

As above, interpretation of $\kappa_2(q)$ is facilitated by the associations of figure 4, in which the symbol $d(q,m)$ is identified with the placing of a message of type m into the buffer of process q and the symbol $u(q,m)$ with a use of the buffer in which it is presumed to contain a value of message type m (such as when a particular branch of a conditional is executed or a particular message is sent through an outbound port). The constraint $\kappa_2(q)$ is therefore seen to require that any use of the buffer of q, which presumes that it contains a message of a given type, is preceded by placement of a message of that type into the buffer, with no intervening modifications to the buffer. The constraint does not, however, require that any use of the buffer actually occur between successive modifications of its contents. Thus $\kappa_2(q)$ assures that the buffer of q is used and modified in a consistent fashion, in accordance with the SDYMOL semantics. One such constraint is required for every process in the system.

The third type of constraint relates to system termination. We would like to assure that all behaviors described by a constrained expression representation are complete behaviors. This means that no further system events are possible, because each process has either finished the execution of the sequence of statements that appear in its code or become blocked waiting for a communication that cannot be realized. (A process that executes a **do forever** statement, therefore, must eventually starve.) Constraints are needed in order to assure this because prefixes of the system expression, rather than strings from its language, are used in forming the interpreted language of a constrained expression (for reasons that were explained in §II.4). We use the constraint alphabet $S_3(q) = \{ stop(q), w(p) \}_{p \in P(q)}$, where, as defined in (2.1), $P(q)$ denotes the set of inbound ports of the process q, for

formulating this restriction on the behavior of q. The corresponding constraint,

$$\kappa_3(q) = \Big(\bigvee_{p \in P(q)} w(p) \Big) \vee stop(q),$$

is easily seen to require that q either starves waiting for a communication on one (and only one) of its inbound ports or runs to completion. The constraining context therefore contains a constraint $c_3(q)$ for every $q \in Q$.

The fourth type of constraint simply assures that strings representing behaviors do not contain any non-event (*ne*) symbols. As explained above, it prevents certain illegal patterns of events from occurring. For each $q \in Q$, we use the constraint alphabet, $S_4(q) = \{ ne(q) \}$, and the constraint,

$$\kappa_4(q) = \lambda,$$

to filter out strings containing the non-event symbol associated with q. Because of the manner in which non-event symbols are placed in the derived system expression by the translation rules in figure 5, these constraints eliminate prefixes representing event sequences in which a process exits a **do forever** loop or a process participates in some event after it has starved. The constraint $\kappa_4(p_1)$ in the constraining context for the derived constrained expression representation of the solution to the dining philosophers problem, for example, assures that the prefix of (2.4) is not retained in the set of constrained prefixes.

The fifth type of constraint relates to the transmission of messages in Σ. In any legal SDYMOL system behavior, each reception of a message of type m from a link l must be preceded at some point in the computation by a corresponding placement of a message of that type into that link. Hence, a constraint is needed to assure that at any point in a particular event sequence at least as many messages of type m have been placed in l as there have been messages of type m received from l. We use the constraint alphabet $S_5(l,m) = \{ s(l,m), r(l,p,m) \}_{p \in P}$ in

stating this requirement. The corresponding constraint is

$$\kappa_5(l,m) = s(l,m)^* \,\Delta\, \left(s(l,m)\Big(\bigvee_{p \in P} r(l,p,m) \Big) \right)^{\dagger} .$$

To understand this constraint, notice that the event expression $(ab)^{\dagger}$ represents a set which may be described as "all strings containing equal numbers of a's as b's such that any prefix contains at least as many a's as b's." Thus, as each $s(l,m)$ symbol is associated with the placing of a message of type m into link l and each $r(l,p,m)$ symbol is associated with the receipt of a message of type m from link l, this constraint forces the reception of a message of type m from link l to be preceded by a corresponding placement, although more messages of type m may be placed than are ever received during system operation (allowed by the interleaved $s(l,m)^*$). This, of course, is exactly the intended constraint upon message transmission in behaviors of an SDYMOL system, and so a constraint $\kappa_5(l,m)$ is required for each link l and message type $m \neq \mathbb{Q}$ in Σ.

The final type of constraint pertains to process starvation. According to the SDYMOL semantics, a process can starve waiting for a communication through a port that is connected to a link only if there are no messages that can be received from the link. We state this requirement for a given link l and inbound port p in separate constraints, one for each type of message m that could possibly be received. Thus, for $m \neq \mathbb{Q}$, we define the constraint alphabet $S_6(l,p,m) = \{\, ch(l,p),\ s(l,m),\ r(l,p',m),\ w(p) \,\}_{p' \in P}$ and the corresponding constraint,

$$\kappa_6(l,p,m) = ch(l,p) \left(s(l,m)\Big(\bigvee_{p' \in P} r(l,p',m) \Big) \right)^{\dagger} w(p) \left(s(l,m)\Big(\bigvee_{\substack{p' \in P \\ p' \neq p}} r(l,p',m) \Big) \right)^{\dagger}$$
$$\vee \left(s(l,m) \vee \Big(\bigvee_{p' \in P} r(l,p',m) \Big) \right)^{*} \Delta\, (ch(l,p) \vee w(p) \vee \lambda) .$$

The first alternative of this constraint applies when the link l and port p are connected by a channel (assured by the $ch(l,p)$ symbol) and the process containing the port starves waiting for a communication through it (assured by the $w(p)$ symbol). In such situations the constraint requires that there are no messages of type m to be received from l, both at the time that an attempt to service the receive request is made (assured by the first "daggered subexpression") and also at the end of the behavior, i.e., any messages of type m that are subsequently placed in l are used to service other receive requests (assured by the second daggered subexpression). The second alternative of this constraint applies when there is no channel connecting the link l to the port p or the process containing the port does not starve waiting for a communication through this port (as a $ch(l,p)$ symbol and $w(p)$ symbol cannot both appear). In such situations the constraint permits any pattern of symbols representing events in which messages of type m are placed in l or received from l (provided by the first interleaved subexpression in this alternative), so that no restriction is imposed on the order and number of such events. Taken together, then, the constraints $\kappa_6(l,p,m)$, for all $m \in M$ such that $m \neq @$, assure that if the process containing the port p starves waiting for a communication through this port and if this port is connected to the link l, there is no message of any type that can be received from l, which is precisely the required constraint on SDYMOL system behavior under these conditions. If l and p are not connected by a communication channel, however, or if the process containing p does not starve waiting to receive a message through p, none of these constraints restrict the order or number of messages that are sent to or received from l. The collection of constraints $\kappa_6(l,p,m)$, for $l \in L$, $p \in P$, and $m \neq @$, therefore, accurately express the SDYMOL semantics pertaining to the starvation of processes.

These six types of constraints restrict the order and number of events that occur in strings representing legitimate SDYMOL system behaviors, in accordance with that part of the SDYMOL semantics not expressed by the translation rules

used for deriving the system expression from the design of Σ. The constraints and constraint alphabets required in the derived constrained expression representation of the solution to the dining philosophers problem, for example, are obtained from the expressions for the six types of constraints and constraint alphabets given above using the values for Q, P, L, and M given in (2.2).

Generation of the constraints and constraint alphabets, as described in the preceding paragraphs, completes the derivation of the constrained expression representation for a system Σ from its SDYMOL design. The interpreted language associated with the constrained expression derived in this manner represents all the possible behaviors of Σ. We summarize this construction below.

(2.5) *Definition.* The *derived system expression* ϵ of an SDYMOL system Σ is obtained by taking

$$\epsilon = \iota\Big(\underset{q\in Q}{\triangle} t(\mathbf{code}_q)\Big),$$

where ι denotes the initial expression derived from the design of Σ, as described above, and t is the map that translates the SDYMOL code for a process into a regular expression, as defined in (2.3).

The *derived constraining context*, $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, is obtained by taking

(i) $A = \{\, r(l,p,m),\ s(l,m),\ ch(l,p),\ d(q,m'),\ u(q,m'),\ w(p),\ stop(q),\ ne(q) \,\}$
$\cup F$,

where F is a (possibly empty) set of event symbols, q ranges over Q, l ranges over L, p ranges over P, m and m' range over M, and $m \neq \mathbb{Q}$,

(ii) $E \subseteq A$,

(iii) $\widehat{S} = \{\, S_1(l,p),\ S_2(q),\ S_3(q),\ S_4(q),\ S_5(l,m),\ S_6(l,p,m) \,\}_{q\in Q, l\in L, p\in p, m\neq \mathbb{Q}}$,

where

$$S_1(l,p) = \{\, ch(l,p),\ r(l,p,m) \,\}_{m\neq \mathbb{Q}},$$

$$S_2(q) = \{\, d(q,m),\ u(q,m) \,\}_{m\in M},$$

$$S_3(q) = \{\, stop(q),\ w(p) \,\}_{p\in P(q)},$$

$$S_4(q) = \{\, ne(q) \,\},$$

$$S_5(l,m) = \{\, s(l,m),\ r(l,p,m) \,\}_{p\in P}, \text{ and}$$

$$S_6(l,p,m) = \{\, ch(l,p),\ s(l,m),\ r(l,p',m),\ w(p) \,\}_{p'\in P},$$

and

(iv) $\widehat{C} = \{\, \kappa_1(l,p), \kappa_2(q), \kappa_3(q), \kappa_4(q), \kappa_5(l,m), \kappa_6(l,p,m) \,\}_{q\in Q, l\in L, p\in P, m\neq \Theta}$,

where

$$\kappa_1(l,p) = ch(l,p)\Big(\bigvee_{m\neq\Theta} r(l,p,m)\Big)^* \vee \lambda,$$

$$\kappa_2(q) = \Big(\bigvee_{m\in M} d(q,m)u(q,m)^*\Big)^*,$$

$$\kappa_3(q) = \Big(\bigvee_{p\in P(q)} w(p)\Big) \vee stop(q),$$

$$\kappa_4(q) = \lambda,$$

$$\kappa_5(l,m) = s(l,m)^* \,\Delta\, \left(s(l,m)\Big(\bigvee_{p\in P} r(l,p,m)\Big)\right)^\dagger, \text{ and}$$

$$\kappa_6(l,p,m) = ch(l,p)\left(s(l,m)\Big(\bigvee_{p'\in P} r(l,p',m)\Big)\right)^\dagger w(p)$$

$$\left(s(l,m)\Big(\bigvee_{\substack{p'\in P\\ p'\neq p}} r(l,p',m)\Big)\right)^\dagger$$

$$\vee \left(s(l,m) \vee \Big(\bigvee_{p'\in P} r(l,p',m)\Big)\right)^* \,\Delta\, (ch(l,p) \vee w(p) \vee \lambda).$$

The *derived constrained expression representation* for Σ is defined by $\mathbf{D} = (\mathbf{F}, \epsilon)$, where $\mathbf{F}$ denotes the derived constraining context of Σ and ϵ denotes its derived system expression.

CHAPTER IV

REPRESENTING PETRI NET LANGUAGES AND CSP SYSTEMS

To illustrate the generality of the constrained expression framework, we show how constrained expressions can be generated to represent the behaviors of systems expressed in two notations, Petri net languages and CSP, which are both quite different from SDYMOL and from one another.

§1. Constrained expression representations of Petri net languages

If symbols from an alphabet are associated to some or all the transitions of a Petri net, each firing sequence can be associated with a string over this alphabet. A Petri net language is thus determined by all firing sequences of the Petri net, or by all firing sequences that reach a given final marking [HACK75]. In this section we explain how to derive a constrained expression representation for a Petri net language.

A labeled Petri net, $\mathcal{LP}$, taken from [HACK75, p. 20], is shown in figure 1. The Petri net $\mathcal{LP}$ has two *places*, P_1 and P_2, represented by circles, and three *transitions*, t_1, t_2, and t_3, represented by bars and labeled with symbols from the alphabet $\{a, b, c\}$. A *marking* of a Petri net is signified by putting dots, representing tokens, in places. There is thus a single token in the place P_1 and no tokens in the place P_2 in the initial marking of $\mathcal{LP}$.

The arcs connecting transitions and places in a Petri net determine the rules governing the firing of transitions. A transition t_m is *fireable* if there is a token in the place P_i, for every arc from a place P_i to t_m. When t_m *fires*, a token is removed from the place P_i, for every arc from a place P_i to t_m, and a token is added to the place P_j, for every arc from t_m to a place P_j. In $\mathcal{LP}$, therefore, the

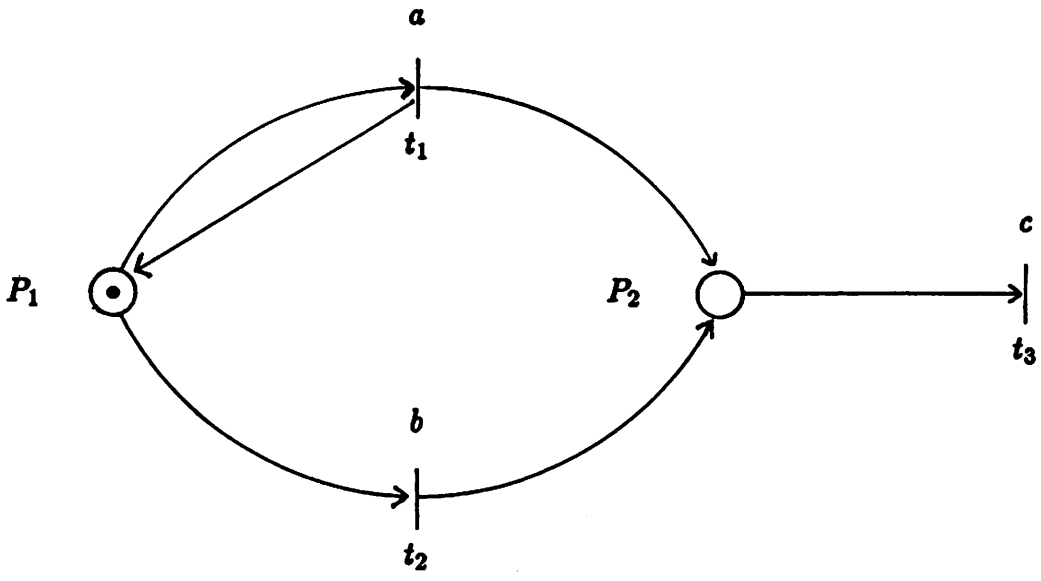


Figure 1

A labeled Petri net

transition t_1 is fireable, and if fired, results in a token being added to the place P_2. (The arc from P_1 to t_1 results in a token being removed from P_1, while the arc from t_1 to P_1 results in a token being placed in P_1, so the net effect is to leave the number of tokens in P_1 unchanged.) The transition t_2 is also fireable. The firing of t_2 results in a token being removed from P_1 and added to P_2. A firing sequence is associated with a string by replacing transitions with their transition labels and erasing transitions that have no labels. The firing sequence $t_1t_1t_2t_3t_3$ in $\mathcal{LP}$ thus determines the string $aabcc$.

When using constrained expressions to express the rules governing the firing of transitions, we require symbols to represent the addition of tokens to places and removal of tokens from places. We also require a special symbol, which we do not associate with any particular event, for reasons that are explained below. We

therefore define the augmented alphabet for a constrained expression representation of a Petri net language to consist of the transition labels in the Petri net, the symbols put_i, signifying the addition of a token to P_i, and $take_i$, signifying the removal of a token from P_i, for each place P_i in the Petri net, and the special symbol $\flat$. The augmented alphabet for a constrained expression representation of the language determined by $\mathcal{LP}$ is thus given by,

$$A = \{a, b, c\} \cup \{put_1, take_1, put_2, take_2\} \cup \{\flat\}.$$

Naturally, the terminal alphabet is just the set of transition labels. In this example, therefore, we have $E = \{a, b, c\}$.

The system expression, ϵ, for the constrained expression that is derived from a Petri net has the form,

$$\epsilon = \iota\left(\bigvee_m \pi_m\right)^* \flat,$$

where m ranges over the indices of the transitions in the Petri net and $\flat$ is the special symbol in the augmented alphabet mentioned above. We refer to ι as the *initial expression* and to the regular expressions π_m as the *transition expressions.*

The initial expression is simply a concatenation of put_i symbols, one for each token in the initial marking that resides in a place P_i. The initial expression thus indicates the initial number of tokens and their locations in the Petri net. The initial expression derived from the labeled Petri net $\mathcal{LP}$ is shown in figure 2.

The transition expressions represent the firing of the transitions in the Petri net. The transition expression π_m, associated with a transition t_m, consists of the concatenation of (i) a sequence of $take_i$ symbols, one for each arc from a place P_i to t_m, (ii) the label associated with the transition, if there is any, and (iii) a sequence of put_j symbols, one for each arc from t_m to a place P_j. The transition expressions encode the semantics of the firing of the transitions. From the transition expression π_1 for the transition t_1 of $\mathcal{LP}$ (see figure 2), for example, it is evident

$$\iota = put_1$$

$$\pi_1 = take_1\, a\, put_1\, put_2$$
$$\pi_2 = take_1\, b\, put_2$$
$$\pi_3 = take_2\, c$$

$$\epsilon = \iota\,(\pi_1 \vee \pi_2 \vee \pi_3)^*$$

Figure 2

Initial expression, transition expressions, and system expression derived from $\mathcal{LP}$

that the firing of t_1 removes a token from place P_1, adds a token to both P_1 and P_2, and is associated with the symbol a. The other transition expressions are similarly interpreted.

The language of the system expression derived from a Petri net represents all conceivable firing sequences of the net. Many of these firing sequences are not possible, however, because they represent sequences in which transitions fire when they are not fireable. A proper prefix of the system expression, furthermore, may end in the middle of a transition, i.e., it may end with a proper prefix of a string from the language of a transition expression. Such a prefix does not encode the full semantics of the firing of this transition as not all the necessary token movements have taken place. The Petri net language constraints filter out proper prefixes of the system expression and strings from the language of the system expression that do not represent legitimate firing sequences.

Two types of constraints are required for this purpose. The first consists of

a single constraint,

$$\kappa_1 = b,$$

defined over the constraint alphabet $S_1 = \{b\}$. Clearly, the constraint κ_1 filters out proper prefixes of the system expression, and hence all strings that end in the middle of a transition.

The form of the second type of constraint is determined by whether the Petri net language consists of the strings associated with all possible firing sequences, or only those that lead to a given final marking. If the Petri net language is determined by all possible firing sequences, then the constraints of the second type need only assure that the rules governing the firing of transitions are followed. According to these rules, a transition is fireable as long as the tokens that must be removed from a place when the transition fires are currently residing in that place. The constraint,

$$\kappa_2(P_i) = put_i^* \,\Delta\, (put_i\, take_i)^\dagger,$$

and constraint alphabet $S_2(P_i) = \{ put_i, take_i \}$, assures that a token is available in a place P_i whenever one is removed. The constraints $\kappa_2(P_i)$, for the places P_i in a labeled Petri net, therefore, assure that constrained prefixes correspond to legal firing sequences.

If a Petri net language is determined by only those firing sequences that lead to a given final marking, then the second type of Petri net language constraint must also assure that the required number of tokens reside in each place at the end of a firing sequence. In this case, therefore, we define the constraints $\kappa_2(P_i)$, for the places P_i in the Petri net, by,

$$\kappa_2(P_i) = (put_i\, take_i)^\dagger \underbrace{put_i \cdots put_i}_{n_i},$$

where $n_i \geq 0$ denotes the number of tokens required in the place P_i by the final marking for the Petri net.

Consider, for example, the Petri net of figure 1. If the final marking for this Petri net requires a single token in the place P_2 and no tokens in the place P_1, the procedure described above produces the system expression and constraints shown in figure 3.

System expression:

$$\epsilon = put_1\,(take_1\,a\,put_1\,put_2 \vee take_1\,b\,put_2 \vee take_2\,c)^*$$

Constraints:

$$\kappa_1 = b$$

$$\kappa_2(P_1) = (put_1\,take_1)^\dagger$$

$$\kappa_2(P_2) = (put_2\,take_2)^\dagger\,put_2$$

Figure 3

System expression and constraints derived from the Petri net of figure 1 given the final marking (0,1)

---

§2. CSP constrained expression representations

In this section we present an example illustrating the derivation of a constrained expression represention for a CSP system. A general procedure for deriving constrained expression representations for arbitrary CSP systems is not presented for two reasons. First, the "definition" of CSP presented in [HOAR78] is not meant to formally define a programming language. Semantic ambiguities, therefore, preclude the formulation of a general procedure for deriving CSP constrained expressions based on the description of CSP provided in that paper. Second, the semantics of CSP are considerably more complex than the semantics of SDYMOL. A procedure for

CSP are considerably more complex than the semantics of SDYMOL. A procedure for deriving CSP constrained expressions is thus correspondingly more complex than a procedure for deriving SDYMOL constrained expressions. The description of such a procedure is beyond the scope of this chapter.

The example below illustrates two important aspects of the use of constrained expressions for representing the possible behaviors of CSP systems. It shows how constrained expressions are used to represent the semantics of CSP (synchronized) communication primitives and also CSP guarded commands, when used in conjunction with repetitive commands. Important aspects not illustrated by this example include the representation of the complete semantics of expression evaluation and of CSP assignment commands. We briefly indicate how these are handled following the example.

The CSP system

We use the CSP system shown in figure 4. It is based on the integer semaphore example presented in [HOAR78]. The reader is assumed to have a basic familiarity with CSP, and so we only briefly discuss the aspects of the CSP semantics that pertain to this example.

The system, BS, implements a binary semaphore, S, shared by two user processes, U_1 and U_2. The processes T_1 and T_2 permit the user processes to (eventually) terminate.

The semaphore process begins by assigning the value 1 to the variable *VAL*. It then executes as many interations as possible of the alternative command in lines S2–5. The process S thus repeatedly waits until one of the user processes is ready to send a v-signal to S (S2–3), or the value of *VAL* is greater than 0 and one of the user processes is ready to send a p-signal to S, at which point the corresponding *guard(s)* is (are) said to *succeed*. It then increments or decrements the value of *VAL*, depending on which of the successful guards is arbitrarily selected

$$BS = [S :: SEM \,\|\, U_1 :: USER_1 \,\|\, U_2 :: USER_2 \,\|\, T_1 :: TERMINATOR_1 \,\|\, T_2 :: TERMINATOR_2]$$

$SEM \equiv$

(S1) VAL **integer**; $VAL := 1$;

(S2) $*[U_1?v() \rightarrow VAL := VAL + 1$

(S3) $\quad \square U_2?v() \rightarrow VAL := VAL + 1$

(S4) $\quad \square VAL > 0; U_1?p() \rightarrow VAL := VAL - 1$

(S5) $\quad \square VAL > 0; U_2?p() \rightarrow VAL := VAL - 1]$

$USER_i \equiv$

(U1) $CONT_i$ **boolean**; $CONT_i := \mathbf{t}$;

(U2) $*[T_i?e() \rightarrow CONT_i := \mathbf{f}$

(U3) $\quad \square CONT_i \rightarrow S!p(); ur_i; S!v()]$

$TERMINATOR_i \equiv U_i!e()$

Figure 4

CSP implementation of a binary semaphore system

for execution. A repetitive command in CSP terminates[1] only if all the guards in the corresponding alternative command fail (i.e., do not succeed) and all the sources named by input commands of guards that would succeed if the appropriate input command could be executed have terminated. The repetitive command in the body of S, therefore, terminates only when both U_1 and U_2 have terminated. If either of the user processes never terminates and if, after some point, none of the guards in the alternative command ever succeed, then S starves. If S starves and the value of VAL is not positive, then S starves waiting for v-signals from the user processes. On the other hand, if S starves and the value of VAL is positive, then S starves waiting for both v-signals and p-signals from the user processes.

The user process U_i, for $i = 1,2$, begins by initializing the boolean variable $CONT_i$. It then repeatedly executes the alternative command in lines U2–3. As long as $CONT_i$ is not altered, the second guard in the alternative command succeeds. If the second guard is selected for execution, the statement corresponding to the guard is then executed. The process U_i thus waits until the semaphore process is ready to receive a p-signal from U_i, uses the resource, an activity that is abstractly represented by the ur_i command, and waits until the semaphore process is ready to receive a v-signal from U_i. The first guard in the alternative command only succeeds when the process T_i is ready to send an e-signal to U_i. If this guard is selected for execution, $CONT_i$ is assigned the value **f**. Thus, the user process U_i repeatedly cycles, sending a p-signal, using the resource and sending a v-signal, until it either starves (waiting for S to become ready to receive either a v-signal or a p-signal) or receives an e-signal from T_i. If U_i receives an e-signal from T_i, the second guard in the alternative command can never again succeed, and so U_i either starves waiting for T_i to become ready to send it an e-signal, or waits for T_i to terminate, after which it also terminates.

---

[1] We use "terminate" here, and in the remainder of this section as in [HOAR78]. Termination of a CSP process corresponds to the "running to completion" of an SDYMOL process, or what is sometimes called "successful termination".

The process T_i, for $i = 1,2$, waits for U_i to become ready to receive an e-signal from T_i, sends the signal and then terminates, or, if U_i never becomes ready to receive the e-signal, starves waiting for the communication to be realized.

Deriving a CSP constrained expression

The event symbols required for a constrained expression representation of $\mathcal{BS}$ are shown in figure 5. As indicated in the first six lines of this figure, event symbols are required to represent the use and modification of variables, interprocess communications (i.e., signals between process), the use of the resource, and the starvation and termination of processes.

The other symbols in this figure are required to represent various aspects of the CSP semantics. For a particular communication, $\alpha = (P, Q, c)$, representing a c-signal from a *source process*, P, to a *target process*, Q, an $s(\alpha)$ symbol indicates that the source process is ready to communicate, while an $s'(\alpha)$ symbol indicates that the communication has been realized and the source process is ready to continue with its sequential activity. These, along with the $r(\alpha)$ symbols, representing the actual communication event, are used in constraints that assure communicating processes are properly synchronized. If P' is either the source or target for the communication α, a $w(P', \alpha)$ symbol encodes information about the starvation event $w(P')$. Specifically, it indicates that the process P' starves waiting for the communication α to occur. These symbols are used in constraints that assure processes do not starve waiting on communications that can be realized. Similarly, the $starve(P)$ and $term(P)$ symbols are used to assure that if the behavior of a process is predicated on the assumption that other processes either starve or have terminated, these processes do in fact starve or have in fact terminated. As when representing an SDYMOL system, the non-event symbols, $ne(P)$, are used to assure that symbols involving a process do not appear after a symbol signifying the starvation of the process has already appeared. The symbols shown in figure 5 thus

Symbol	*Associated Event*
$u(X,a)$	Use of variable X that presumes it to have the value a
$d(X,a)$	Assignment of value a to variable X
$r(P,Q,c)$	c-signal from process P to process Q
ur_i	Use of the resource by process U_i $(i=1,2)$
$w(P)$	Starvation of process P
$stop(P)$	Termination of process P
$s(P,Q,c)$	Process P ready to send c-signal to process Q
$s'(P,Q,c)$	Process P ready to resume execution after sending c-signal to process Q
$w(P,P,Q,c)$	Process P starves waiting for process Q to become ready to receive a c-signal from P
$w(Q,P,Q,c)$	Process Q starves waiting for process P to become ready to send a c-signal to Q
$starve(P)$	Process P is assumed to (eventually) starve
$term(P)$	Process P is assumed to have terminated
$ne(P)$	Non-event symbol for process P

Figure 5

CSP event symbols

comprise the augmented alphabet of the constrained expression representation for the CSP system $\mathcal{BS}$.

The system expression for the constrained expression representation of $\mathcal{BS}$ consists of the interleave of five process expressions:

$$\epsilon = \pi_S \Delta \pi_{U_1} \Delta \pi_{U_2} \Delta \pi_{T_1} \Delta \pi_{T_2}.$$

The process expressions, which are shown in figure 6, represent the sequential activity of the five processes in the system.

The initial $d(VAL, 1)$ symbol in the process expression π_S for the semaphore process is produced by the assignment command S1 in the body of S. Except for the $stop(S)$ symbol, the remainder of the process expression is produced by the repetitive command S2–5. The iterated subexpression of π_S and the enclosing star operator (lines 2–8) represent the repeated execution of the alternative command. Lines 2–5 of π_S represent the possibility that one of the four guards succeeds and is chosen for execution. The interpretation of these lines and the manner in which they are generated is evident, given the interpretations of the event symbols described above. Lines 6 and 7 represent the possibility that S starves waiting for a guard of the alternative command to succeed. Two alternatives are produced because there are two possible truth values for the guard lists preceding the input commands in the guards of the alternative command ($VAL > 0$ has the value t or f). Line 6 represents the possibility that VAL is not positive when S starves. In this case, S starves waiting to receive v-signals from the user processes, represented by the $w(S, U_i, S, v)$ symbols, for $i = 1, 2$. According to the CSP semantics, however, if both U_1 and U_2 terminate, the repetitive command terminates and so one of U_1 or U_2 must also starve, represented by the disjunction of the $starve(U_i)$ symbols, for $i = 1, 2$. Finally, the $w(S)$ symbol signifies the starvation of S and an $ne(S)$ symbol immediately follows the $w(S)$ symbol, since no further symbols from the process expression can appear after a $w(S)$ symbol in a behavioral trace

$\pi_S =$

(1) $d(VAL,1)$

(2) $\Big[r(U_1,S,v) \Big(\bigvee_{j \in Z} u(VAL,j)\, d(VAL,j+1) \Big)$

(3) $\quad \vee\, r(U_2,S,v) \Big(\bigvee_{j \in Z} u(VAL,j)\, d(VAL,j+1) \Big)$

(4) $\quad \vee \Big(\bigvee_{j>0} u(VAL,j) \Big) r(U_1,S,p) \Big(\bigvee_{j \in Z} u(VAL,j)\, d(VAL,j-1) \Big)$

(5) $\quad \vee \Big(\bigvee_{j>0} u(VAL,j) \Big) r(U_2,S,p) \Big(\bigvee_{j \in Z} u(VAL,j)\, d(VAL,j-1) \Big)$

(6) $\quad \vee \Big(\bigvee_{j \leq 0} u(VAL,j) \Big) w(S,U_1,S,v)\, w(S,U_2,S,v) \Big(starve(U_1) \vee starve(U_2) \Big) w(S)\, ne(S)$

(7) $\quad \vee \Big(\bigvee_{j>0} u(VAL,j) \Big) w(S,U_1,S,v)\, w(S,U_2,S,v)\, w(S,U_1,S,p)\, w(S,U_2,S,p)$

(8) $\qquad\qquad \Big(starve(U_1) \vee starve(U_2) \Big) w(S)\, ne(S) \Big]^*$

(9) $\quad term(U_1)\, term(U_2)\, stop(S)$

$\pi_{U_i} =$

(1) $d(CONT_i,\mathbf{t})$

(2) $\Big[r(T_i,U_i,e)\, d(CONT_i,\mathbf{f})$

(3) $\quad \vee\, u(CONT_i,\mathbf{t}) \Big(s(U_i,S,p)\, s'(U_i,S,p) \vee w(U_i,U_i,S,p)\, w(U_i)\, ne(U_i) \Big)$

(4) $\qquad\qquad ur_i \Big(s(U_i,S,v)\, s'(U_i,S,v) \vee w(U_i,U_i,S,v)\, w(U_i)\, ne(U_i) \Big)$

(5) $\quad \vee\, u(CONT_i,\mathbf{f})\, w(U_i,T_i,U_i,e)\, starve(T_i)\, w(U_i)\, ne(U_i) \Big]^*$

(6) $\quad u(CONT_i,\mathbf{f})\, term(T_i)\, stop(U_i)$

$\pi_{T_i} = \Big(s(T_i,U_i,e)\, s'(T_i,U_i,e) \vee w(T_i,T_i,U_i,e)\, w(T_i)\, ne(T_i) \Big) stop(T_i)$

Figure 6

Process expression derived from *BS*

of the system. The interpretation and generation of line 7, which represents the possibility that VAL is positive when S starves, so that S starves waiting for both v-signals and p-signals from the user processes, is similar. The repetitive command terminates only if U_1 and U_2 have terminated. The representation of the repeated execution of the alternative command is thus followed by $term(U_i)$ symbols, for $i = 1, 2$. Finally, the $stop(S)$ symbol is generated to represent the termination of S.

The generation of the process expression π_{U_i}, for $i = 1, 2$, is similar, except that there are two output commands in line U3 of figure 4 that must be translated. Each output command produces a disjunction of two alternatives. One alternative represents the possibility that the communication is realized, and consists of a symbol representing the readiness of U_i to send the appropriate signal, followed by a symbol representing the readiness of U_i, after having sent the signal, to commence execution of the next command in its body. The second alternative represents the possibility that U_i starves waiting for S to become ready to receive the appropriate signal. Since the output command is part of a command list (i.e., does not appear in a guard), we do not assume that S also starves in this case. No $starve(S)$ symbol is thus required in this alternative. (As the same is true of an input command that appears in a command list, input commands are translated differently, according to whether they appear in guards or in command lists.) The interpretation and generation of the process expression π_{T_i}, for $i = 1, 2$, is obvious.

Seven types of constraints are required to appropriately restrict the number and order of symbols in strings representing legal behaviors of a CSP system. For the description of these constraints, let V denote the set of variables in the system, and C denote the set of possible communications between processes, where a communication along the *constructor* c (i.e., a c-signal) with source process P and target process Q is represented by the triple (P, Q, c). For each $X \in V$, let $t(X)$ denote the type of the variable X, and for a communication $\alpha = (P, Q, c)$,

let $source(\alpha) = P$ denote the source process and $target(\alpha) = Q$ denote the target process.

The first type of CSP constraint pertains to the use and modification of variables in a CSP system. For each variable $X \in V$, we require a constraint,

$$\kappa_1(X) = \Big(\bigvee_{v \in \iota(X)} d(X,v)\, u(X,v)^* \Big)^*,$$

over the constraint alphabet $S_1(X) = \{ d(X,v), u(S,v) \}_{v \in \iota(X)}$. Clearly, these constraints assure that the dependency of guarded command execution on the value of a process' variables is accurately represented in the constrained expression representation of a CSP system, in the same way that the corresponding SDYMOL constraints assure that the dependency of iterative and conditional statement executions on the contents of a process' buffer is accurately represented in an SDYMOL constrained expression representation of an SDYMOL system.

The second type of constraint relates to the communication between processes. The CSP semantics require that corresponding input and output commands be synchronized for interprocess communication to take place, where the input command of a process is said to *correspond* to the output command of another process if the input command specifies the second process as the source, the output command specifies the first process as the destination, and, in this simple example, the constructors of the commands are the same. Corresponding input and output commands, when executed, are executed simultaneously. To assure that communication between processes occurs only when processes are ready to execute corresponding commands, we use the constraint,

$$\kappa_2(\alpha) = \Big(s(\alpha)\, r(\alpha)\, s'(\alpha) \Big)^*,$$

defined over the constraint alphabet $S_2(\alpha) = \{ s(\alpha), s'(\alpha), r(\alpha) \}$, for each possible communication $\alpha \in C$. Given the interpretations of the event symbols shown in

figure 5, this constraint is easily understood. For a communication $\alpha = (P, Q, c)$, an $r(\alpha)$ symbol is associated with the simultaneous execution of the corresponding commands, $Q!c()$ in the body of the process P and $P?c()$ in the body of process Q. An $s(\alpha)$ symbol is associated with the readiness of P to send a c-signal to Q and an $s'(\alpha)$ symbol is associated with the readiness of P, after having sent the c-signal, to begin execution of the next command in its body. These later two symbols are used as markers in the process expression π_P for the process P. In every string from the language of π_P, and so in every string representing a potential behavioral trace of P, an $s(\alpha)$ symbol follows all symbols representing events that precede the output event, while an $s'(\alpha)$ symbol precedes all symbols representing events that follow the output event. Besides representing the execution of corresponding input and output commands, an $r(\alpha)$ symbol is used in a similar fashion in the process expression π_Q for the process Q. An $r(\alpha)$ symbol follows symbols representing events that precede the input event and precedes symbols representing events that follow the input event. The constraint $\kappa_2(\alpha)$ is thus seen to assure that the processes P and Q are properly synchronized whenever the communication α takes place.

The third type of CSP constraint relates to system termination. As when representing SDYMOL systems, we require constraints to assure that each process in the system either terminates (i.e., runs to completion) or starves waiting for a communication that cannot be realized. This is easily achieved using the constraints

$$\kappa_3(P) = stop(P) \vee w(P),$$

and corresponding constraint alphabets $S_3(P) = \{ stop(P), w(P) \}$, for each process P in the system.

The fourth type of constraint is also analogous to a type of SDYMOL constraint. The constraints of this type simply assure that strings representing behaviors do not contain any non-event symbols. The constrained expression representation for

a CSP system thus contains constraints,

$$\kappa_4(P) = \lambda,$$

over the constraint alphabets $S_4(P) = \{ ne(P) \}$, for the processes P in the system. Given the description of the system expression presented above, it is evident that these constraints eliminate prefixes representing event sequences in which a process participates in an event after it has starved.

The next two types of CSP constraints relate to the starvation of processes. The first of these assure that processes do not starve waiting on communications that can be realized. For each communication $\alpha \in C$, we define the constraint,

$$\kappa_5(\alpha) = w(source(\alpha), \alpha) \vee w(target(\alpha), \alpha) \vee \lambda,$$

and the constraint alphabet $S_5(\alpha) = \{ w(source(\alpha), \alpha), w(target(\alpha), \alpha) \}$, to express this restriction on behaviors. These constraints filter out prefixes of the system expression that represent event sequences in which both the target and source processes of a communication starve waiting for the communication to be realized.

The second type of CSP constraints relating to the starvation of processes assures that if the behavior of a process is predicated on the assumption that another process starves, then this latter process does indeed starve. For a process P in a CSP system, this is assured by the constraint,

$$\kappa_6(P) = \Big(starve(P)^* \, \Delta \, w(P) \Big) \vee \lambda,$$

and the corresponding constraint alphabet $S_6(P) = \{ starve(P), w(P) \}$. From the description of the generation of the process expressions, we know that, if a string from the language of a process expression π_Q contains a $w(Q)$ symbol that is produced by the translation of a guard in an alternative command, then it also contains a $starve(P)$ symbol, for one of the processes P from which Q is waiting

for a signal. The constraints $\kappa_6(P)$, for the processes P in a CSP system, thus assure that if a process is waiting to execute a guard in a repetitive command and none of the processes upon which this process is waiting for input starve (i.e., they all terminate), then the repetitive command terminates, as required by the CSP semantics.

While the constraints $\kappa_6(P)$, for the processes P in a CSP system, and the manner in which the process expressions are generated assure that the iterated subexpressions of a process expression are not repeated too many times in a constrained prefix, a final type of CSP constraint is required to assure that they are repeated as many times as possible. As explained above, the representation of the repeated execution of an alternative command in the process expression for a CSP process is followed by a sequence of $term(P)$ symbols, for those processes P that must terminate before the repetitive command containing the alternative command terminates. The constraints,

$$\kappa_7(P) = stop(P)\, term(P)^* \vee \lambda,$$

and associated constraint alphabets $S_7(P) = \{ stop(P), term(P) \}$, for the processes P in a CSP system, therefore, assure that repetitive commands do not terminate prematurely.

These seven types of constraints assure that constrained prefixes of the constrained expression representation of a CSP system represent legal behavioral traces of the system. They are summarized in figure 7.

In the system $\mathcal{BS}$, interprocess communication is limited to the exchange of simple timing signals. In a more general CSP system, however, a source process can provide values to be assigned to target variables provided by the target process. This transfer of information between CSP processes is represented in the constrained expression representation of a CSP system in essentially the same way as the tranfer of information between SDYMOL processes is represented in the constrained expres-

$$\kappa_1(X) = \Big(\bigvee_{v \in t(X)} d(X,v)\, u(X,v)^* \Big)^*$$

$$\kappa_2(\alpha) = \Big(s(\alpha)\, r(\alpha)\, s'(\alpha) \Big)^*$$

$$\kappa_3(P) = stop(P) \vee w(P)$$

$$\kappa_4(P) = \lambda$$

$$\kappa_5(\alpha) = w(source(\alpha), \alpha) \vee w(target(\alpha), \alpha) \vee \lambda$$

$$\kappa_6(P) = \Big(starve(P)^* \triangle w(P) \Big) \vee \lambda$$

$$\kappa_7(P) = stop(P) term(P)^* \vee \lambda$$

Figure 7

CSP constraints

sion representation of an SDYMOL system.

The full semantics of CSP expression evaluation and assignment commands have not been modeled in this example. In CSP, the evaluation of an expression and the execution of an assignment command can fail. The evaluation of an expression fails if any of the operations it requires are undefined. The execution of an assignment command fails if the structure of the value to be assigned to a target variable and the structure of the target variable do not *match*, where the notion of matching structures is made precise in [HOAR78]. If the possible structures of the variables in a CSP system are appropriately restricted, the full semantics of these evaluations and commands can be represented, following the example illustrating the representation of the failure of SDYMOL expression evaluations presented in §VIII.2.

Before leaving this example, we comment briefly on the complexity of the derivation procedure illustrated above. The derivation of a CSP system expression is considerably more complex than the derivation of an SDYMOL system expression.

This is primarily due to the complex semantics of the CSP repetitive command. When translating a CSP repetitive command an alternative is produced to represent the execution of each guard and the statement associated with the guard. Additionally, to represent the failure of the repetitive command, the domain of variables in the guard lists of the command are partitioned according to the possible truth value combinations of the guard lists. A distinct alternative is then produced for variables belonging to each set of the partition. Despite this additional complexity, the derivation of a CSP system expression is a fairly straightforward compilation task.

The generation of the CSP constraints is no more complex than the generation of the SDYMOL constraints. Unlike the SDYMOL constraints, furthermore, the CSP constraints are all regular. Because the representation of synchronous communication does not require the dagger operator, this suggests that it may actually be easier to analyze the constrained expression representations of systems in which communication is synchronous, rather than asynchronous.

The use of constrained expressions for representing the behaviors of CSP systems shows that the constrained expression framework can be used with more semantically complex development languages than SDYMOL and with languages that provide synchronized communication primitives.

CHAPTER V

GENERAL THEORY

The previous chapters show how constrained expressions are used to represent the behaviors of distributed systems. We now turn to the analysis of constrained expressions. In this chapter we develop a general theory for manipulating constrained expressions. In subsequent chapters we show how this theory is used for simplifying and analyzing SDYMOL constrained expressions.

We have identified a number of techniques that facilitate the analysis of constrained expressions. One of these involves "simplifying" the constrained expression, or transforming it into a constrained expression that is easier to analyze, but describes the same set of event sequences. This chapter contains the general theoretical framework required for such simplifications. These techniques are applied to SDYMOL constrained expressions in chapter VI, where a "reduction procedure", which is the heart of the simplification process for SDYMOL constrained expressions, is developed. In chapter VII we describe further simplifications that can be performed once an SDYMOL constrained expression is reduced.

Other techniques to facilitate analysis involve "factoring" the constrained expression in ways that allow the analyses of individual "factors" to be combined to give results about the composite expression. By reducing the analysis of the original constrained expression to the analysis of a number of simpler constrained expressions, these techniques essentially modularize the analysis. In this chapter we describe some general results that can be used for simplifying constrained expressions and for modularizing their analysis.

The first section introduces an equivalence relation required for the simplification of constrained expressions and gives a brief overview of the simplification process. Specific results for simplifying constrained expressions are presented in

the second section. The last two sections describe techniques that can be used to modularize the analysis of constrained expressions.

§1. Equivalence of constrained expressions

Since we are ultimately interested in the interpreted language of a constrained expression, it is natural to consider two constrained expressions equivalent if they generate the same interpreted language.

(1.1) *Definition.* Two constrained expressions, $\mathbf{D}$ and $\mathbf{D}'$, are *equivalent*, written $\mathbf{D} \sim \mathbf{D}'$, if $\mathcal{IL}(\mathbf{D}) = \mathcal{IL}(\mathbf{D}')$.

When simplifying a constrained expression, we replace it with progressively simpler, equivalent constrained expressions. This typically involves simplifying the system expression and the constraint set of the original constrained expression, and does not affect the augmented and terminal alphabets. Results for demonstrating the equivalence of constrained expressions, therefore, generally assume the constrained expressions have identical augmented alphabets and identical terminal alphabets.

Suppose two constrained expressions have the same augmented alphabet and the same terminal alphabet. To show that these constrained expressions are equivalent, it is clearly sufficient to show that they have the same constrained prefixes. This sufficient condition is not necessary for equivalence in general, but it is necessary in the case where the terminal alphabet of a constrained expression is the same as the augmented alphabet. We therefore consider a notion of equivalence which is slightly stronger than the one defined above.

(1.2) *Definition.* Two constrained expressions, $\mathbf{D}$ and $\mathbf{D}'$, are *strongly equivalent*, written $\mathbf{D} \approx \mathbf{D}'$, if $A = A'$, $E = E'$, and $\mathcal{P}(\epsilon)|_{\widehat{C}} = \mathcal{P}(\epsilon')|_{\widehat{C}'}$, where $\mathbf{D} = (\mathbf{F}, \epsilon)$, $\mathbf{D}' = (\mathbf{F}', \epsilon')$, $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, and $\mathbf{F}' = (A', E', \widehat{S}', \widehat{C}')$.

Clearly, strong equivalence implies equivalence, so the equivalence relation $\approx$ is at least as fine as the equivalence relation $\sim$.

Because we would like the results in the next section to apply to constrained expressions with identical augmented alphabets and identical terminal alphabets, but without imposing any further restrictions on their terminal alphabets, we demonstrate strong equivalence, rather than equivalence of the appropriate constrained expressions. The equivalence of the constrained expressions then follows.

§2. Simplifying constrained expressions

We simplify a constrained expression in steps, so that a series of progressively simpler constrained expressions is produced, with each constrained expression belonging to the equivalence class of the original constrained expression. In each step, either the current constraining context is replaced with a simpler constraining context or the current system expression is replaced with a simpler system expression. The results for simplifying constrained expressions, therefore, are of two different types: those that produce a simpler constraining context and those that produce a simpler system expression.

Simplifying the constraining context

The constraining context of a constrained expression can often be simplified by eliminating superfluous constraints and constraint alphabets. A constraint that is "implied" by the system expression and other constraints, i.e., that is satisfied by every prefix of the system expression satisfying these other constraints, is superfluous, and so can be eliminated from the constraining context of a constrained expression. Formally, we have

(2.1) Proposition. *Consider the constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$. *Let* $\kappa_j \in \widehat{C}$ *be a constraint over the constraint alphabet* $S_j \in \widehat{S}$, *and* $\widehat{B} \subseteq \widehat{C}$ *be a set of constraints, with* $\kappa_j \notin \widehat{B}$. *If* $\rho_{S_j}\left(P(\epsilon)|_{\widehat{B}}\right) \subseteq \mathcal{L}(\kappa_j)$ *then* $\mathbf{D} \approx (\mathbf{F}', \epsilon)$, *where* $\mathbf{F}' = (A, E, \widehat{S}', \widehat{C}')$, $\widehat{S}' = \widehat{S} - \{S_j\}$ *and* $\widehat{C}' = \widehat{C} - \{\kappa_j\}$.

A constraint that cannot be eliminated from the constraining context of a constrained expression can often be replaced with a simpler event expression. The following proposition gives conditions under which an event expression can be used to replace a constraint in the constraining context of a constrained expression.

(2.2) **Proposition.** *Consider the constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$. *Let* $\kappa_j \in \widehat{C}$ *be a constraint over the constraint alphabet* $S_j \in \widehat{S}$, *and let* $\kappa'_j \in \mathcal{EE}(S_j)$. *If* $\mathcal{L}(\kappa'_j) \subseteq \mathcal{L}(\kappa_j)$ *and every constrained prefix of* $\mathbf{D}$ *satisfies* κ'_j, *when viewed as a constraint over the constraint alphabet* S_j, *then* $\mathbf{D} \approx (\mathbf{F}', \epsilon)$, *where* $F' = (A, E, \widehat{S}, \widehat{C}')$, *and* $\widehat{C}'$ *is obtained from* $\widehat{C}$ *by replacing the constraint* κ_j *in* $\widehat{C}$ *with* κ'_j.

The next proposition shows that, under certain conditions, both a constraint and its constraint alphabet can be simplified.

(2.3) **Proposition.** *Consider the constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, *and a constraint* $\kappa_j \in \widehat{C}$, *with corresponding constraint alphabet* $S_j \in \widehat{S}$. *Let* $S'_j \subseteq S_j$, *and suppose that* $\kappa'_j \in \mathcal{EE}(S'_j)$ *satisfies* $\rho_{S'_j}(\mathcal{L}(\kappa_j)) \subseteq \mathcal{L}(\kappa'_j)$. *If there is some set of constraints* $\widehat{B} \subseteq \widehat{C}$ *such that* $\kappa_j \notin \widehat{B}$ *and every string in* $P(\epsilon)|_{\widehat{B} \cup \{\kappa'_j\}}$ *satisfies the constraint* κ_j *(i.e.,* $\rho_{S_j}(P(\epsilon)|_{\widehat{B} \cup \{\kappa'_j\}}) \subseteq \mathcal{L}(\kappa_j)$, *when* S'_j *is taken as the constraint alphabet coresponding to* κ'_j*), then* $\mathbf{D} \approx (\mathbf{F}', \epsilon)$, *where* $\mathbf{F}' = (A, E, \widehat{S}', \widehat{C}')$, $\widehat{S}'$ *is obtained from* $\widehat{S}$ *by replacing* S_j *with* S'_j, *and* $\widehat{C}'$ *is obtained from* $\widehat{C}$ *by replacing* κ_j *with* κ'_j.

Proof. $P(\epsilon)|_{\widehat{C}} \subseteq P(\epsilon)|_{\widehat{C}'}$ follows easily from the definition of $\widehat{C}'$ and $\widehat{S}'$ and the fact that $\rho_{S'_j}(\mathcal{L}(\kappa_j)) \subseteq \mathcal{L}(\kappa'_j)$, which implies that any string satisfying κ_j also satisfies κ'_j.

The reverse inclusion is established just as easily. Clearly, $\widehat{B} \cup \{\kappa'_j\} \subseteq \widehat{C}'$ implies $P(\epsilon)|_{\widehat{C}'} \subseteq P(\epsilon)|_{\widehat{B} \cup \{\kappa'_j\}}$. As every string in $P(\epsilon)|_{\widehat{B} \cup \{\kappa'_j\}}$ satisfies the constraint κ_j by hypothesis, this shows that every string in $P(\epsilon)|_{\widehat{C}'}$ satisfies κ_j, and so we have $P(\epsilon)|_{\widehat{C}'} \subseteq P(\epsilon)|_{\widehat{C}}$. ∎

Simplifying the system expression

As shown in chapters III and IV, the system expression in a constrained expression representation for a system consisting of numerous sequential processes executing concurrently is typically expressed as an initial expression, representing the initial "configuration" of the system, concatenated with the interleave of process expressions, representing the sequential activities of the processes in the system. The following propositions permit the simplification of the process expressions in system expressions of this form.

(2.4) **Proposition.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be a constrained expression, where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota(\eta \Delta \pi)$, and ι *is a concatenation of event symbols, and consider a regular expression* $\pi' \in \mathcal{RE}(A)$, *and a set of constraints* $\widehat{D} \subseteq \widehat{C}$. *If*

(i) $\kappa_j \in \widehat{D}$ *implies* $\rho_{S_j}(\eta) = \lambda$, *where* S_j *denotes the constraint alphabet corresponding to* κ_j, *and*

(ii) $P(\iota\pi)|_{\widehat{D}} = P(\iota\pi')|_{\widehat{D}}$,

then $\mathbf{D} \approx (\mathbf{F}, \epsilon')$, *where* $\epsilon' = \iota(\eta \Delta \pi')$.

Proof. To show $P(\epsilon)|_{\widehat{C}} \subseteq P(\epsilon')|_{\widehat{C}}$, we choose a string $w \in P(\epsilon) = P\left(\iota(\eta \Delta \pi)\right)$ satisfying the constraints of $\widehat{C}$. If $w \in P(\iota)$, then the desired inclusion is established trivially. Otherwise, $w = \iota u_1 v_1 u_2 v_2 \cdots u_n v_n$, for some $n \geq 1$ and strings $u_i, v_i \in A^*$, for $1 \leq i \leq n$, such that $u_1 u_2 \cdots u_n \in P(\pi)$ and $v_1 v_2 \cdots v_n \in P(\eta)$.

Now, if $\kappa_j \in \widehat{D}$, then using hypothesis (i) of the proposition we reason that $\rho_{S_j}(w) = \rho_{S_j}(\iota u_1 u_2 \cdots u_n)$, where S_j denotes the constraint alphabet corresponding to the constraint κ_j. Since the string w satisfies the constraints of $\widehat{D}$, this implies that $\iota u_1 u_2 \cdots u_n \in P(\iota\pi)|_{\widehat{D}} = P(\iota\pi')|_{\widehat{D}}$, and so that $u_1 u_2 \cdots u_n \in P(\pi')$. We therefore conclude that $w \in P\left(\iota(\eta \Delta \pi')\right)$ and, as w satisfies the constraints of $\widehat{C}$ by hypothesis, that $w \in P(\epsilon')|_{\widehat{C}}$, as desired.

The reverse inclusion is established in precisely the same fashion. ∎

(2.5) **Corollary.** *Hypothesis (ii) of proposition (2.4) can be replaced with the hypothesis*

(ii) $P\left(\rho_S(\iota)\pi\right)\big|_{\widehat{D}} = P\left(\rho_S(\iota)\pi'\right)\big|_{\widehat{D}}$, *where* S *is the union of the constraint alphabets corresponding to the constraints of* $\widehat{D}$.

When a constrained expression is reduced, (2.4) and (2.5) take the following simpler forms.

(2.6) **Proposition.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be a reduced constrained expression, where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota(\eta \,\Delta\, \pi)$, *and* ι *is a concatenation of event symbols, and consider a regular expression* $\pi' \in \mathcal{RE}(A)$, *and a set of constraints* $\widehat{D} \subseteq \widehat{C}$. *If*

(i) $\kappa_j \in \widehat{D}$ *implies* $\rho_{S_j}(\eta) = \lambda$, *where* S_j *denotes the constraint alphabet corresponding to* κ_j,

(ii) $\mathcal{L}(\iota\pi)|_{\widehat{D}} = \mathcal{L}(\iota\pi')|_{\widehat{D}}$, *and*

(iii) the constrained expression $(\mathbf{F}, \epsilon')$ *is reduced, where* $\epsilon' = \iota(\eta \,\Delta\, \pi')$,

then $\mathbf{D} \approx (\mathbf{F}, \epsilon')$.

(2.7) **Corollary.** *Hypothesis (ii) of proposition (2.6) can be replaced with the hypothesis*

(ii) $\mathcal{L}\left(\rho_S(\iota)\pi\right)\big|_{\widehat{D}} = \mathcal{L}\left(\rho_S(\iota)\pi'\right)\big|_{\widehat{D}}$, *where* S *is the union of the constraint alphabets corresponding to the constraints of* $\widehat{D}$.

§3. Projection on constraint alphabets

One technique for modularizing the analysis of a constrained expression involves focusing on a small number of constraints and determining how these constraints restrict the order and number of event symbols from the associated constraint alphabets that appear in constrained prefixes of the constrained expression. (We consider constrained prefixes rather than strings from the interpreted language

so as not to have to impose any restrictions on the terminal alphabet of the constrained expression.) The idea behind this approach is as follows. Suppose the constraints in some subset, say $\hat{D}$, of the set of constraints of a constrained expression use symbols from a subset S of the augmented alphabet, but do not constrain any other symbols in the augmented alphabet. Then symbols from the set S are the only symbols in the system expression that are of interest when reasoning about how the constraints of $\hat{D}$ interact to restrict the order and number of symbols that appear in constrained prefixes of the constrained expression. It should be possible, therefore, to characterize the patterns of symbols from S that appear in constrained prefixes of the original constrained expression by projecting the system expression on the alphabet S and characterizing the patterns of symbols from S that appear in prefixes of the projected system expression satisfying the constraints of $\hat{D}$.

The subset S would presumably be chosen because some property of the modeled system can be stated in terms of the patterns of symbols from S that appear in legal behavioral traces of the system, while the set $\hat{D}$ would be chosen because the designer expects that the constraints of $\hat{D}$ assure the system exhibits the desired property. For example, to demonstrate the mutually exclusive use of the messages residing in a link l in an SDYMOL system, we would take S to be the set of event symbols representing the transmission of messages to l and the reception of messages from l, and $\hat{D}$ to be the set of constraints that relate to the flow of messages through the link l. Projecting the system expression on S focuses the analysis on the symbols required to express the property, while restricting the constraint set to $\hat{D}$ focuses the analysis on the constraints required to establish the property. Proving a property of constrained prefixes of a constrained expression is thus reduced to proving the same property of constrained prefixes of a simpler constrained expression, namely, one whose system expression is the projection of the original system expression and whose constraints are given by the subset $\hat{D}$.

A formal basis for this reasoning is provided by the following easy proposition.

(3.1) **Proposition.** *Consider the constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \hat{S}, \hat{C})$, *and subsets* $S \subseteq A$ *and* $\hat{D} \subseteq \hat{C}$. *If the alphabet* S *contains the constraint alphabet corresponding to each constraint in* $\hat{D}$, *then*

$$\rho_S\left(P(\epsilon)|_{\hat{C}}\right) \subseteq P\left(\rho_S(\epsilon)\right)\Big|_{\hat{D}}.$$

This proposition implies that any restrictions imposed by the constraints of $\hat{D}$ on the order and number of symbols from S that appear in prefixes of the projected system expression are also imposed on the order and number of symbols from S that appear in constrained prefixes of the original constrained expression. For example, to demonstrate that a process in an SDYMOL system doesn't starve waiting for a communication through a particular port p, we might take $\hat{D}$ equal to the set of constraints $\kappa_6(l, p, m)$, for $l \in F(p)$ and $m \neq @$, which relate to the starvation event $w(p)$, and S equal to the union of the constraint alphabets of the constraints in $\hat{D}$. If we could demonstrate that no string in $P\left(\rho_S(\epsilon)\right)\Big|_{\hat{D}}$ contains a $w(p)$ symbol, where ϵ denotes the system expression of the constrained expression representation of the system, we could then conclude that no constrained prefix of this constrained expression representation contains a $w(p)$ symbol, i.e., that the process cannot starve waiting for a communication through p. (Of course, these constraints may not be sufficient to imply that no constrained prefix contains a $w(p)$ symbol. In such cases it may be necessary to add certain other constraints to the set $\hat{D}$.) Various examples of the use of this proposition are presented in chapter VIII.

When analyzing the constrained expression representation of an SDYMOL system we usually obtain a reduced constrained expression representation for the system before focusing on the interaction of specific constraints in this manner. This is because, as shown in chapter VI, a reduced constrained expression representation for an SDYMOL system is easily obtained from its derived constrained expression representation (without unnecessarily complicating the representation). In this case we

use the following corollary of (3.1).

(3.2) **Proposition.** *Consider the constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \hat{S}, \hat{C})$, *and subsets* $S \subseteq A$ *and* $\hat{D} \subseteq \hat{C}$. *If the alphabet* S *contains the constraint alphabet corresponding to each constraint in* $\hat{D}$, *then*

$$\rho_S\left(\mathcal{L}(\epsilon)|_{\hat{C}}\right) \subseteq \mathcal{L}\left(\rho_S(\epsilon)\right)\Big|_{\hat{D}}.$$

§4. Focusing on "subsystems"

A different technique for modularizing the analysis of a constrained expression is suggested by the following scenario. Suppose we have a system Σ that is described using some formal design notation and we have analyzed a constrained expression representation, $\mathbf{D}$, of Σ. If we expand Σ, perhaps introducing new processes or adding detail to processes that already exist, we obtain a new system Σ' that contains the old system "embedded" within it (the system Σ can be viewed as a "subsystem" of Σ').[1] If this embedding is nice enough, many of the properties of the behavior of Σ which were revealed by analysis of $\mathbf{D}$ should describe the behavior of the image of Σ in Σ' (i.e., of the subsystem of Σ' corresponding to Σ). If $\mathbf{D}'$ is the constrained expression representation of Σ', furthermore, relationships between the constrained expressions $\mathbf{D}$ and $\mathbf{D}'$ should indicate when properties of the behavior of Σ can be inferred to hold for the behavior of the subsystem of Σ' corresponding to Σ.

Intuitively, if a system Σ is a subsystem of a larger system Σ', every event in Σ is also an event in Σ', and so every event symbol in the augmented alphabet

[1] We purposely leave the notions of "subsystem" and "embedding" informal. We use these notions only to indicate how we were led to consider the proposition presented in this section and why it is useful. The proposition, however, is stated in terms of constrained expressions, independent of any systems that the constrained expressions might represent.

of a constrained expression representation for Σ also appears in the augmented alphabet of a constrained expression representation for Σ'. Thus, if $\mathbf{D} = (\mathbf{F}, \epsilon)$, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, is a constrained expression representation for Σ, and $\mathbf{D}' = (\mathbf{F}', \epsilon')$, where $\mathbf{F}' = (A', E', \widehat{S}', \widehat{C}')$ is a constrained expression representation for Σ', we have $A \subseteq A'$.

Using the proposition below, conditions can be identified that imply every constrained prefix of $\mathbf{D}'$, when projected on the alphabet A, is a constrained prefix of $\mathbf{D}$, and that every string from the constrained language of $\mathbf{D}'$, when projected on the alphabet A, belongs to the constrained language of $\mathbf{D}$. In terms of the subsystem Σ and the system Σ', the first of these results assures that if the subsystem Σ does not exhibit a certain behavior when viewed as a separate system, then it does not exhibit the behavior when viewed as a subsystem of Σ'. The second of these results is useful when working with reduced constrained expressions, since the constrained language of a reduced constrained expression is identical to its set of constrained prefixes. (As usual, we consider the sets of constrained prefixes and the constrained languages of the constrained expressions, rather than the interpreted languages, so that our results apply to constrained expressions when no restrictions are imposed on their terminal alphabets.)

(4.1) **Proposition.** *Consider the constrained expressions* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *and* $\mathbf{D}' = (\mathbf{F}', \epsilon')$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\mathbf{F}' = (A', E', \widehat{S}', \widehat{C}')$, *and* $A \subseteq A'$. *Let* $\widehat{D} \subseteq \widehat{C}$ *and* $B \subseteq A$. *If*

(i) $\mathcal{L}\left(\rho_B(\epsilon')\right) \subseteq \mathcal{L}\left(\rho_B(\epsilon)\right)$, *and*

(ii) for every $c_j \in \widehat{D}$, *there is some* $c_i' \in \widehat{C}'$ *such that* $S_j = S_i' \cap B$ *and* $\mathcal{L}\left(\rho_{S_j}(c_i')\right) \subseteq \mathcal{L}(c_j)$, *where* S_j *and* S_i' *denote the constraint alphabets of* c_j *and* c_i', *respectively,*

then $\rho_B\left(P(\epsilon')|_{\widehat{C}'}\right) \subseteq P\left(\rho_B(\epsilon)\right)\Big|_{\widehat{D}}$ *and* $\rho_B\left(\mathcal{L}(\epsilon')|_{\widehat{C}'}\right) \subseteq \mathcal{L}\left(\rho_B(\epsilon)\right)\Big|_{\widehat{D}}$.

Proof. To prove that $\rho_B\left(P(\epsilon')|_{\widehat{C}'}\right) \subseteq P\left(\rho_B(\epsilon)\right)\Big|_{\widehat{D}}$, we choose a string $u' \in$

$P(\epsilon')|_{\widehat{C}'}$ and show that $\rho_B(u') \in P\left(\rho_B(\epsilon)\right)\Big|_{\widehat{D}}$. Clearly, $\rho_B(u') \in \rho_B\left(P(\epsilon')\right) = P\left(\rho_B(\epsilon')\right) \subseteq P\left(\rho_B(\epsilon)\right)$ follows from hypothesis (i) and the assumption that $u' \in P(\epsilon')$. To see that $\rho_B(u')$ satisfies the constraints of $\widehat{D}$, choose a constraint $c_j \in \widehat{D}$ and let S_j denote its constraint alphabet. Then choose the constraint $c'_i \in \widehat{C}'$, with corresponding constraint alphabet S'_i, assured by hypothesis (ii). Now, since u' satisfies the constraints of $\widehat{C}'$, we have $\rho_{S'_i}(u') \in \mathcal{L}(c'_i)$. Because of this and hypothesis (ii) we reason that $\rho_{S_j}\left(\rho_{S'_i}(u')\right) \in \rho_{S_j}\left(\mathcal{L}(c'_i)\right) = \mathcal{L}\left(\rho_{S_j}(c'_i)\right) \subseteq \mathcal{L}(c_j)$. But $\rho_{S_j} \circ \rho_{S'_i} = \rho_{S_j} = \rho_{S_j} \circ \rho_B$, and so we conclude that $\rho_{S_j}\left(\rho_B(u')\right) \in \mathcal{L}(c_j)$, as desired. The proof that $\rho_B\left(\mathcal{L}(\epsilon')|_{\widehat{C}'}\right) \subseteq \mathcal{L}\left(\rho_B(\epsilon)\right)\Big|_{\widehat{D}}$ is similar. ∎

Taking $B = A$ and $\widehat{D} = \widehat{C}$ in the above proposition, we observe that if (i) every pattern of symbols from A that appears in a prefix of ϵ' appears in some prefix of ϵ, and (ii) the symbols of A are restricted at least as strongly by the constraints of $\widehat{C}'$ as they are by the constraints of $\widehat{C}$, then every constrained prefix of $\mathbf{D}'$, when projected on the alphabet A, is a constrained prefix of $\mathbf{D}$, and that every string from the constrained language of $\mathbf{D}'$, when projected on the alphabet A, belongs to the constrained language of $\mathbf{D}$. The use of proposition (4.1) is illustrated in chapter VIII.

CHAPTER VI

REDUCED SDYMOL CONSTRAINED EXPRESSIONS

Having shown how to derive a constrained expression representation of a system expressed in SDYMOL and having developed some theory for simplifying general constrained expressions, we now show how this theory can be applied to transform the derived constrained expression representation of an SDYMOL system into an equivalent reduced constrained expression. As explained in chapter II, this means that every prefix of the system expression satisfying the constraints is a complete string in the language of the system expression. When analyzing this constrained expression representation of the SDYMOL system, therefore, we need only consider complete strings of the system expression, making the analysis much easier. In the process of doing this reduction, certain alternatives in the derived system expression are eliminated, a number of constraints in the derived constraining context are eliminated, and other constraints are simplified.

In the first section of this chapter, we describe some standard simplifications that are performed before the constrained expression representation of an SDYMOL system is reduced. These simplifications are routinely applied to the derived constrained expression representation of every SDYMOL system. In the second section we describe some additional simplifications that can be performed, if necessary, to keep the system expression from becoming too large when reducing the constrained expression representation. These simplifications are optional, and they are often easier to perform once a reduced constrained expression representation for the system has been obtained. Finally, in the third section we describe an algorithm for reducing any appropriately simplified constrained expression representation of an SDYMOL system.

§1. Preliminary simplifications

Although it is not necessary to simplify the derived constrained expression representation of a system before reducing it, we perform certain standard simplifications before reducing a representation to facilitate the eventual reduction of the representation. It is just as easy (and in some cases, easier) to perform these preliminary simplifications before reducing the constrained expression representation as it is to perform them after. The simplified constrained expression representation of the system, furthermore, is easier to reduce and results in a simpler reduced constrained expression. These simplifications are thus routinely applied to the derived constrained expression representation of an SDYMOL system before the representation is reduced. They involve eliminating "impossible" alternatives from the derived system expression, eliminating unnecessary constraints and constraint alphabets from the derived constraining context, and simplifying other constraints. For convenience, we describe the application of these simplifications in a procedural fashion. In general, however, the order in which particular simplifications are performed is not significant.

For the discussion in this section, let the constrained expression $\mathbf{D} = (\mathbf{F}, \epsilon)$, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, denote the derived constrained expression representation of an SDYMOL system Σ, as defined in (III.2.5). We refer to the components of Σ using the notation established in (III.2.1) throughout this chapter.

Simplification of the derived system expression is possible because restrictions imposed on the flow of messages within a system by the absence of channels connecting specific links and ports are not reflected in the rules for deriving the system expression. When generating the system expression for the derived constrained expression representation of Σ, a **receive** p statement in the SDYMOL code for a process q produces alternatives associated with each link in Σ to represent the possibility that execution of the statement results in a message being transmitted

from the associated link through the designated port (provided by the disjunction over all message types $m \neq \Omega$ and links $l \in L$ in T_2 of figure III.5). If there is no channel connecting a particular link l to the designated port p, however, the SDYMOL semantics preclude the reception of messages from l through p, and so alternatives generated by a **receive** p statement that contain symbols representing such receptions are impossible (i.e., if a prefix of the derived system expression contains strings from any such alternatives, it is filtered out of the set of constrained prefixes by the constraints in $\hat{C}$). These alternatives can therefore be eliminated. If Σ is a realistically large system and if the interconnectivities between the processes comprising Σ are kept to a minimum, as one would expect in a good, modular design, this can significantly reduce the number of impossible alternatives in the derived system expression of Σ. To formalize this argument we require the following definition.

(1.1) *Definition.* For $p \in P$, we define

$$F(p) = \{ l \in L \mid l \text{ and } p \text{ are connected by a channel} \},$$

so that $F(p)$ represents the links that messages received through p can come from. We then define t' to be the map that associates the SDYMOL code for a process q with a regular expression in the same manner as the map t, which was defined in (III.2.3) using the SDYMOL translations rules shown in figure III.5, except that the translation rule,

$$\textbf{receive } p \quad \rightarrow \quad \Big(\bigvee_{\substack{l \in F(p) \\ m \neq \Omega}} r(l,p,m)\, d(q,m) \Big) \vee w(p)\, ne(q),$$

is used in place of the translation rule T_2. Finally, we define a simplified system expression $\epsilon' \in \mathcal{RE}(A)$ by

$$\epsilon' = \iota\Big(\triangle_{q \in Q} t'(\textbf{code}_q) \Big),$$

where ι is the initial expression derived from the SDYMOL design of Σ, as described in section (III.2).

(1.2) Proposition. $\mathbf{D} \approx (\mathbf{F}, \epsilon')$, *where* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *is the derived constrained expression representation of* Σ *and* ϵ' *is defined in (1.1).*

Proof. For each $q \in Q$, we apply proposition (V.2.4), to show that the process expression $t(\mathbf{code}_q)$ can be replaced with the regular expression $t'(\mathbf{code}_q)$.

To apply proposition (V.2.4), for a given $q \in Q$, we take $\pi = t(\mathbf{code}_q)$, η equal to the interleave of the process expressions of all the processes except the process q, $\pi' = t'(\mathbf{code}_q)$, and $\hat{D} = \{\kappa_1(l,p)\}_{p \in P(q), l \in L}$. Hypothesis (i) of proposition (V.2.4) is clearly satisfied. To see that hypothesis (ii) is also satisfied, note that $t'(\mathbf{code}_q)$ is obtained from $t(\mathbf{code}_q)$ by replacing every subexpression of the form $\left(\bigvee_{\substack{l \in L \\ m \neq \mathbb{Q}}} r(l,p,m)\, d(q,m) \right) \vee w(p)\, ne(q)$ with the regular expression $\left(\bigvee_{\substack{l \in F(p) \\ m \neq \mathbb{Q}}} r(l,p,m)\, d(q,m) \right) \vee w(p)\, ne(q)$, for every $p \in P(q)$. Using proposition (II.2.5), therefore, we conclude that every prefix of the regular expression $\iota\, t(\mathbf{code}_q)$ that is not also a prefix of the regular expression $\iota\, t'(\mathbf{code}_q)$ contains an $r(l,p,m)$ symbol, for some $p \in P(q)$, $l \notin F(p)$, and $m \neq \mathbb{Q}$. If $l \notin F(p)$, however, the initial expression ι does not contain a $ch(l,p)$ symbol. Such prefixes, therefore, violate a constraint $\kappa_1(l,p)$, for some $p \in P(q)$ and $l \notin F(p)$. Hence $\mathcal{P}(\iota\, t(\mathbf{code}_q))|_{\hat{D}} \subseteq \mathcal{P}(\iota\, t'(\mathbf{code}_q))|_{\hat{D}}$. The reverse inclusion is established by noting that $t'(\mathbf{code}_q) \subseteq t(\mathbf{code}_q)$. The hypotheses of proposition (V.2.4) are satisfied, therefore, and so the process expression $t(\mathbf{code}_q)$ can be replaced with the process expression $t'(\mathbf{code}_q)$, as desired. ∎

The derived system expression is thus simplified by replacing the process expression $t(\mathbf{code}_q)$, with the process expression $t'(\mathbf{code}_q)$, for each $q \in Q$. Once this has been done, certain constraints in the derived constraining context can be eliminated, while others can be simplified, as we show below.

For convenience, the results for simplifying the derived constraining context are stated in terms of eliminating constraints and constraint alphabets from the constraining context of a given constrained expression representation of the system or replacing constraints and constraint alphabets with simpler ones. More formally, what we mean by this is that the constrained expression representation produced when the constraining context is so altered is strongly equivalent to the given constrained expression representation of the system.

We begin simplifying the constraining context $\mathbf{F}$ by eliminating the constraints $\kappa_1(l,p)$, for $l \in L$ and $p \in P$, which assure that messages are received from a link l through a port p only if l and p are connected to one another by a channel. These constraints are not needed when the simplified system expression defined in (1.1) is used in place of the derived system expression, as the simplified system expression does not contain any symbols representing the reception of messages between links and ports that are not connected by a channel. This reasoning results in the following proposition.

(1.3) **Proposition.** *For every $l \in L$ and $p \in P$, the constraint $\kappa_1(l,p)$ and the corresponding constraint alphabet $S_1(l,p)$ can be eliminated from the constraining context of the constrained expression $(\mathbf{F}, \epsilon')$, where $\mathbf{F}$ is the derived constraining context of Σ and ϵ' is the simplified system expression defined in (1.1).*

Proof. For a given $p \in P$ and $l \in L$, we first observe that every prefix of the simplified system expression $\epsilon' = \iota\Big(\underset{q\in Q}{\triangle}\, t'(\mathbf{code}_q)\Big)$ satisfying the constraints in the set $\widehat{C}_3 = \{\kappa_3(q)\}_{q\in Q}$, where $\kappa_3(q) = stop(q) \vee \Big(\underset{p'\in P(q)}{\bigvee} w(p')\Big)$, also satisfies the constraint $\kappa_1(l,p) = ch(l,p)\Big(\underset{m\neq \mathbf{Q}}{\bigvee} r(l,p,m)\Big)^* \vee \lambda$. If there is no channel connecting l to p, neither the initial expression ι nor the process expressions $t'(\mathbf{code}_q)$, for $q \in Q$, contain any $ch(l,p)$ or $r(l,p,m)$ symbols, for $m \neq \mathbf{Q}$, and so every prefix of ϵ' satisfies $\kappa_1(l,p)$. On the other hand, if there is a channel connecting l to p,

then ι contains a single $ch(l,p)$ symbol and no $r(l,p,m)$ symbols, for $m \neq \mathbb{Q}$, while $t'(\mathbf{code}_q)$ does not contain any $ch(l,p)$ symbols and, if $p \in P(q)$, may contain any number of $r(l,p,m)$ symbols. If a prefix of ϵ' satisfies the constraints of $\hat{C}_3$, furthermore, it contains the string represented by ι as an initial substring. In this case, therefore, the projection on $S_1(l,p)$ of a prefix of ϵ' satisfying the constraints of $\hat{C}_3$ lies in the language of $ch(l,p)\left(\bigvee_{m\neq\mathbb{Q}} r(l,p,m)\right)^*$, and so every such prefix satisfies $\kappa_1(l,p)$.

For each $p \in P$ and $l \in L$, we then apply proposition (V.2.1), with $\hat{B} = \hat{C}_3$, to give the desired result. ∎

Once the constraints $\kappa_1(l,p)$, for $l \in L$ and $p \in P$, have been eliminated from the constraining context derived from the design of Σ, we proceed to simplify the constraints $\kappa_5(l,m)$, for all links $l \in L$ and message types $m \neq \mathbb{Q}$.

The constraint $\kappa_5(l,m)$, for a given link l and message type $m \neq \mathbb{Q}$, assures that every constrained prefix of $\mathbf{D}$ represents an event sequence in which there is always a message of type m available in link l at any point when such a message is received from link l. It assures this by restricting the order and number of $s(l,m)$ and $r(l,p,m)$ symbols, for all ports $p \in P$, in prefixes of the system expression that are retained as constrained prefixes. If a link l is not connected to a particular port p, however, the simplified system expression defined in (1.1) does not contain any $r(l,p,m)$ symbols corresponding to this link and port. When this simplified system expression is used in place of the derived system expression, therefore, the constraint $\kappa_5(l,m)$ need only restrict the order and number of $s(l,m)$ and $r(l,p,m)$ symbols, for those ports p that are connected to l by a channel, rather than all ports $p \in P$. If a port is not connected to the link l, therefore, the $r(l,p,m)$ symbol in $\kappa_5(l,m)$ corresponding to this particular port p can be eliminated. This reasoning results in a proposition for simplifying the constraints $\kappa_5(l,m)$, for all links $l \in L$ and message types $m \neq \mathbb{Q}$. The following definition is used in the statement of this proposition.

(1.4) *Definition.* For $l \in L$ we define

$$T(l) = \{p \in P \mid l \text{ and } p \text{ are connected by a channel}\},$$

so that $T(l)$ represents the ports that messages residing in link l can be received through.

(1.5) **Proposition.** *For every link $l \in L$ and message type $m \neq \Omega$, the constraint $\kappa_5(l,m)$ can be replaced by the expression,*

$$\kappa_5'(l,m) = s(l,m)^* \Delta \left(s(l,m)\left(\bigvee_{p \in T(l)} r(l,p,m)\right)\right)^\dagger,$$

in the constraining context of the constrained expression $(\mathbf{F}', \epsilon')$, where ϵ' is the simplified system expression defined in (1.1) and $\mathbf{F}'$ is obtained from the derived constraining context of Σ by eliminating the constraints $\kappa_1(l,p)$, for all $l \in L$ and $p \in P$, along with their corresponding constraint alphabets.

Proof. Observe that $\mathcal{L}\left(\kappa_5'(l,m)\right) \subseteq \mathcal{L}\left(\kappa_5(l,m)\right)$. We therefore show that every constrained prefix of $\mathbf{D}$ satisfies $\kappa_5'(l,m)$, when viewed as a constraint over the alphabet $S_5(l,m)$, and apply proposition (V.2.2).

To see that, for a given link $l \in L$ and message type $m \neq \Omega$, every constrained prefix of $\mathbf{D}$ satisfies $\kappa_5'(l,m)$,observe that if a string from $\mathcal{L}\left(\kappa_5(l,m)\right)$ does not belong to $\mathcal{L}\left(\kappa_5'(l,m)\right)$, it must contain an $r(l,p,m)$ symbol, for some $p \notin T(l)$. Since $\rho_{S_j}\left(P(\epsilon')|_{\widehat{G}}\right) \subseteq \mathcal{L}\left(\kappa_5(l,m)\right)$ and the system expression ϵ' does not contain any $r(l,p,m)$ symbols, for $p \notin T(l)$, this clearly implies that $\rho_{S_j}\left(P(\epsilon')|_{\widehat{G}}\right) \subseteq \mathcal{L}(\kappa_5'(l,m))$, and so the constrained prefixes of $\mathbf{D}$ satisfy the constraint $\kappa_5'(l,m)$. The desired result then follows from proposition (V.2.2). ∎

To further simplify the derived constraining context, we next eliminate certain of the constraints $\kappa_6(l,p,m)$, for $l \in L$, $p \in P$, and $m \neq \Omega$, which assure that if a link l and port p are connected and a process starves waiting for a

communication through p, then there are no messages of type m residing in l to be received. Specifically, we eliminate the constraints $\kappa_6(l,p,m)$ for all links l and ports p in Σ that are not connected to one another by a channel and all message types $m \neq \mathbb{Q}$. As noted when these constraints were defined in §III.2, if a link l and port p are not connected by a communication channel, the constraint $\kappa_6(l,m,p)$, where $m \neq \mathbb{Q}$, does not restrict the order or number of $s(l,m)$ or $r(l,p',m)$ symbols, for any $p' \in P$, that appear in constrained prefixes of $\mathbf{D}$. It is easy to see, furthermore, that because of the form of the system expression and because of restrictions imposed by other constraints, the constraint $\kappa_6(l,p,m)$ is not needed for restricting the order or number of $w(p)$ or $ch(l,p)$ symbols that appear in prefixes of the simplified system expression. If a link l is not connected to a port p, therefore, the constraint $\kappa_6(l,p,m)$, where $m \neq \mathbb{Q}$, is not needed for restricting the order or number of any of the symbols from $S_6(l,p,m)$ appearing in prefixes of this simplified system expression. Hence these constraints can be eliminated from the derived constraining context of Σ.

(1.6) **Proposition.** *For every port $p \in P$, link $l \notin F(p)$, and message type $m \neq \mathbb{Q}$, the constraint $\kappa_6(l,p,m)$ and the corresponding constraint alphabet $S_6(l,p,m)$ can be eliminated from the constraining context of the constrained expression $(\mathbf{F}', \epsilon')$, where ϵ' is the simplified system expression defined in (1.1) and $\mathbf{F}'$ is obtained from the derived constraining context of Σ by eliminating the constraints $\kappa_1(l,p)$, for all $l \in L$ and $p \in P$, along with their corresponding constraint alphabets, and by simplifying the constraints $\kappa_5(l,m)$, for all $l \in L$ and $m \neq \mathbb{Q}$, as described in (1.5).*

The proof of this proposition is similar to the proof of proposition (1.3).

Finally, we simplify the remaining constraints $\kappa_6(l,p,m)$, for $p \in P$, $l \in F(p)$, and $m \neq \mathbb{Q}$. The constraint $\kappa_6(l,p,m)$, for a given port p, link $l \in F(p)$ and message type $m \neq \mathbb{Q}$, involves the symbols $s(l,m)$, $ch(l,p)$, $w(p)$ and $r(l,p',m)$, where p' ranges over all of P. As the simplified system expression

defined in (1.1) does not contain any $r(l,p',m)$ symbols for ports p' that are not connected to l by a channel, when this system expression is used in place of the derived system expression, the constraint $\kappa_6(l,p,m)$ need only restrict the order and number of $s(l,m)$, $ch(l,p)$, $w(p)$ and $r(l,p',m)$ symbols for those ports p' that are connected to l. If a port p' is not connected to the link l, therefore, the $r(l,p',m)$ symbol in $\kappa_6(l,p,m)$ corresponding to this particular port p' can be eliminated. As $l \in F(p)$, furthermore, the initial expression derived from the design of Σ contains a $ch(l,p)$ symbol. Every constrained prefix, therefore, contains a $ch(l,p)$ symbol. This permits us to eliminate alternatives of $\kappa_6(l,p,m)$ that do not contain a $ch(l,p)$ symbol. Finally, the $ch(l,p)$ symbols can be dropped from the constraint altogether, as we show below.

(1.7) **Proposition.** *For every port $p \in P$, link $l \in F(p)$ and message type $m \neq \mathbb{Q}$, the constraint $\kappa_6(l,p,m)$ can be replaced with the expression,*

$$\left(s(l,m)\left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{\dagger} w(p) \left(s(l,m)\left(\bigvee_{\substack{p' \in T(l) \\ p' \neq p}} r(l,p',m)\right)\right)^{\dagger}$$
$$\vee \left(s(l,m) \vee \left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{*},$$

if the corresponding constraint alphabet is replaced with the alphabet

$$\{\, s(l,m), r(l,p',m), w(p) \,\}_{p' \in P},$$

in the constraining context of the constrained expression $(\mathbf{F}', \epsilon')$, where ϵ' is the simplified system expression defined in (1.1) and $\mathbf{F}'$ is obtained from the derived constraining context of Σ by eliminating the constraints $\kappa_1(l,p)$, for all $l \in L$ and $p \in P$, and the constraints $\kappa_6(l,p,m)$, for all $p \in P$, $l \notin F(p)$ and $m \neq \mathbb{Q}$, along with their corresponding constraint alphabets, and by simplifying the constraints $\kappa_5(l,m)$, for all $l \in L$ and $m \neq \mathbb{Q}$, as described in (1.5).

Proof. We first note that, for a given port $p \in P$, link $l \in F(p)$ and message type $m \neq \mathbb{Q}$, an argument similar to the proof of proposition (1.5) shows that the constraint $\kappa_6(l,p,m)$ can be replaced with the expression,

$$ch(l,p)\left(s(l,m)\left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{\dagger} w(p)\left(s(l,m)\left(\bigvee_{\substack{p' \in T(l) \\ p' \neq p}} r(l,p',m)\right)\right)^{\dagger}$$
$$\vee \left(s(l,m) \vee \left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{*} \Delta \left(ch(l,p) \vee w(p) \vee \lambda\right). \tag{1.8}$$

Next we show the alternatives in (1.8) that do not contain a $ch(l,p)$ symbol can be eliminated. For this, we write expression (1.8) as,

$$ch(l,p)\left(s(l,m)\left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{\dagger} w(p)\left(s(l,m)\left(\bigvee_{\substack{p' \in T(l) \\ p' \neq p}} r(l,p',m)\right)\right)^{\dagger}$$
$$\vee \left[\left(s(l,m) \vee \left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{*} \Delta\, ch(l,p)\right]$$
$$\vee \left[\left(s(l,m) \vee \left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{*} \Delta \left(w(p) \vee \lambda\right)\right].$$

We then note, as in the proof of proposition (1.3), that the initial expression derived from the design of Σ is an initial substring of every prefix of the system expression satisfying the constraints of $\hat{C}_3 = \{\kappa_3(q)\}_{q \in Q}$. Because $l \in F(p)$, furthermore, the initial expression contains a $ch(l,p)$ symbol. We conclude, therefore, that every string in $P(\epsilon')|_{\hat{C}_3}$, and hence every constrained prefix, contains a $ch(l,p)$ symbol. Applying proposition (V.2.2), therefore, we conclude that, when used in place of $\kappa_6(l,p,m)$, the constraint of (1.8) can be replaced with the expression,

$$ch(l,p)\left(s(l,m)\left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{\dagger} w(p)\left(s(l,m)\left(\bigvee_{\substack{p' \in T(l) \\ p' \neq p}} r(l,p',m)\right)\right)^{\dagger}$$
$$\vee \left(s(l,m) \vee \left(\bigvee_{p' \in T(l)} r(l,p',m)\right)\right)^{*} \Delta\, ch(l,p). \tag{1.9}$$

Finally, we complete the simplification of the constraint $\kappa_6(l,p,m)$ by showing that, when used in place of $\kappa_6(l,p,m)$, the constraint (1.9) can be replaced with the expression

$$\begin{aligned}(1.10)\qquad &\left(s(l,m)\left(\bigvee_{p'\in T(l)} r(l,p',m)\right)\right)^{\dagger} w(p) \left(s(l,m)\left(\bigvee_{\substack{p'\in T(l)\\ p'\neq p}} r(l,p',m)\right)\right)^{\dagger} \\ &\vee \left(s(l,m)\vee\left(\bigvee_{p'\in T(l)} r(l,p',m)\right)\right)^{*},\end{aligned}$$

provided that the $ch(l,p)$ symbol is eliminated from the corresponding constraint alphabet. Every prefix of the system expression satisfying the constraints of $\hat{C}_3$ contains a single $ch(l,p)$ symbol, which precedes any $s(l,m)$, $w(p)$ or $r(l,p',m)$ symbols that may also appear in the prefix. A prefix of ϵ' that satisfies the constraints of $\hat{C}_3$, therefore, satisfies (1.9) if and only if it satisfies (1.10).

More formally, we apply proposition (V.2.3) with κ_j equal to (1.9), S'_j equal to the constraint alphabet $S_6(l,p,m)$ minus the symbol $ch(l,p)$, κ'_j equal to the expression of (1.10), and $\hat{B}$ equal to $\hat{C}_3$. Since every string in $\mathcal{L}(\kappa_j)$ clearly satisfies κ'_j, we need only verify that every string in $P(\epsilon')|_{\hat{C}_3}$ satisfying κ'_j also satisfies κ_j to show that proposition (V.2.3) applies, and so conclude that (1.9) can be replaced by (1.10), if the $ch(l,p)$ symbol is eliminated from the corresponding constraint alphabet.

Consider, therefore, an arbitrary prefix of ϵ' satisfying the constraints of $\hat{C}_3$ and also satisfying κ'_j (i.e., the expression of (1.10) considered as a constraint over the constraint alphabet S'_j). As noted above, this prefix contains the string associated with the initial expression that is derived from the design of Σ as an initial substring. By construction, furthermore, the initial expression begins with a string of $ch(l,p)$ symbols, one of which is the $ch(l,p)$ symbol corresponding to the given link l and port p. Additionally, there are no other $ch(l,p)$ symbols corresponding to the given link l and port p in this prefix of ϵ'. The projection

of the prefix on the alphabet $S_6(l,p,m)$, therefore, begins with a single $ch(l,p)$ symbol and contains no other $ch(l,p)$ symbols. Because the prefix also satisfies κ'_j, we conclude that its projection on $S_6(l,p,m)$ lies in the language of

$$ch(l,p)\left(s(l,m)\Big(\bigvee_{p'\in T(l)} r(l,p',m)\Big)\right)^{\dagger} w(p)\left(s(l,m)\Big(\bigvee_{\substack{p'\in T(l)\\ p'\neq p}} r(l,p',m)\Big)\right)^{\dagger}$$

$$\vee\, ch(l,p)\left(s(l,m)\vee\Big(\bigvee_{p'\in T(l)} r(l,p',m)\Big)\right)^{*},$$

which is clearly contained in the language of (1.9). The prefix, therefore, satisfies κ'_j, as desired. ∎

Simplification of the constraints $\kappa_6(l,m,p)$, for $p \in P$, $l \in F(p)$, and $m \neq \mathbb{Q}$, completes the preliminary simplifications that are performed before a derived constrained expression representation is reduced. These simplifications are summarized below.

(1.11) **Theorem.** *The constrained expression* $\mathbf{D} = (\mathbf{F}, \epsilon)$, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, *obtained from the* SDYMOL *design of a system* Σ *by taking*

(i) $\epsilon = \iota\Big(\bigtriangleup_{q\in Q} t'(\mathbf{code}_q)\Big)$, *where* ι *is the initial expression derived from the* SDYMOL *design of* Σ, *and* t' *is the map defined in (1.1),*

(ii) A *equal to the derived augmented alphabet,*

(iii) E *equal to the derived terminal alphabet,*

(iv) $\widehat{S} = \{\, S_2(q), S_3(q), S_4(q), S_5(l,m), S_6(l',m,p)\,\}_{q\in Q, l\in L, p\in P, m\neq \mathbb{Q}, l'\in F(p)}$, *where*

$$S_2(q) = \{\, ch(l,p), r(l,p,m)\,\}_{m\neq \mathbb{Q}},$$

$$S_3(q) = \{\, d(q,m), u(q,m)\,\}_{m\in M},$$

$$S_4(q) = \{\, ne(q)\,\},$$

$$S_5(l,m) = \{\, s(l,m), r(l,p,m)\,\}_{p\in P}, \text{ and}$$

$$S_6(l',p,m) = \{\, s(l',m), r(l',p',m), w(p)\,\}_{p'\in P},$$

and

(v) $\hat{C} = \{ \kappa_2(q), \kappa_3(q), \kappa_4(q), \kappa_5(l,m), \kappa_6(l',m,p) \}_{q\in Q, l\in L, p\in P, m\neq\Theta, l'\in F(p)}$,

where

$$\kappa_2(q) = \Big(\bigvee_{m\in M} d(q,m)\, u(q,m)^* \Big)^*,$$

$$\kappa_3(q) = \Big(\bigvee_{p\in P(q)} w(p) \Big) \vee stop(q),$$

$$\kappa_4(q) = \lambda,$$

$$\kappa_5(l,m) = s(l,m)^* \Big(s(l,m) \Big(\bigvee_{p\in T(l)} r(l,p,m) \Big) \Big)^\dagger, \text{ and}$$

$$\kappa_6(l',m,p) = \Big(s(l',m) \Big(\bigvee_{p'\in T(l)} r(l',p',m) \Big) \Big)^\dagger w(p)$$

$$\Big(s(l',m) \Big(\bigvee_{\substack{p'\in T(l) \\ p'\neq p}} r(l',p',m) \Big) \Big)^\dagger$$

$$\vee \Big(s(l',m) \vee \Big(\bigvee_{p'\in T(l)} r(l',p',m) \Big) \Big)^*,$$

is strongly equivalent to the derived constrained expression representation of Σ.

The system expression and constraints that are obtained by applying these simplifications to the derived constrained expression representation of the solution to the dining philosophers problem (see figures III.1 and III.3) are shown in figures 1 and 2.

§2. Additional simplifications

The constrained expression representation of an SDYMOL system described in (1.11) can be reduced with the procedure described in the next section. It is

$$\iota = \prod_{0 \le i \le 4} ch(p_i.ld, f_i.d)\, ch(p_i.rd, f_{i-1}.d)\, ch(f_i.u, p_i.lu)\, ch(f_i.u, p_{i+1}.ru)$$
$$\prod_{0 \le i \le 4} d(p_i, @)\, d(f_i, @).$$

$$t'(\mathbf{code}_{p_i}) =$$
$$\Big[\ think_i$$
$$\Big(r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok) \vee w(p_i.ru)\, ne(p_i)\Big)$$
$$\Big(r(f_i.u, p_i.lu, ok)\, d(p_i, ok) \vee w(p_i.lu)\, ne(p_i)\Big)$$
$$eat_i$$
$$\Big(u(p_i, ok)\, s(p_i.ld, ok) \vee u(p_i, @)\Big)$$
$$\Big(u(p_i, ok)\, s(p_i.rd, ok) \vee u(p_i, @)\Big)\Big]^*\ stop(p_i)$$

$$t'(\mathbf{code}_{f_i}) =$$
$$d(f_i, ok)$$
$$\Big[\ \Big(u(f_i, ok)\, s(f_i.u, ok) \vee u(f_i, @)\Big)$$
$$\Big(r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \vee r(p_{i+1}.rd, f_i.d, ok) d(f_i, ok)$$
$$\vee w(f_i.d)\, ne(f_i)\Big)\Big]^*\ ne(f_i)\ stop(f_i)$$

$$\epsilon = \iota\left(\left(\underset{0 \le i \le 4}{\triangle} t'(\mathbf{code}_{p_i})\right) \triangle \left(\underset{0 \le i \le 4}{\triangle} t'(\mathbf{code}_{f_i})\right)\right)$$

Figure 1

System expression for a constrained expression representation of a solution to the dining philosophers problem

$$\kappa_2(f_i) = \Big(d(f_i, ok)\, u(f_i, ok)^* \vee d(f_i, @)\, u(f_i, @)^*\Big)^*$$

$$\kappa_2(p_i) = \Big(d(p_i, ok)\, u(p_i, ok)^* \vee d(p_i, @)\, u(p_i, @)^*\Big)^*$$

$$\kappa_3(f_i) = w(f_i.d) \vee stop(f_i)$$

$$\kappa_3(p_i) = w(p_i.ru) \vee w(p_i.lu) \vee stop(p_i)$$

$$\kappa_4(f_i) = \lambda$$

$$\kappa_4(p_i) = \lambda$$

$$\kappa_5(f_i.u, ok) = s(f_i.u, ok)^* \triangle \Big(s(f_i.u, ok)\Big[r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big]\Big)^\dagger$$

$$\kappa_5(p_i.ld, ok) = s(p_i.ld, ok)^* \triangle \Big(s(p_i.ld, ok)\, r(p_i.ld, f_i.d, ok)\Big)^\dagger$$

$$\kappa_5(p_i.rd, ok) = s(p_i.rd, ok)^* \triangle \Big(s(p_i.rd, ok)\, r(p_i.rd, f_{i-1}.d, ok)\Big)^\dagger$$

$$\kappa_6(f_i.u, p_i.lu, ok) =$$

$$\Big(s(f_i.u, ok)\Big[r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big]\Big)^\dagger\, w(p_i.lu)$$

$$\Big(s(f_i.u, ok)\, r(f_i.u, p_{i+1}.ru, ok)\Big)^\dagger$$

$$\vee \Big(s(f_i.u, ok) \vee r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)^*$$

Figure 2

Constraints for a constrained expression representation of a solution to the dining philosophers problem

$$\kappa_6(f_i.u, p_{i+1}.ru, ok) =$$
$$\Big(s(f_i.u, ok)\,\big[r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\big]\Big)^{\dagger}\, w(p_{i+1}.ru)$$
$$\Big(s(f_i.u, ok)\, r(f_i.u, p_i.lu, ok)\Big)^{\dagger}$$
$$\vee \Big(s(f_i.u, ok) \vee r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)^{*}$$

$$\kappa_6(p_i.rd, f_{i-1}.d, ok) =$$
$$\Big(s(p_i.rd, ok)\, r(p_i.rd, f_{i-1}.d, ok)\Big)^{\dagger}\, w(f_{i-1}.d)$$
$$\vee \Big(s(p_i.rd, ok) \vee r(p_i.rd, f_{i-1}.d, ok)\Big)^{*}$$

$$\kappa_6(p_i.ld, f_i.d, ok) =$$
$$\Big(s(p_i.ld, ok)\, r(p_i.ld, f_i.d, ok)\Big)^{\dagger}\, w(f_i.d)$$
$$\vee \Big(s(p_i.ld, ok) \vee r(p_i.ld, f_i.d, ok)\Big)^{*}$$

Figure 2 (continued)

sometimes desirable, however, to simplify the system expression of this constrained expression further, before reducing it. Any simplifications of the system expression that are performed before a constrained expression is reduced make it easier to apply the reduction procedure and result in a simpler reduced constrained expression. To perform these simplifications at this point, however, requires a certain amount of insight on the part of the system designer.

If the reduction and simplification of constrained expressions were automated, the constrained expression representation of a system described in (1.11) would be mechanically generated, reduced, and then simplified using the algorithms described in chapter VII. The only reason for simpifying this representation before reducing it would then be to avoid exhausting the available computing resources. (It wouldn't matter if the system expression produced during reduction were large, as it would then be mechanically simplified. The same end result would be obtained whether simplification was performed before or after reduction.) When manually reducing and simplifying a constrained expression representation of a system, however, we often simplify the representation of a system described in (1.11) before reducing it, so that the application of the reduction procedure will be less tedious and error-prone. In this section we give examples of additional simplifications that can be performed in such cases.

As the reduction procedure described in the next section requires a constrained expression with certain properties, and the constrained expression representation of a system described in (1.11) is easily seen to satisfy these properties, we require that any additional simplifications performed before the expression is reduced preserve these properties. We therefore examine these properties before considering specific simplifications that can be used to facilitate the reduction of a constrained expression.

Process expression reduction criteria

For the discussion in this section, let $\mathbf{D} = (\mathbf{F}, \epsilon)$ be a constrained expression representation of an SDYMOL system Σ, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$. The reduction procedure produces an equivalent, reduced constrained expression when applied to $\mathbf{D}$ provided (i) that the constraint set $\widehat{C}$ contains the constraints $\kappa_3(q)$, for $q \in Q$, which assure that the processes in the system run to completion, and $\kappa_4(q)$, for $q \in Q$, which eliminate prefixes containing non-event symbols, and (ii) that the system expression ϵ can be expressed as $\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q\right)$, where ι is the initial expression derived from the SDYMOL design of Σ and the process expressions π_q, for $q \in Q$, meet the four criteria, called the *process expression reduction criteria*, described below.

First, to show that the reduction procedure produces an equivalent constrained expression when applied to $\mathbf{D}$, we require that the process expression π_q, for $q \in Q$, only contain symbols that can appear in the derived process expression for q. We use the following definition to state this requirement.

(2.1) *Definition.* Given an SDYMOL system Σ, the *derived alphabet* of the process q, for $q \in Q$, is defined by

$$A_q = \{\, u(q,m), d(q,m), s(l',m), r(l,p',m), w(p'), ne(q), stop(q) \,\} \cup F_q,$$

where m ranges over M, l' ranges over $L(q)$, l ranges over L, and p' ranges over $P(q)$, and where F_q denotes the set of identifier symbols associated with the process q.

Thus, for example,

$$A_{p_i} = \{\, u(p_i, ok), u(p_i, @), d(p_i, ok), d(p_i, @), r(f_i.u, p_i.lu, ok), r(f_{i-1}.u, p_i.ru, ok),$$
$$s(p_i.ld, ok), s(p_i.rd, ok), w(p_i.lu), w(p_i.ru), ne(p_i), stop(p_i), think_i, eat_i \,\}$$

and

$$A_{f_i} = \{\, u(f_i, ok), u(f_i, @), d(f_i, ok), d(f_i, @), r(p_{i+1}.rd, f_i.d, ok), \\ r(p_i.ld, f_i.d, ok), s(f_i.u, ok), w(f_i.u), ne(f_i), stop(f_i) \,\}$$

for the SDYMOL solution to the dining philosophers problem presented in figures III.1 and III.3.

The derived alphabet A_q, therefore, denotes the alphabet of symbols appearing on the right hand sides of the translation rules (see figure III.5) that can be used when deriving a process expression for the process q, and we require that $\pi_q \in \mathcal{RE}(A_q)$. This requirement is certainly met as long as new symbols are not introduced in a process expression when simplifying it.

Second, in order to apply the reduction procedure we require that the process expression π_q, for $q \in Q$, be expressed in the special form, called *disjunctive form*, defined below.

(2.2) *Definition.* The expressions of $\mathcal{RE}(A)$ that are in *disjunctive form* are defined recursively to be the regular expressions of the form $\bigvee_{1 \le j \le m} \alpha_{1j}\alpha_{2j}\ldots\alpha_{n_j j}$, where (i) $m \ge 1$, (ii) for all $1 \le j \le m$, $n_j \ge 1$, and (iii) for all $1 \le j \le m$ and $1 \le i \le n_j$, either α_{ij} is one of the empty regular expression ($\emptyset$), the empty string (λ), or a single event symbol ($a \in A$), or α_{ij} has the form β^*, where β is in disjunctive form.[1]

Given a regular expression $\bigvee_{1 \le j \le m} \alpha_{1j}\alpha_{2j}\ldots\alpha_{n_j j}$ in disjunctive form, the expressions $\alpha_{1j}\alpha_{2j}\ldots\alpha_{n_j j}$, for $1 \le j \le m$, are its *terms* and the expresssions α_{ij}, for $1 \le j \le m$ and $1 \le i \le n_j$, are the *components* of the jth term. A component is *basic* if it is one of $\emptyset$, λ, or a, where $a \in A$, and is *complex* otherwise. The *subterms* of a regular expression in disjunctive form consist of its terms and the

[1] As a notational convenience, we assume that concatenation and disjunction associate from left to right. Thus, for example, we write abc for $(ab)c$ and $a \vee b \vee c$ for $(a \vee b) \vee c$.

terms of its iterated subexpressions, where an *iterated subexpression* of a regular expression is any subexpression that constitutes the scope of a star operator, called the *enclosing star operator.*

Clearly, a regular expression that does not involve the shuffle operator can be put in disjunctive form by simply distributing concatenation over disjunction wherever possible, and, if necessary, reassociating the operands in series of concatenations and/or disjunctions. Since there are other ways to put such regular expressions in disjunctive form, and since some of these produce quite different looking results, we consider this method to be the *standard fashion* for putting a regular expression that does not involve the shuffle operator in disjunctive form.

The process expressions $t'(\mathbf{code}_q)$, for $q \in Q$, can be put in disjunctive form in the standard fashion. Doing this to the process expression $t'(\mathbf{code}_{f_i})$, where $0 \leq i \leq 4$ (see figure 1), for example, produces the expression consisting of the single term,

$$
\begin{aligned}
& d(f_i, ok) \\
& \quad [u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \qquad \vee u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok) \\
& \qquad \vee u(f_i, ok)\, s(f_i.u, ok)\, w(f_i.d)\, ne(f_i) \\
& \qquad \vee u(f_i, @)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \qquad \vee u(f_i, @)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok) \\
& \qquad \vee u(f_i, @)\, w(f_i.d)\, ne(f_i)]^* \\
& \quad ne(f_i)\, stop(f_i).
\end{aligned}
\tag{2.3}
$$

This term has four components. The first, third and fourth components are all basic components and consist of the single event symbols $d(f_i, ok)$, $ne(f_i)$, and $stop(f_i)$, respectively. The second component is a complex component and consists of the single iterated subexpression of (2.3) and the enclosing star operator. The expression (2.3) has seven subterms, one being the expression itself and the

remaining six being the terms of the iterated subexpression.

Finally, to show that the reduction procedure described in section §3, when applied to **D**, produces an equivalent constrained expression that is also reduced, we require that the process expressions π_q, for $q \in Q$, meet two additional criteria. We require that each $stop(q)$ symbol in π_q constitute the last component of some term, and that every $w(p)$ symbol in π_q be immediately followed by an $ne(q)$ symbol (i.e., if a subterm of π_q contains a component consisting of a $w(p)$ symbol, then this component is not the last component of the subterm and the next component in the subterm is an $ne(q)$ symbol). The process expression reduction criteria are summarized in figure 3.

(i) $\pi_q \in \mathcal{RE}(A_q)$,

(ii) π_q is in disjunctive form,

(iii) every term of π_q containing a $stop(q)$ symbol contains only one such symbol, and the single $stop(q)$ symbol in a term is the last component of the term, and

(iv) every $w(p)$ symbol in π_q is immediately followed by an $ne(q)$ symbol (i.e., if a subterm $\beta_1 \ldots \beta_n$ of π_q contains a component $\beta_k = w(p)$, then $k < n$ and $\beta_{k+1} = ne(q)$).

Figure 3

Process expression reduction criteria

In summary, the reduction procedure described in the next section can be applied to the constrained expression **D** to produce an equivalent reduced constrained expression representation of the system Σ as long as the set of constraints $\hat{C}$ contains the constraints $\kappa_3(q) = stop(q) \vee \left(\bigvee_{p \in P(q)} w(p) \right)$ and $\kappa_4(q) = \lambda$, for all

$q \in Q$, and the system expression can be expressed as $\epsilon = \iota\left(\underset{q\in Q}{\triangle} \pi_q\right)$, where ι is the system expression derived from the design of Σ and the process expressions π_q, for $q \in Q$, satisfy the process expression reduction criteria. When the process expressions $t'(\mathbf{code}_q)$ in the system expression of the constrained expression representation for Σ described in (1.11) are put in disjunctive form in the standard fashion, the resulting process expressions meet the process expression reduction criteria. As the constraint set also contains the constraints $\kappa_3(q)$ and $\kappa_4(q)$, for $q \in Q$, the reduction procedure can be applied directly to this constrained expression representation of Σ. Any further simplifications to this constrained expression, if performed before reducing it, must preserve these characteristics of the representation.

Some additional simplifications

The following three propositions describe simplifications that can be performed before the constrained expression representation of a system is reduced. In each of these propositions, the system expression of the given representation, when expressed as required for the reduction procedure, is simplified by simplifying one of its process expressions. As the simplification of the process expression preserves the process expression reduction criteria, these are examples of ways in which the process expressions can be simplified before a constrained expression representation of a system is reduced.

(2.4) **Proposition.** *Given a constrained expression representation* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *of an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\left(\underset{q\in Q}{\triangle} \pi_q\right)$, ι *is the initial expression derived from the design of* Σ, *and the process expressions* π_q, *for* $q \in Q$, *meet the process expression reduction criteria (i)–(iv) of figure 3. If the process expression* π_q, *for some* $q \in Q$, *contains a subterm of the form* $\beta_1 \cdots \beta_n$ *with basic component* β_k, *for some* $1 \leq k \leq n$, *consisting of an* $ne(q)$ *symbol, then the "tail" of the subterm, beginning with the* $k+1$ *th component, can be eliminated (the last*

$n-k$ components of the subterm can be replaced with λ), to produce a process expression π'_q that meets the process expression reduction criteria and can be used in place of the process expression π_q.

Proof. Certainly π'_q meets the process expression reduction criteria, since π_q does and eliminating the tail beginning with the $k+1$ th component of a subterm does not affect the process expression reduction criteria (i)–(iii), and can only affect the criterion (iv) if the k th component of the subterm is a $w(p)$ symbol.

We show that $\mathcal{P}(\pi_q)|_{\kappa_4(q)} = \mathcal{P}(\pi'_q)|_{\kappa_4(q)}$, and then use proposition (V.2.5) to conclude that the process expression π'_q can be used in place of the process expression π_q in the system expression of $\mathbf{D}$.

To see that $\mathcal{P}(\pi_q)|_{\kappa_4(q)} \subseteq \mathcal{P}(\pi'_q)|_{\kappa_4(q)}$, we note that π'_q is equal to the regular expression obtained from π_q by replacing the given subterm with the regular expression $\beta_1 \cdots \beta_k$. If a string belongs to $\mathcal{P}(\pi_q)$ and not to $\mathcal{P}(\pi'_q)$, therefore, proposition (II.2.5) implies that it contains a substring belonging to the set $\mathcal{L}(\beta_1 \cdots \beta_n) - \mathcal{L}(\beta_1 \cdots \beta_k) \subseteq \mathcal{L}(\beta_1 \cdots \beta_n)$ or a tail belonging to the set $\mathcal{P}(\beta_1 \cdots \beta_n) - \mathcal{P}(\beta_1 \cdots \beta_k) \subseteq \mathcal{L}(\beta_1 \cdots \beta_k)\mathcal{P}(\beta_{k+1} \cdots \beta_n)$. In either case, it contains an $ne(q)$ symbol, since $\beta_k = ne(q)$, and so $\mathcal{P}(\pi_q)|_{\kappa_4(q)} \subseteq \mathcal{P}(\pi'_q)|_{\kappa_4(q)}$, as desired.

Similarly, to establish the reverse inclusion, we note that proposition (II.2.5) implies that if a string belongs to $\mathcal{P}(\pi'_q)$ and not to $\mathcal{P}(\pi_q)$, then it contains a substring belonging to $\mathcal{L}(\beta_1 \cdots \beta_k) - \mathcal{L}(\beta_1 \cdots \beta_n) \subseteq \mathcal{L}(\beta_1 \cdots \beta_k)$ (it does not contain a tail belonging to the set $\mathcal{P}(\beta_1 \cdots \beta_k) - \mathcal{P}(\beta_1 \cdots \beta_n) \subseteq \emptyset$). As $\beta_k = ne(q)$, it contains an $ne(q)$ symbol, and so $\mathcal{P}(\pi'_q)|_{\kappa_4(q)} \subseteq \mathcal{P}(\pi_q)|_{\kappa_4(q)}$. Proposition (V.2.5) thus implies that the process expression π'_q can be used in place of the process expression π_q. ∎

(2.5) **Proposition.** *Given a constrained expression representation $\mathbf{D} = (\mathbf{F}, \epsilon)$ of an* SDYMOL *system Σ, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\left(\bigtriangleup_{q \in Q} \pi_q\right)$, ι is the initial*

expression derived from the design of Σ, *and the process expressions* π_q, *for* $q \in Q$, *meet the process expression reduction criteria (i)–(iv) of figure 3. If the process expression* π_q, *for some* $q \in Q$, *contains a subterm of the form* $\beta_1 \cdots \beta_n$ *with two basic components* β_k *and* β_l, *for some* $1 \leq k < l \leq n$, *such that*

(i) β_k *consists of either a* $d(q,m)$ *symbol or a* $u(q,m)$ *symbol, for some* $m \in M$,

(ii) β_l *consists of a* $u(q,m')$ *symbol, for some* $m' \neq m$,

(iii) there are no intervening $d(q,m')$ *symbols (i.e.,* $\rho_{d(q,m')}(\beta_i) = \lambda$, *for* $k \leq i \leq l$), *and*

(iv) there are no $w(p)$ *symbols in any of the first* l *components of the subterm (i.e.,* $\rho_{w(p)}(\beta_i) = \lambda$, *for* $1 \leq i \leq l$),

then the subterm can be eliminated (replaced with $\emptyset$), *to produce a process expression* π'_q *that can be used in place of the process expression* π_q.

Proof. The regular expression π'_q meets the process expression reduction criteria since π_q does and eliminating an entire subterm of π_q does not affect these criteria.

We define $\widehat{D} = \{\kappa_2(q), \kappa_3(q), \kappa_4(q)\}$ and show that the prefixes of $d(q,@)\,\pi_q$ that satisfy the constraints of $\widehat{D}$ and the prefixes of $d(q,@)\,\pi'_q$ that satisfy the constraints of $\widehat{D}$ are the same. (Recall that $\kappa_2(q) = \left(\bigvee_{m \in M} d(q,m)\,u(q,m)^*\right)^*$.) Proposition (V.2.5) then implies that the regular expression π'_q can be used in place of the process expression π_q in the system expression of $\mathbf{D}$.

To demonstrate $P\left(d(q,@)\,\pi_q\right)\Big|_{\widehat{D}} \subseteq P\left(d(q,@)\,\pi'_q\right)\Big|_{\widehat{D}}$, we assume that $u \in P\left(d(q,@)\,\pi_q\right)\Big|_{\widehat{D}}$ but $u \notin P\left(d(q,@)\,\pi'_q\right)\Big|_{\widehat{D}}$, and show that this assumption results in a contradiction. For this, we first observe that hypotheses (i)–(iii) of the proposition imply that no string in $\mathcal{L}(\beta_k \cdots \beta_l)$ satisfies the constraint $\kappa_2(q)$. Hence u does not contain a substring belonging to this language. Furthermore, since none of the components $\beta_1, \ldots, \beta_l$ contain any $w(p)$ symbols (by hypothesis (iv)) or $stop(q)$ symbols (implied by the process expression reduction criterion (iii)) and the constraint $\kappa_3(q)$ assures that every constrained prefix contains a $stop(q)$ symbol

or $w(p)$ symbol, for some $p \in P(q)$, we conclude that $P\left(d(q, @)\beta_1 \cdots \beta_n\right)\Big|_{\widehat{D}} = \emptyset$.

Using these observations we now show that some non-null tail of u belongs to $P(\beta_1 \cdots \beta_{l-1})$. For this, we note that the subterm $\beta_1 \cdots \beta_n$ is either a term of π_q or a term of an iterated subexpression of π_q. It is not equal to π_q, since $P\left(d(q, @)\beta_1 \cdots \beta_n\right)\Big|_{\widehat{D}} = \emptyset$ and $u \in P\left(d(q, @)\pi_q\right)\Big|_{\widehat{D}}$. If it is a term of π_q, therefore, there is some regular expression $\delta \neq \emptyset$ such that $\pi_q = \delta \vee \beta_1 \cdots \beta_n$ and $\pi'_q = \delta$ (take δ equal to the disjunction of all the other terms of π_q, for instance). Proposition (II.2.5) thus implies that u contains a substring belonging to $\mathcal{L}(\delta \vee \beta_1 \cdots \beta_n) - \mathcal{L}(\delta) \subseteq \mathcal{L}(\beta_1 \cdots \beta_n)$ or a tail belonging to $P(\delta \vee \beta_1 \cdots \beta_n) - P(\delta) \subseteq P(\beta_1 \cdots \beta_n) - \{\lambda\}$. Since u does not contain a substring belonging to $\mathcal{L}(\beta_l \cdots \beta_k)$, we conclude that some non-null tail of u belongs to $P(\beta_1 \cdots \beta_{l-1})$, as desired. Similarly, if the subterm $\beta_1 \cdots \beta_n$ is a term of an iterated subexpression of π_q and the iterated subexpression is not equal to the subterm $\beta_1 \cdots \beta_n$ (so that π'_q is equal to the regular expression obtained from π_q by replacing the iterated subexpression with the disjunction of all the other terms of the iterated subexpression, which is not equal to $\emptyset$), we again use proposition (II.2.5) to conclude that some non-null tail of u belongs to $P(\beta_1 \cdots \beta_{l-1})$. Otherwise, if the subterm $\beta_1 \cdots \beta_n$ is an iterated subexpression of π_q, then π'_q is equal to the regular expression obtained from π_q by replacing the subterm and the enclosing star operator with $\emptyset^* = \lambda$. Proposition (II.2.5) thus implies that u has a substring belonging to $\mathcal{L}\left((\beta_1 \cdots \beta_n)^*\right) - \mathcal{L}(\lambda)$ or a tail belonging to $P\left((\beta_1 \cdots \beta_n)^*\right) - P(\lambda)$. Since u does not contain a substring belonging to $\mathcal{L}(\beta_k \cdots \beta_l)$, we again conclude that some non-null tail of u belongs to $P(\beta_1 \cdots \beta_{l-1})$, as desired.

To arrive at the desired contradiction, we finally observe that the process expression reduction criteria (i), (iii), and (iv), and the constraints $\kappa_3(q)$ and $\kappa_4(q)$ imply that the final symbol in u is either a $stop(q)$ symbol or a $w(p)$ symbol, where $p \in P(q)$. But this is impossible since none of the components $\beta_1, \ldots, \beta_{l-1}$ contain any $stop(q)$ or $w(p)$ symbols and some non-null tail of u belongs to $P(\beta_1 \cdots \beta_{l-1})$.

The desired inclusion is thus established.

The reverse inclusion is established by observing $P(\pi'_q) \subseteq P(\pi_q)$, which is implied by the definition of π'_q and proposition (II.2.5).

We conclude that $P\left(d(q, \mathbb{Q})\,\pi'_q\right)\Big|_{\widehat{D}} = P\left(d(q, \mathbb{Q})\,\pi_q\right)\Big|_{\widehat{D}}$. Thus proposition (V.2.5) implies that the process expression π_q can be replaced with the process expression π'_q. ∎

(2.6) **Proposition.** *Given a constrained expression representation* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *of an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\left(\underset{q \in Q}{\triangle}\, \pi_q\right)$, ι *is the initial expression derived from the design of* Σ, *and the process expressions* π_q, *for* $q \in Q$, *meet the process expression reduction criteria (i)–(iv) of figure 3. If the process expression* π_q, *for some* $q \in Q$, *contains a subterm of the form* $\beta_1 \cdots \beta_n$ *with a basic component* β_k, *for some* $1 \leq k \leq n$, *such that*

(i) β_k *consists of an* $r(l, p, m)$ *symbol, for some* $l \in L$, $p \in P(q)$, *and* $m \neq \mathbb{Q}$,

(ii) there are no $s(l, m)$ *symbols in the process expression of the process* q' *containing the link* l *or in the initial expression (i.e.,* $\rho_{s(l,m)}(\iota\pi_{q'}) = \lambda$*), and*

(iii) there are no $w(p)$ *symbols in any of the first* k *components of the subterm (i.e.,* $\rho_{w(p)}(\beta_i) = \lambda$, *for* $1 \leq i \leq k$*),*

then the subterm can be eliminated (replaced with $\emptyset$*) to produce a process expression* π'_q *that meets the process expression reduction criteria and can be used in place of* π_q.

Proof. Just as in the proof of the previous proposition, we observe that the regular expression π'_q meets the process expression reduction criteria and that $P(\pi'_q) \subseteq P(\pi_q)$. This latter result, of course, implies that $P(\epsilon')|_{\widehat{C}} \subseteq P(\epsilon)|_{\widehat{C}}$, where ϵ' denotes the system expression obtained from ϵ by replacing the process expression π_q with π'_q.

We show that $P(\epsilon)|_{\widehat{C}} \subseteq P(\epsilon')|_{\widehat{C}}$ by taking $\widehat{D} = \{\kappa_3(q), \kappa_4(q), \kappa_5(l, m)\}$ and demonstrating that $P(\epsilon)|_{\widehat{D}} \subseteq P(\epsilon')|_{\widehat{D}}$. For this, we assume that $u \in P(\epsilon)|_{\widehat{D}}$ but

$u \notin P(\epsilon')|_{\widehat{D}}$, and show that this assumption leads to a contradiction (essentially the same contradiction as in the previous proposition).

Clearly, $u \in P(\epsilon)$ and $u \notin P(\epsilon')$ implies that there is a string v such that $v \neq \lambda$, $v \in P(\pi_q) - P(\pi'_q)$, and $\rho_{A_q}(u) = iv$, where $i = \rho_{A_q}(\iota)$ and A_q is the derived alphabet of q. Now v does not contain a substring belonging to $\mathcal{L}(\beta_k) = \{r(l,p,m)\}$, since u satisfies the constraint $\kappa_5(l,m)$ and hypothesis (ii) of the proposition and property (i) of the process expression reduction criteria imply that there are no $s(l,m)$ symbols in u. An argument similar to the one presented in the previous proposition, therefore, shows that some non-null tail of v, and hence, of $\rho_{A_q}(u)$, belongs to $P(\beta_1 \cdots \beta_{k-1})$. The process expression reduction criteria (i), (iii), and (iv), and the constraints $\kappa_3(q)$ and $\kappa_4(q)$, however, imply that the final symbol in $\rho_{A_q}(u)$ is either a $stop(q)$ symbol or a $w(p)$ symbol, for some $p \in P(q)$. Since none of the components $\beta_1, \ldots, \beta_{k-1}$ contain any $stop(q)$ or $w(p)$ symbols, this is impossible, and so the desired contradiction is obtained.

This proves that $(\mathbf{F}, \epsilon')$ is strongly equivalent to $(\mathbf{F}, \epsilon)$, and so the process expression π_q can be replaced by the process expression π'_q, as desired. ∎

It is fairly easy to recognize when the above propositions can be applied to simplify a given constrained expression representation. Additionally, their application can dramatically reduce the tedium of (manually) applying the reduction procedure and result in a much smaller system expression, which is therefore easier to read and simplify further. Of course, any additional simplifications that preserve the process expression reduction criteria can be performed before a constrained expression representation of a system is reduced. It is, however, often easier to identify and perform such additional simplifications after the representation is reduced.

To see how these propositions are applied, consider the process expression for the philosopher process p_i, where $0 \leq i \leq 4$, shown in figure 1. When this process expression is put in disjunctive form in the standard fashion it contains seventeen subterms, one being the process expression itself and the remaining sixteen

being terms of its single iterated subexpression. All but one subterm of the iterated subexpression, however, can be simplified or eliminated. For instance, proposition (2.4) implies that the eight subterms beginning $think_i\ w(p_i.ru)\ ne(q)\cdots$ can all be replaced with $think_i\ w(p_i.ru)\ ne(q)$, and proposition (2.5) implies that the remaining subterms that contain any $u(p_i, @)$ symbols can all be eliminated. Using these propositions, therefore, the process expression is easily simplified, to produce the process expression,

$$
\begin{aligned}
(2.7)\qquad & [think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) && (1)\\
& \quad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok) && (2)\\
& \vee think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ w(p_i.lu)\ ne(p_i) && (3)\\
& \vee think_i\ w(p_i.ru)\ ne(p_i)]^* && (4)\\
& stop(p_i). && (5)
\end{aligned}
$$

The seventeen subterms of the original process expression are thus reduced to four simpler subterms. Clearly, (2.7) meets the process expression reduction criteria shown in figure 3.

§3. Reducing the constrained expression representation

In this section we describe a procedure for reducing any appropriately simplified constrained expression representation of an SDYMOL system Σ. As explained in the previous section, this reduction procedure requires a constrained expression representation of the system whose constraint set contains the constraints $\kappa_3(q)$ and $\kappa_4(q)$, for $q \in Q$, and whose system expression can be expressed as $\epsilon = \iota\left(\underset{q\in Q}{\triangle}\ \pi_q\right)$, where ι is the initial expression derived from the design of Σ and the process expressions π_q, for $q \in Q$, meet the process expression reduction criteria (i)–(iv) shown in figure 3. Since the process expressions $t'(\mathbf{code}_q)$, for $q \in Q$, meet the process expression reduction criteria when put in disjunctive form in the standard

fashion, the reduction procedure can be applied to the constrained expression representation of a system described in (1.11). Alternately, the system expression of this representation can be further simplified, as described in the previous section, and then the reduction procedure applied.

The reduction procedure involves generating a new system expression that can be used in place of the given system expression, with the property that every prefix satisfying the constraints of the given constrained expression is a full string in the language of the new system expression. Of course, generating a system expression with just these properties is actually trivial. In proposition (II.4.8) we defined a procedure that transforms any regular expression into a regular expression whose language is exactly the set of prefixes of the original regular expression. This procedure can be applied to any system expression ϵ to obtain a regular expression ϵ' satisfying $P(\epsilon) = \mathcal{L}(\epsilon')$. As this equality implies that $P(\epsilon') = \mathcal{L}(\epsilon')$, for any given set of constraints $\widehat{C}$, we then have $P(\epsilon)|_{\widehat{C}} = \mathcal{L}(\epsilon')|_{\widehat{C}} = P(\epsilon')|_{\widehat{C}}$, so that ϵ' can be used in place of ϵ, and $P(\epsilon')|_{\widehat{C}} = \mathcal{L}(\epsilon')|_{\widehat{C}}$, so that the prefixes of ϵ' satisfying the constraints of $\widehat{C}$ are all full strings in the language of ϵ', as desired. The problem with using this procedure to produce a reduced system expression of an SDYMOL system, however, is that it usually produces an unnecessarily long and complex system expression. The reduction procedure described below produces a much simpler system expression. Only those prefixes of the original system expression satisfying the constraints $\kappa_3(q)$ and $\kappa_4(q)$, for $q \in Q$, belong to the language of this system expression. Many impossible prefixes of the original system expression are thus eliminated by the reduction procedure described in this section.

For the remainder of this section, assume that $\mathbf{D} = (\mathbf{F}, \epsilon)$ is a constrained expression representation of an SDYMOL system Σ, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$. Assume, additionally, that the constraints $\kappa_3(q), \kappa_4(q) \in \widehat{C}$, for $q \in Q$, and that $\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q\right)$, where ι is the initial expression derived from the design of Σ, and the process expressions π_q, for $q \in Q$, meet the process expression reduction

criteria (i)–(iv) of figure 3. We explain how to generate a new system expression, ϵ', that can be used in place of ϵ, to produce a reduced constrained expression representation $\mathbf{D}' = (\mathbf{F}, \epsilon')$ for Σ, such that the language of ϵ' contains only those prefixes of ϵ satisfying the constraints $\kappa_3(q)$ and $\kappa_4(q)$, for $q \in Q$.

Recall that, for a given $q \in Q$, the constraint $\kappa_3(q)$ assures that a single $stop(q)$ or $w(p)$ symbol, for $p \in P(q)$, appears in every constrained prefix of $\mathbf{D}$, while the constraint $\kappa_4(q)$ assures that no $ne(q)$ symbols appear in any constrained prefix of $\mathbf{D}$. To obtain the desired system expression from the original system expression, therefore, we transform the process expression π_q, for each $q \in Q$, into a regular expression, which we denote by π'_q and refer to as the *reduced process expression*, so that the language of π'_q consists of exactly those prefixes of π_q that contain a single $stop(q)$ or $w(p)$ symbol, for some $p \in P(q)$, and do not contain any $ne(q)$ symbols, and so that the single $stop(q)$ or $w(p)$ symbol appearing in every string from the language of π'_q is the last symbol in the string. We then substitute the reduced process expressions for the original process expressions in the expression for ϵ, so that $\epsilon' = i\left(\underset{q \in Q}{\triangle} \pi'_q\right)$. Informally, the idea is to use the constraints $\kappa_3(q)$ and $\kappa_4(q)$ to assure that the process expression π_q, for each $q \in Q$, can be replaced with the reduced process expression π'_q. Every string from the language of the system expression ϵ' obtained in this manner satisfies the constraints $\kappa_3(q)$ and $\kappa_4(q)$, for $q \in Q$, and conversely, every prefix of ϵ' satisfying the constraints $\kappa_3(q)$, for $q \in Q$, consists of the initial expression followed by some interleaving of full strings, one for each $q \in Q$, from the languages of the π'_q (since the single $stop(q)$ or $w(p)$ symbol in every string from the language of π'_q, for every $q \in Q$, is the last symbol in the string), and so constitutes a full string in the language of ϵ'.

Before explaining how to reduce an arbitrary process expression that meets the process expression reduction criteria, we consider an example. The example should help clarify the more general explanation presented later. For this exam-

ple, we use the process expression for the philosopher process p_i generated in the previous section. We therefore take π_{p_i} to be the process expression of (2.7).

We first transform π_{p_i} into a process expression with the following properties: (i) the first component containing an $ne(p_i)$ symbol in each term of the new process expression is a basic component (and so consists of this single $ne(q)$ symbol), and (ii) the prefixes of the new process expression form a subset of the set of prefixes of π_{p_i} and include all the prefixes of π_{p_i} that do not contain any $ne(p_i)$ symbols. For this, we replace the first component of π_{p_i} (lines 1–4 of (2.7)), which is the first component of π_{p_i} containing any $ne(p_i)$ symbols, with the expression,

$$
\begin{aligned}
&[think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) && (1)\\
&\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^* && (2)\\
&\vee [think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) && (3)\\
&\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^* && (4)\\
&\qquad think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ w(p_i.lu)\ ne(p_i) && (5)\\
&\vee [think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) && (6)\\
&\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^* && (7)\\
&\qquad think_i\ w(p_i.ru)\ ne(p_i). && (8)
\end{aligned}
\tag{3.1}
$$

The first term of (3.1) (lines 1–2) is obtained by collecting those terms of the iterated subexpression of π_{p_i} not containing any $ne(p_i)$ symbols (in this case there is only one such term, in lines 1–2 of (2.7)), forming their disjunction, and applying the star operator to this disjunction. Each of the last two terms (lines 3–5 and 6–8 of (3.1)) is obtained by concatenating a term of the iterated subexpression containing an $ne(q)$ symbol (line 3 or line 4 of (2.7)) to the end of the expression obtained for the first term. Strings in the language of (3.1) are thus formed by repeatedly selecting strings corresponding to alternatives in the iterated subexpression of π_{p_i}

not containing any $ne(p_i)$ symbols, or, after an arbitrary number of such selections, eventually selecting a single string corresponding to some alternative in the iterated subexpression containing an $ne(p_i)$ symbol. The expression (3.1) can be substituted for the first component of π_{p_i} because we are only required to keep prefixes of π_{p_i} that do not contain any $ne(p_i)$ symbols, and any prefix of π_{p_i} that is not a prefix of the expression produced by performing this substitution contains at least one $ne(p_i)$ symbol. This substitution essentially pulls those terms of the iterated subexpression containing $ne(p_i)$ symbols (lines 3 and 4 of (2.7)) out of the scope of the enclosing star operator. Replacing the first component of π_{p_i} with (3.1) and putting the resulting expression in disjunctive form in the standard fashion, we obtain the expression,

$$
\begin{aligned}
&[think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) \\
&\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^*\ stop(p_i) \\
&\vee [think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) \\
&\qquad\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^* \\
&\qquad think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ w(p_i.lu)\ ne(p_i)\ stop(p_i) \\
&\vee [think_i\ r(f_{i-1}.u, p_i.ru, ok)\ d(p_i, ok)\ r(f_i.u, p_i.lu, ok) \\
&\qquad\qquad d(p_i, ok)\ eat_i\ u(p_i, ok)\ s(p_i.ld, ok)\ u(p_i, ok)\ s(p_i.rd, ok)]^* \\
&\qquad think_i\ w(p_i.ru)\ ne(p_i)\ stop(p_i).
\end{aligned}
\tag{3.2}
$$

Note that the first component containing an $ne(p_i)$ symbol in every term of this expression is basic and that the prefixes of this expression form a subset of the set of prefixes of π_{p_i} and include all prefixes of π_{p_i} that do not contain any $ne(p_i)$ symbols, as desired.

Next we transform (3.2) into a process expression whose prefixes are precisely those prefixes of π_{p_i} that do not contain any $ne(p_i)$ symbols, and in which every

$stop(p_i)$, $w(p_i.lu)$, and $w(p_i.ru)$ symbol is the last component of a term. We do this by eliminating the "tail" of each term of (3.2), beginning with the first $ne(p_i)$ symbol that appears in the term (i.e., by replacing the first component that contains an $ne(p_i)$ symbol, and so consists of this single $ne(p_i)$ symbol, along with all subsequent components, with λ). This produces the expression,

$$
\begin{aligned}
&[think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, r(f_i.u, p_i.lu, ok) \\
&\quad d(p_i, ok)\, eat_i\, u(p_i, ok)\, s(p_i.ld, ok)\, u(p_i, ok)\, s(p_i.rd, ok)]^*\, stop(p_i) \\
&\vee [think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, r(f_i.u, p_i.lu, ok) \\
&\qquad d(p_i, ok)\, eat_i\, u(p_i, ok)\, s(p_i.ld, ok)\, u(p_i, ok)\, s(p_i.rd, ok)]^* \\
&\qquad think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, w(p_i.lu) \\
&\vee [think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, r(f_i.u, p_i.lu, ok) \\
&\qquad d(p_i, ok)\, eat_i\, u(p_i, ok)\, s(p_i.ld, ok)\, u(p_i, ok)\, s(p_i.rd, ok)]^* \\
&\qquad think_i\, w(p_i.ru).
\end{aligned}
\tag{3.3}
$$

At this point, since every term in (3.3) ends with a $stop(p_i)$, $w(p_i.lu)$, or $w(p_i.ru)$ symbol, the reduction of the process expression π_{p_i} is complete. Clearly, the language of (3.3) consists of precisely those prefixes of π_{p_i} that contain a single $stop(p_i)$, $w(p_i.lu)$, or $w(p_i.ru)$ symbol and do not contain any $ne(p_i)$ symbols, and the last symbol in every string from this language is a $stop(p_i)$, $w(p_i.lu)$, or $w(p_i.ru)$ symbol. The expression (3.3), therefore, is the reduced process expression π'_{p_i}.

In general, not every term of the expression produced in this manner from a process expression π_q that meets the process expression reduction criteria ends with a $stop(q)$ or $w(p)$ symbol. If a term of this expression does not end with a $stop(q)$ or $w(p)$ symbol, however, then it does not contain any $stop(q)$ or $w(p)$ symbols. In such cases, therefore, it is necessary to eliminate such terms to complete

the reduction of the process expression. For example, if the procedure illustrated above is applied to the process expression π_{f_i} for the fork process shown in (2.3), i.e., if the $ne(f_i)$ symbol in the second component of π_{f_i} is pulled out of the scope of the enclosing star operator, the resulting expression is put in disjunctive form in the standard fashion, and then the tail of each term, beginning with the first $ne(f_i)$ component, is eliminated, the expression (3.4) is produced. Of course, the prefixes of this expression are exactly the prefixes of π_{f_i} that do not contain any $ne(f_i)$ symbols, and every $stop(f_i)$ and $w(f_i.d)$ symbol in this expression is the last component of a term. The first term of the expression, however, does not contain any $stop(f_i)$ or $w(f_i.d)$ symbols. The reduced process expression is obtained from (3.4), therefore, by eliminating this term, producing the reduced process expression π'_{f_i} shown in figure 4.

In general, the procedure for reducing a process expression π_q that meets the process expression reduction criteria (i)–(iv) of figure 3 is viewed as consisting of three stages. In the first stage, the process expression is transformed into a process expression in which the first component containing an $ne(q)$ symbol in every term is a basic component, and whose prefixes constitute a subset of the prefixes of π_q and include all prefixes of π_q that do not contain any $ne(q)$ symbols. In the second stage, this process expression is transformed into a process expression in which every $stop(q)$ and $w(p)$ symbol, for $p \in P(q)$, constitutes the last component of a term, and whose prefixes consist of exactly the prefixes of π_q that do not contain any $ne(q)$ symbols. In the third stage, this process expression is transformed into the reduced process expression π'_q. We describe the general algorithms for each of these stages, and informally argue for their correctness below. A more precise description of the algorithms and a proof of their correctness is given in the appendix.

In the first stage of the procedure for reducing the process expression π_q, the subterms containing $ne(q)$ symbols in the first component containing $ne(q)$ symbols in every term of π_q are pulled out of the scope of all star operators. If

$$
\begin{aligned}
& d(f_i, ok)\,[u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)]^* \\
& \vee d(f_i, ok)\,[u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)]^* \\
& \quad u(f_i, ok)\, s(f_i.u, ok)\, w(f_i.d) \\
& \vee d(f_i, ok)\,[u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok) \\
& \quad \vee u(f_i, @)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)]^* \\
& \quad u(f_i, @)\, w(f_i.d)
\end{aligned}
\tag{3.4}
$$

the first component containing any $ne(q)$ symbols in each term of π_q is basic, therefore, this stage is already complete. Otherwise, π_q has at least one iterated subexpression containing an $ne(q)$ symbol in which the first component containing an $ne(q)$ symbol in each term is basic. (This subexpression, of course, may be nested inside other iterated subexpressions.) One of these subexpressions is selected and all terms (of the selected subexpression) containing an $ne(q)$ symbol are pulled out of the scope of the enclosing star operator, as described in the example above. Clearly, no new prefixes are introduced in this step. Furthermore, if a prefix is lost

$$\iota = \prod_{0\le i\le 4} ch(p_i.ld, f_i.d)\, ch(p_i.rd, f_{i-1}.d)\, ch(f_i.u, p_i.lu)\, ch(f_i.u, p_{i+1}.ru)$$
$$\prod_{0\le i\le 4} d(p_i, @)\, d(f_i, @).$$

$$\pi'_{p_i} = \beta^* stop(p_i)$$
$$\vee\, \beta^* think_i\, w(p_i.ru)$$
$$\vee\, \beta^* think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, w(p_i.lu), \quad \text{where}$$

$$\beta = think_i\, r(f_{i-1}.u, p_i.ru, ok)\, d(p_i, ok)\, r(f_i.u, p_i.lu, ok)\, d(p_i, ok)$$
$$eat_i\, u(p_i, ok)\, s(p_i.ld, ok)\, u(p_i, ok)\, s(p_i.rd, ok)$$

$$\pi'_{f_i} = d(f_i, ok)\, \gamma^*\, u(f_i, ok)\, s(f_i.u, ok)\, w(f_i.d)$$
$$\vee\, d(f_i, ok)\, \gamma^*\, u(f_i, @)\, w(f_i.d), \quad \text{where}$$

$$\gamma = u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok)$$
$$\vee\, u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)$$
$$\vee\, u(f_i, @)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok)$$
$$\vee\, u(f_i, @)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)$$

$$\epsilon' = \iota\left(\left(\underset{0\le i\le 4}{\triangle} \pi'_{p_i}\right) \triangle \left(\underset{0\le i\le 4}{\triangle} \pi'_{f_i}\right)\right)$$

Figure 4

System expression for a reduced constrained expression representation of a solution to the dining philosophers problem

in this step, it contains a substring corresponding to a term containing an $ne(q)$ symbol of the selected subexpression. As this term has a basic component consisting of an $ne(q)$ symbol, the prefix contains an $ne(q)$ symbol. This step is repeated until the first $ne(q)$ symbol in each term of the resulting process expression does not lie within the scope of any star operators, a state that must eventually be reached since each time the step is repeated the number of star operators that contain a particular $ne(q)$ symbol within their scope decreases, for some of the $ne(q)$ symbols. At this point the process expression π_q has been transformed into a process expression satisfying the properties desired for the first stage, and so this stage is complete.

In the second stage of the procedure for reducing the process expression π_q, the tail of each term of the expression produced in the first stage, beginning with the first component containing an $ne(q)$ symbol, is eliminated. Clearly, properties of the expression produced in the first stage imply that the prefixes of the expression obtained in this manner are exactly the prefixes of π_q that do not contain any $ne(q)$ symbols. To see that every $stop(q)$ and $w(p)$ symbol, for $p \in P(q)$, in this expression is the last component of a term, we use the process expression reduction criteria (iii) and (iv). These properties are not affected by pulling terms out of the scope of star operators, and so the expression produced in the first stage also satisfies these properties. When the appropriate tails are dropped, therefore, terms of this expression in which an $ne(q)$ symbol appears before any $stop(q)$ symbols or $w(p)$ symbols, for $p \in P(q)$, produce terms that do not contain any $stop(q)$ or $w(p)$ symbols, while terms of this expression in which a $w(p)$ symbol, for some $p \in P(q)$, appears before any $stop(q)$ symbols or $ne(q)$ symbols, produce terms that end with a $w(p)$ symbol and contain no other $stop(q)$ or $w(p)$ symbols (since the $w(p)$ symbol is immediately followed by an $ne(q)$ symbol), and finally, terms of this expression in which a $stop(q)$ symbol appears before any $ne(q)$ symbols or $w(p)$ symbols, for any $p \in P(q)$, produce terms that end with a $stop(q)$ symbol

and contain no other $stop(q)$ or $w(p)$ symbols (since the $stop(q)$ symbol is the last component of the term). Every $stop(q)$ and $w(p)$ symbol, for $p \in P(q)$, in the resulting process expression, therefore, is the last component of a term, as desired.

In the third stage, the regular expression obtained in the second stage is transformed into the reduced process expression π'_q by eliminating terms that do not end with a $stop(q)$ or $w(p)$ symbol, for any $p \in P(q)$. The properties of the expression produced in the second stage clearly imply that the language of the resulting regular expression consists of exactly those prefixes of π_q that contain a single $stop(q)$ or $w(p)$ symbol, for some $p \in P(q)$, and do not contain any $ne(q)$ symbols, and that the single $stop(q)$ or $w(p)$ symbol appearing in every string from the language of this regular expression is the last symbol in the string.

The relationship between the given process expression π_q and the reduced process expression π'_q obtained in this fashion is summarized in the following proposition, which is stated more formally and proven in the appendix.

(3.5) **Proposition.** *If the process expression $\pi_q \in \mathcal{RE}(A)$ meets the process expression reduction criteria (i)–(iv) of figure 3, and if $\pi'_q \in \mathcal{RE}(A)$ is the reduced process expression obtained from π_q as described above (or, equivalently, as described in the appendix) then*

(i) $\pi'_q \in \mathcal{RE}(A_q)$,

(ii) $\mathcal{P}(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = \mathcal{L}(\pi'_q)$, and

(iii) $\mathcal{P}(\pi'_q)|_{\kappa_3(q)} = \mathcal{L}(\pi'_q)$.

Property (i) above simply assures that no new symbols are introduced when simplifying the process expression.

We have said that the system expression obtained from the reduced process expressions can be used to replace the system expression of the given constrained expression, and that when this is done, the resulting constrained expression representation of Σ is reduced. The theorem below shows this to be the case.

(3.6) **Theorem.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be the constrained expression representation of an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$. *If*

(i) $\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q\right)$, *where* ι *is the initial expression derived from the design of* Σ *and the process expressions* π_q, *for* $q \in Q$, *meet the process expression reduction criteria (i)–(iv) of figure 3,*

(ii) $\kappa_3(q), \kappa_4(q) \in \widehat{C}$, *for* $q \in Q$, *and*

(iii) π'_q, *for* $q \in Q$, *is the reduced process expression obtained from* π_q *as described above, then*

then $\mathbf{D} \approx (\mathbf{F}, \epsilon')$, *where* $\epsilon' = \iota\left(\underset{q \in Q}{\triangle} \pi'_q\right)$. *The constrained expression representation* $\mathbf{D}' = (\mathbf{F}, \epsilon')$ *for* Σ, *furthermore, is reduced.*

Proof. To see that $\mathbf{D} \approx (\mathbf{F}, \epsilon')$, first observe that proposition (3.5) implies $\mathcal{P}(\pi'_q)|_{\{\kappa_3(q), \kappa_4(q)\}} = \mathcal{L}(\pi'_q)|_{\kappa_4(q)} = \mathcal{L}(\pi'_q) = \mathcal{P}(\pi_q)|_{\{\kappa_3(q), \kappa_4(q)\}}$. For each $q \in Q$, therefore, proposition (V.2.5) can be applied, with $\pi = \pi_q$, η equal to the interleave of the process expressions for the all the processes except the process q, and $\widehat{D} = \{\kappa_3(q), \kappa_4(q)\}$, to show that π_q can be replaced with the reduced process expression π'_q, and thus, that $\mathbf{D} \approx (\mathbf{F}, \epsilon')$.

To see that the constrained expression $(\mathbf{F}, \epsilon')$ is reduced, consider a constrained prefix $u \in \mathcal{P}(\epsilon')|_{\widehat{C}}$. Since u satisfies the constraints $\kappa_3(q)$, for $q \in Q$, and the initial expression ι does not contain any $stop(q)$ or $w(p)$ symbols, for $q \in Q$ or $p \in P$, we reason that u belongs to $\mathcal{L}\left(\iota\left(\underset{q \in Q}{\triangle} u_q\right)\right)$, for some $u_q \in \mathcal{P}(\pi'_q)$. Now, for each $q \in Q$, the equality $\rho_{S_3(q)}(u) = \rho_{S_3(q)}(u_q)$ follows from (i) of (3.5) and $\rho_{S_3(q)}(\iota) = \lambda$. As u satisfies the constraint $\kappa_3(q)$, we therefore conclude that u_q does also. Hypothesis (iii) thus implies that $u_q \in \mathcal{L}(\pi'_q)$, for each $q \in Q$. Hence, every constrained prefix of $\mathbf{D}'$ is formed by concatenating some interleaving of full strings from the languages of the reduced process expressions to the end of the initial expression, and so is a full string from the language of ϵ', as desired. ∎

Finally, notice that the constraints $\kappa_4(q)$, for $q \in Q$, are not required in

the constrained expression $(\mathbf{F}, \epsilon')$ in this proposition, since the system expression ϵ' does not contain any $ne(q)$ symbols, for $q \in Q$. In general, therefore, a reduced constrained expression representation for an SDYMOL system Σ is obtained as follows. The constrained expression representation for Σ described in (1.11) is generated and the process expressions in the system expression of this constrained expression are put in disjunctive form in the standard fashion. At this point, the process expressions in this constrained expression representation of Σ are ready to be reduced. In certain cases, however, they may be simplified further, as described in the previous section, before being reduced. The simplified constrained expression representation of Σ is then reduced, by reducing the process expressions in the system expression as described above, and the constraints $\kappa_4(q)$, for $q \in Q$, are eliminated from the constraining context. A reduced constrained expression representation for the solution to the dining philosophers problem presented in figures III.1 and III.3, for example, is obtained using the system expression of figure 4 and all the constraints of figure 2, except $\kappa_4(f_i)$ and $\kappa_4(p_i)$, for $0 \leq i \leq 4$.

CHAPTER VII

MESSAGE FLOW ANALYSIS

Once a reduced constrained expression representation for an SDYMOL system has been obtained using the techniques of chapter VI, the *message flow analysis* algorithms described in this chapter can be applied to further simplify this representation. These algorithms are applied to the process expressions in the system expression of a reduced constrained expression representation for an SDYMOL system to produce simpler process expressions that can be used in place of the original process expressions. They resemble standard data flow analysis algorithms used in compilers for global optimization of programs and to detect certain data flow anomalies in programs [FOSD76, HECH77].

Message flow analysis depends on graphical representations of the languages of the process expressions for the processes in the system, in much the same way as data flow analysis depends on graphical representations for the flow of control of programs. These graphs are analyzed to identify nodes that, because of restrictions imposed by the SDYMOL language semantics on the manner in which messages can flow through a system, can be "safely" pruned from the graphs. A formal basis for this analysis is provided by the constraints $\kappa_2(q)$, for $q \in Q$, which assure that the buffers of the processes are used and modified in a consistent fashion, and the constraints $\kappa_5(l, m)$, for $l \in L$ and $m \neq \mathbb{Q}$, which assure that messages are received from links only if they are available to be received. The simpler, pruned graphs represent the languages of simpler regular expressions, and these regular expressions can be used in place of the original process expressions in the constrained expression representation of the system.

We begin a more precise explanation of this analysis by introducing some concepts from graph theory, along with the terminology that is used in the remain-

der of the chapter. We then describe the flow graph for a process expression and establish a formal basis for pruning "unattainable" nodes of a flow graph. Finally, the message flow algorithms used to detect such nodes are described.

§1. Necessary concepts from graph theory

The analysis described in the following sections relies on graphs that model the activities of the processes in an SDYMOL system. Formally, a (directed) graph is defined as follows.

(1.1) *Definition.* A *graph* is a pair $G = (N, E)$, where N is a set of *nodes* and E is a set of ordered pair of nodes, called *edges.*

We represent a graph pictorially, using labeled circles to represent nodes and directed arcs to represent edges. For example, the graph $G = (N, E)$, where $N = \{0, 1, 2, 3, 4\}$ and $E = \{(0,1), (0,2), (2,3)(3,2)(2,4)\}$, is shown in figure 1. When there is no need to distinguish the nodes in a graph, we omit the labels inside the nodes and represent the nodes with points instead of circles.

An edge $(n_1, n_2) \in E$ in a graph $G = (N, E)$ is said to *leave* node n_1 and *enter* node n_2. If there are no edges entering a node in a graph, the node is a *root* node. If there are no edges leaving a node, the node is a *terminal* node. In general, a graph may have any number of root and terminal nodes (including none at all). The graph in figure 1, for example, has a single root node (node 0) and two terminal nodes (nodes 1 and 4). Nodes that are neither root nodes nor terminal nodes are called *internal nodes.*

A *path* in a graph is a sequence of nodes $(n_1, n_2, \ldots, n_k)$ such that there is an edge leaving n_i and entering n_{i+1}, for $1 \leq i \leq k-1$. We say that the path $(n_1, \ldots, n_k)$ goes *from* the node n_1, *to* the node n_k, and *passes through* the nodes n_i, for $1 \leq i \leq k$. The path is a *first-arrival* path if $n_k \neq n_i$, for $1 \leq i < k$.

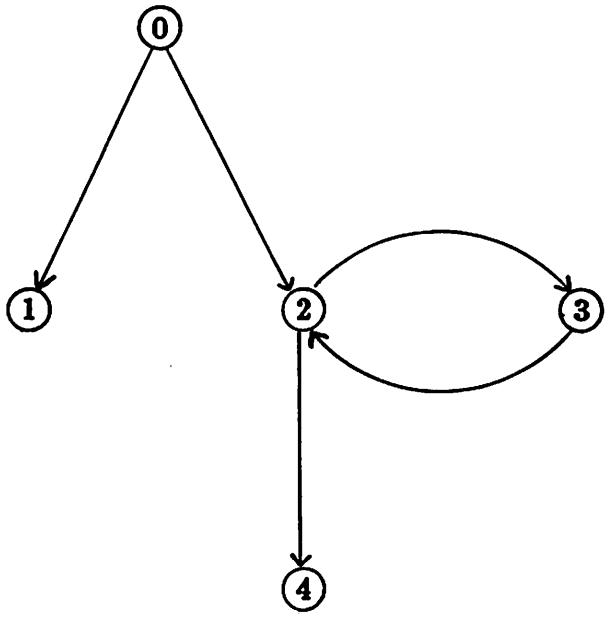


Figure 1

Pictorial representation of a graph

It is *rooted* if n_1 is a root node. It is *complete* if n_1 is a root node and n_k is a terminal node. The path $(0,2,3,2)$ from node 0 to node 2 in the graph of figure 1 is rooted, but it is not complete. It is also not a first-arrival path. The path $(0,1)$, on the other hand, is complete, and hence, it is both a rooted path and a first-arrival path, as is the path $(0,2,3,2,4)$.

A *node labeling* for a graph $G = (N,E)$ is a function from N to a set L of node labels, $l: N \rightarrow L$. If $l(n) = a$, where $n \in N$ and $a \in L$, we call n an *a-node.* An *edge labeling* is a function from E to a set of edge labels. We write the label for a node in a graph beside the circle (or point) representing the node, and the label for an edge beside the arc representing the edge. Figure 2 shows node and edge labeling functions for the graph of figure 1. The node labels belong to the set $L = \{a,b,c\}$ and the edge labels are subsets of L. The root node and terminal nodes are a-nodes. One internal node is a b-node and the other is a c-node. Both

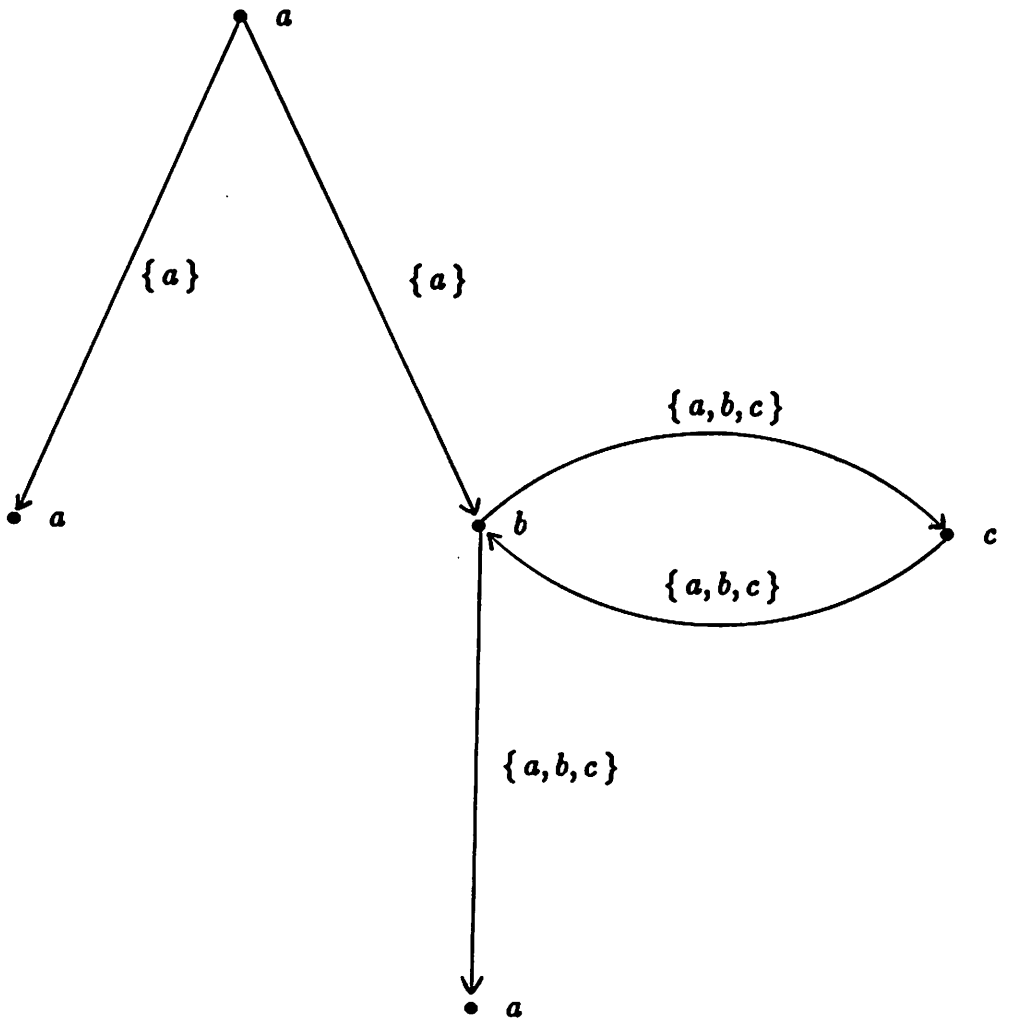


Figure 2

Node and edge labelings for the graph in figure 1

edges leaving the root node are labeled $\{a\}$. The remaining edges are labeled $\{a, b, c\}$.

Graphs can be modified in a number of different ways. Distinct nodes n_1 and n_2 in a graph $G = (N, E)$ can be *identified*, or collapsed into a a single node. The identification of n_1 and n_2 does not affect the edges of G that do not contain n_1 or n_2. If there is an edge from a node n_3 to either n_1 or n_2 in G, then there is an edge from n_3 to the collapsed node when n_1 and n_2 are identified. Similarly, if there is an edge from either n_1 or n_2 to a node n_3 in G, then there is an edge from the collapsed node to n_3 when n_1 and n_2 are identified. The technique of identifying nodes can also be used to combine two graphs into a single, composite graph. The graphs in figure 3, for example, when combined by identifying the internal node of G_1 with either of the nodes of G_2, produce the graph of figure 1. When identifying labeled nodes, a label for the collapsed node must be explicitly specified, unless the identified nodes have the same label, in which case the label of the collapsed node, unless otherwise specified, is assumed to be the same as the label of the identified nodes.

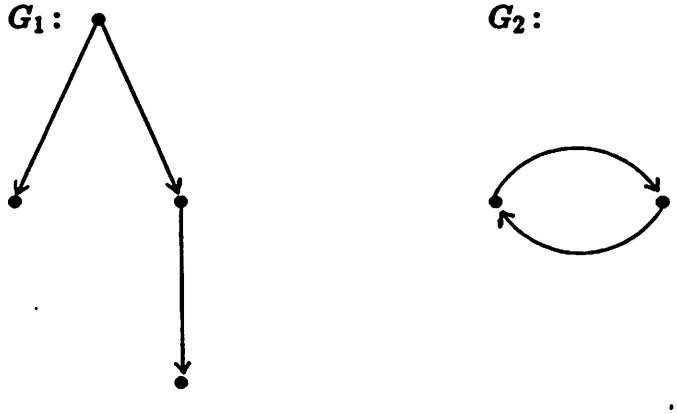


Figure 3

Graphs that combine to produce the graph of figure 1

A graph can also be modified by *pruning* some of its nodes. To prune a node n from a graph $G = (N, E)$, we delete n from N and delete all edges that enter or leave n from E.

§2. Flow graphs and unattainable nodes

If the nodes in a graph are labeled with symbols from some alphabet of event symbols, or with the empty string or empty regular expression, each complete path in the graph can be viewed as representing the string over this alphabet obtained by concatenating the labels of the nodes encountered along the path, or, if the path passes through an $\emptyset$-node, as representing the empty set. The graph thus represents a language determined by its set of complete paths and the labeling of its nodes. When we represent the language of a regular expression by a graph in this manner, the graph is called a *flow graph*

The flow graph for a regular expression is required to have a unique root node and a unique terminal node, both of which must be λ-nodes. Each node in the graph, furthermore, must lie along some complete path. The alphabet of the regular expression is used for labeling the nodes, and the language determined by the graph, of course, must coincide with the language of the regular expression.

(2.1) *Definition.* A *flow graph* for a regular expression $\alpha \in \mathcal{RE}(A)$ is a graph $G = (N, E)$ with a node labeling function $l: N \to A \cup \{\lambda, \emptyset\}$ such that

(i) N contains a single root node n_r and a single terminal node n_t,

(ii) $l(n_r) = l(n_t) = \lambda$,

(iii) for every $n \in N$, there is a complete path in G that passes through n, and

(iv) $u \in \mathcal{L}(\alpha)$ if and only if there exists some complete path $(n_1, \ldots, n_k)$ in G such that $u = l(n_1) \cdots l(n_k)$.

The next definition shows how a flow graph can be constructed for any regular expression that does not involve the shuffle operator. (This construction is similar

to the construction of a finite state automaton representing the language of a regular expression described in [BARR79, p.105].)

(2.2) *Definition.* Given an alphabet of event symbols A, we define a map G, which associates each regular expression $\alpha \in \mathcal{RE}(A)$ that does not involve the shuffle operator with a flow graph $G(\alpha)$, as follows.

(i) $G(\lambda)$ consists of a root node, an internal node and a terminal node, all of which are λ-nodes.

(ii) $G(\emptyset)$ consists of a root node, an internal node and a terminal node; the root and terminal nodes are λ-nodes and the internal node is an $\emptyset$-node.

(iii) $G(a)$, for $a \in A$, consists of a root node, an internal node and a terminal node; the root and terminal nodes are λ-nodes and the internal node is an a-node.

(iv) $G(\alpha\beta)$ is obtained from $G(\alpha)$ and $G(\beta)$ by identifying the terminal node of $G(\alpha)$ with the root node of $G(\beta)$.

(v) $G(\alpha \vee \beta)$ is obtained from $G(\alpha)$ and $G(\beta)$ by identifying the root node of $G(\alpha)$ with the root node of $G(\beta)$ and the terminal node of $G(\alpha)$ with the terminal node of $G(\beta)$.

(vi) $G(\alpha^*)$ is obtained from $G(\alpha)$ and $G(\lambda)$ by identifying the root node of $G(\alpha)$ with the internal node of $G(\lambda)$, and then adding an edge from the terminal node of $G(\alpha)$ to the identified node.

These rules are summarized in figure 4. A routine induction argument shows that $G(\alpha)$ is a flow graph for the regular expression α.

A regular expression that does involve the shuffle operator, of course, can be transformed into a regular expression that does not, and a flow graph obtained from this regular expression using the construction described above. Because the message flow analysis algorithms are applied to process expressions, which do not require the shuffle operator, we restrict our attention in the remainder of this chapter to regular expressions that do not involve the shuffle operator.

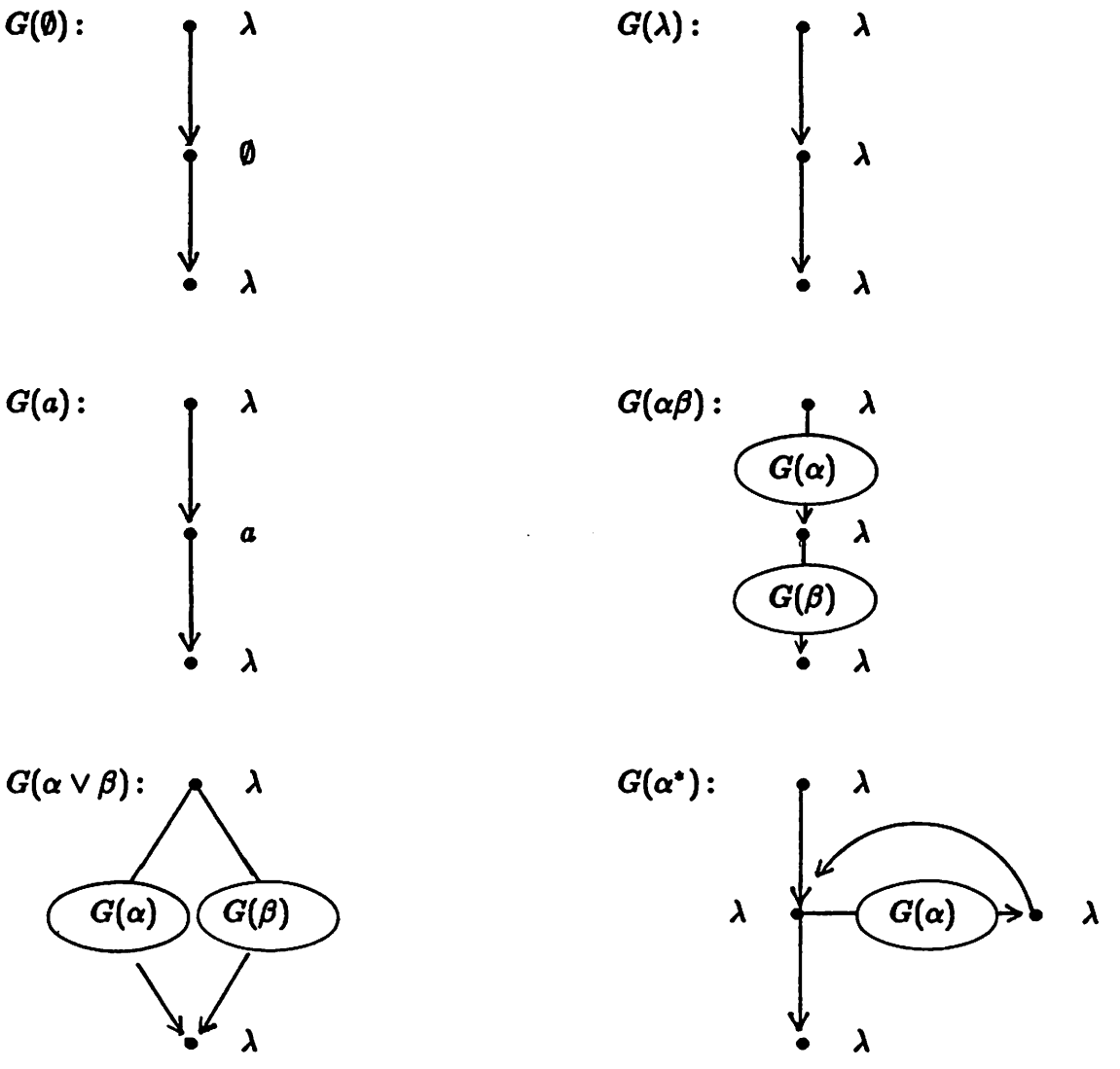


Figure 4

Rules for generating a flow graph from a regular expression that does not involve the shuffle operator

The map G can be applied to the process expressions $\pi_q \in \mathcal{RE}(A_q)$ for an SDYMOL system Σ to produce the flow graphs required for message flow analysis. It is important for this analysis to observe that every a-node in the flow graph $G(\pi_q)$, for $q \in Q$ and $a \in A_q$, corresponds to a subexpression of the process expression π_q composed of the single event symbol a. Consider, for example, a process expression,

$$\pi_q = d(q,1)\Big(u(q,1)\,d(q,2) \vee u(q,2)\,d(q,3) \vee u(q,4)\,d(q,4)\Big)^* \Big(u(q,2) \vee u(q,4)\Big)\,stop(q) \tag{2.3}$$

The flow graph $G(\pi_q)$ is shown in figure 5. The complete path $(0,1,2,3,14,15,17,18,19)$ represents the string $d(q,1)\,u(q,2)\,stop(q)$, which belongs to the language of π_q. The single $d(q,1)$-node in $G(\pi_q)$ corresponds to the single $d(q,1)$ subexpression of π_q. One of the $u(q,2)$-nodes, node 7, corresponds to the first $u(q,2)$ subexpression of π_q, while the other $u(q,2)$-node, node 15, corresponds to the second $u(q,2)$ subexpression of π_q.

Clearly, many of the λ-nodes in the flow graph of figure 5 are unnecessary. The flow graph of figure 6, in which some of the λ-nodes in $G(\pi_q)$ have been identified with other nodes, determines the same language as $G(\pi_q)$. (We retain the λ-nodes 3, 13 and 14 because they make the diagram easier to interpret.) The string $d(q,1)\,u(q,2)\,stop(q)$, for instance, is represented by the path $(0,1,3,14,15,18,19)$ in this graph. As in the graph $G(\pi_q)$, each a-node, for $a \in A_q$, in the flow graph of figure 6 corresponds to a subexpression consisting of the event symbol a. To facilitate the description that follows, we assume that the flow graph for a process expression is obtained using the map G defined in (2.2). Message flow analysis can be performed, however, using flow graphs for the process expressions π_q, for $q \in Q$, obtained from the flow graphs $G(\pi_q)$ by identifying λ-nodes with adjacent nodes having different labels. In practice, therefore, we usually simplify the flow graph produced using G before performing the analysis.

The message flow analysis algorithms described in the next section detect

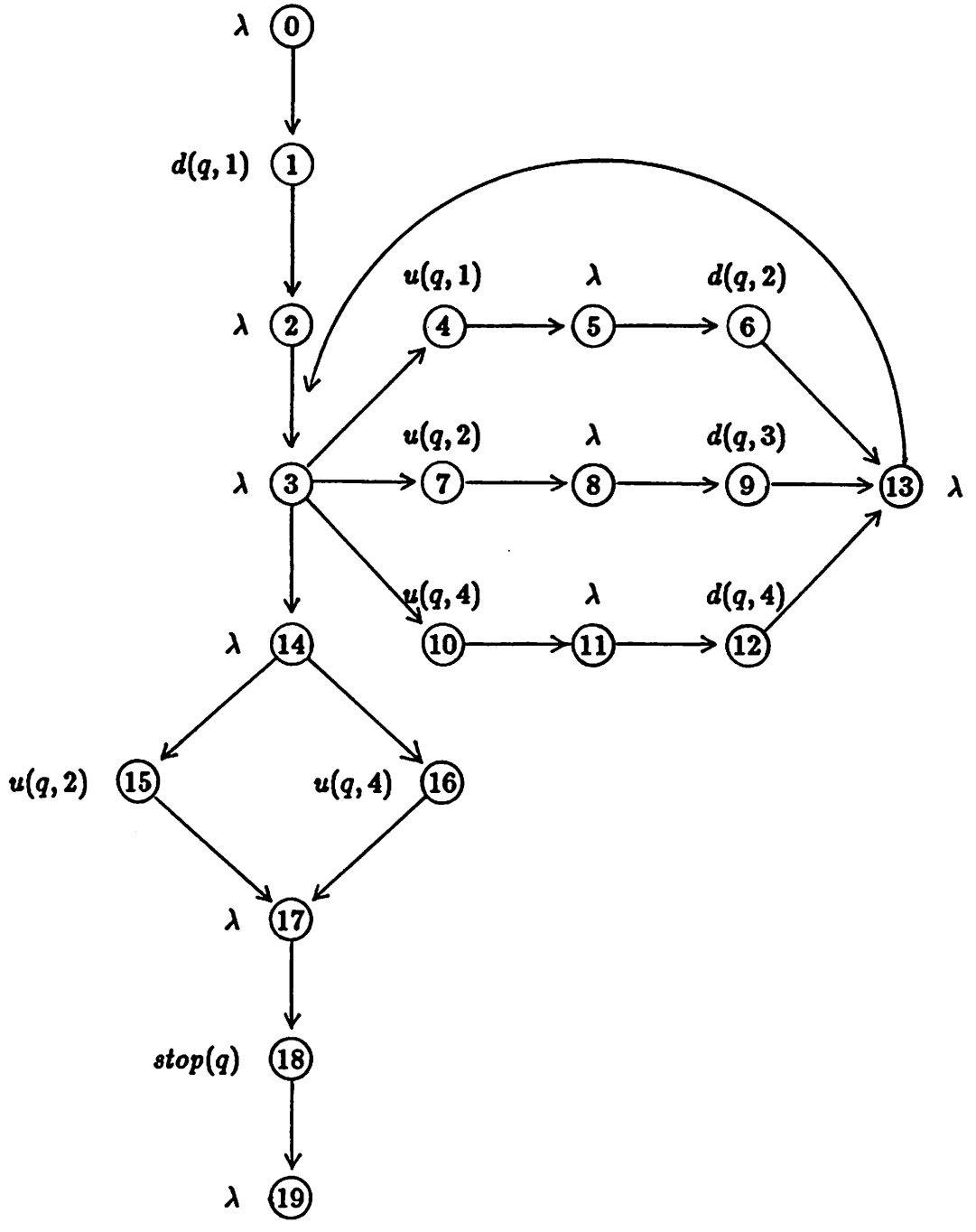


Figure 5

Flow graph representing the language of the expression (2.3)

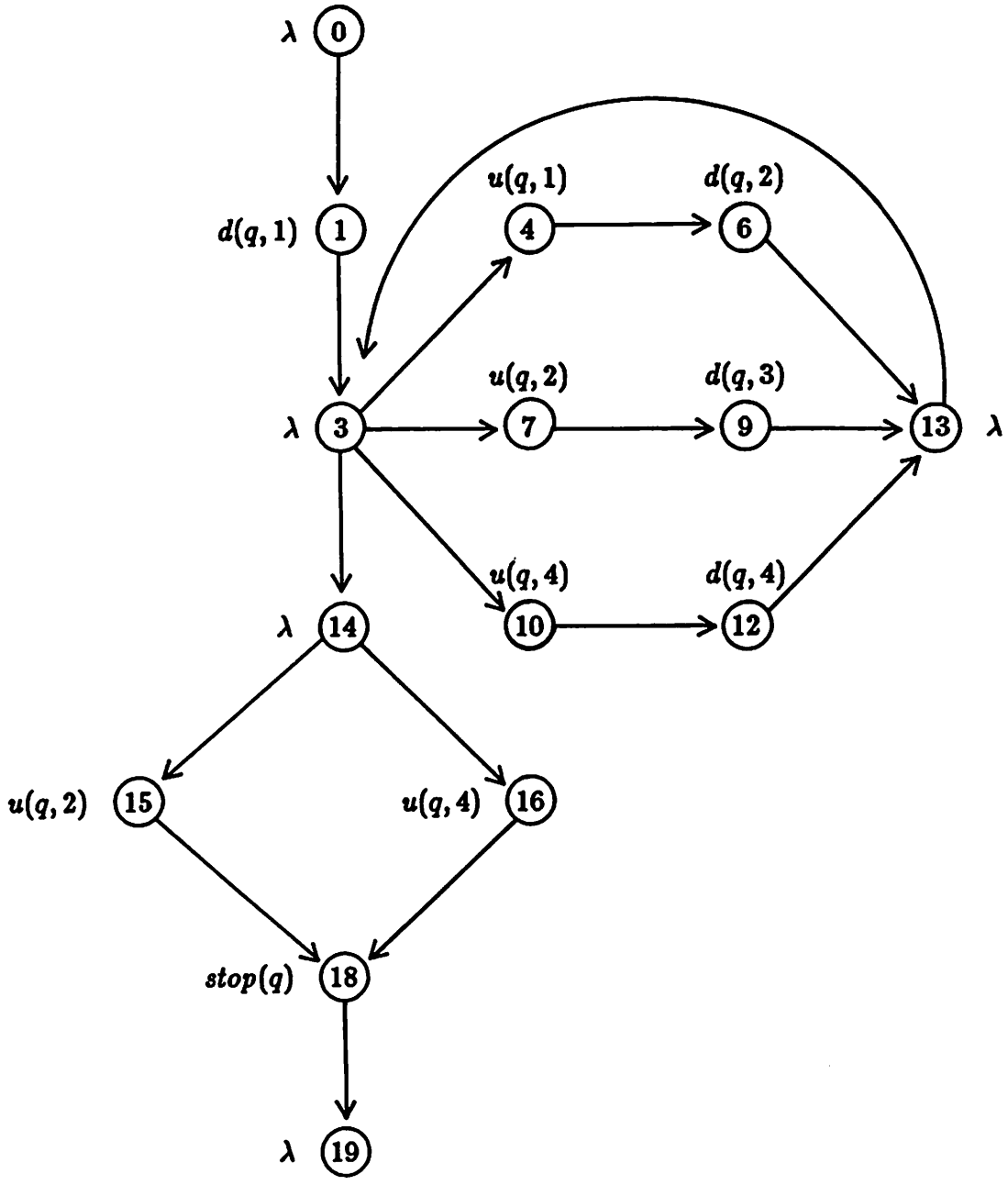


Figure 6

Flow graph obtained from figure 6
by collapsing unnecessary λ-nodes

nodes in the flow graphs for the process expressions π_q, for $q \in Q$, that, because of certain of the SDYMOL constraints, are not required for generating legal behavioral traces of the system. Intuitively, if a flow graph can be obtained by pruning such nodes from the flow graph for a process expression, then the pruned flow graph corresponds to a simpler regular expression that can be used in place of the original process expression. We describe and justify this approach more precisely below.

(2.4) *Definition.* Let $\mathbf{D} = (\mathbf{F}, \epsilon)$ be a reduced constrained expression representation for an SDYMOL system Σ, where $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\Big(\underset{q \in Q}{\triangle} \pi_q\Big)$, ι is the initial expression derived from the design of Σ, and the process expressions $\pi_q \in \mathcal{RE}(A_q)$, for $q \in Q$, do not involve the shuffle operator. A node in a flow graph for a process expression π_q, for $q \in Q$, is *attainable* if there is a string $w \in A^*$ such that $\iota w \in \mathcal{L}(\epsilon)|_{\widehat{C}}$ and $\rho_{A_q}(w)$ is represented by some path passing through the designated node. Otherwise, the node is *unattainable.*

The algorithms described in the next section detect nodes in the flow graph of a process expression that are unattainable. Such nodes are not required for generating legal behavioral traces of a process, and so we would like to prune them from the flow graph and replace the original process expression with a regular expression whose language is the same as the language of the pruned flow graph. In general, however, it is not sufficient to simply prune the unattainable nodes that are detected by these algorithms, since the graph obtained in this manner may not be a flow graph. If every complete path through a given node contains an unattainable node, then the given node must also be pruned. When a flow graph can be obtained by pruning such nodes, a procedure must then be devised for associating the flow graph with the regular expression that it represents. While this approach is not difficult to apply in practice, rigorous descriptions of the general algorithms and proofs of their correcteness are fairly long and complex. We therefore pursue a slightly different approach, which has the same effect as the approach described

above, but is easier to formally characterize and prove correct.

The unattainable nodes in the flow graph for a process expression that are detected by the algorithms described in the next section are all labeled with symbols from the alphabet of the process expression. Thus, if the flow graph is generated using the transformation G defined in (2.2) (or by identifying some of the λ-nodes in this flow graph with adjacent nodes), then each node corresponds to some event symbol in the process expression. Instead of pruning unattainable nodes from the flow graph, we relabel them, making them $\emptyset$-nodes. The resulting graph is clearly a flow graph. In fact, a regular expression that determines the same language as the relabeled graph is obtained from the original process expression by simply replacing the event symbols corresponding to the relabeled nodes with the empty regular expression. Changing a node into an $\emptyset$-node, furthermore, has the same effect on the language of the graph as pruning the node, along with all nodes that only appear along complete paths passing through the node. (A fact that is formally assured by the regular expression identities: $\emptyset \vee a = a \vee \emptyset = a$ and $\emptyset a = a\emptyset = \emptyset$.)

To formalize this argument, we first observe that if an a-node in the flow graph $G(\pi_q)$, for $q \in Q$ and $a \in A_q$, is relabeled with $\emptyset$, a regular expression that determines the same language as the resulting flow graph can be obtained from π_q by simply replacing the subexpression corresponding to the designated a-node with $\emptyset$. This fact is obvious given the definition of G.

We then show that subexpressions of a process expression π_q corresponding to unattainable nodes of the flow graph $G(\pi_q)$ can be replaced with the empty regular expression.

(2.5) **Proposition.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be a reduced constrained expression representation for an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q\right)$, ι *is the initial expression derived from the design of* Σ, *and the process expressions* $\pi_q \in \mathcal{RE}(A_q)$, *for* $q \in Q$, *do not involve the shuffle operator, and let* n *be an unattainable node in the flow graph* $G(\pi_q)$, *for some* $q \in Q$. *If* π'_q *is the regular*

expression obtained from π_q *by replacing the subexpression corresponding to* n *with the empty regular expression, then* π_q' *can be used in place of the process expression* π_q *in the system expression of* $\mathbf{D}$.

Proof. Let ϵ' denote the regular expression obtained from ϵ by replacing the regular expression π_q with π_q'. We show that $\mathbf{D}' = (\mathbf{F}, \epsilon')$ is strongly equivalent to $\mathbf{D}$ and is also reduced by showing that $\mathcal{L}(\epsilon)|_{\widehat{C}} = \mathcal{L}(\epsilon')|_{\widehat{C}} = P(\epsilon')|_{\widehat{C}}$.

First note that $\mathcal{L}(\epsilon') \subseteq \mathcal{L}(\epsilon)$, and so we have $\mathcal{L}(\epsilon')|_{\widehat{C}} \subseteq \mathcal{L}(\epsilon)|_{\widehat{C}}$ trivially. To establish the reverse inclusion, we choose a string $u \in \mathcal{L}(\epsilon)$ such that $u \notin \mathcal{L}(\epsilon')$ and show that u does not satisfy the constraints of $\widehat{C}$. Clearly, $u \in \mathcal{L}(\epsilon)$ and $u \notin \mathcal{L}(\epsilon')$ implies that $u = \iota w$, for some $w \in A^*$ such that $\rho_{A_q}(w) \in \mathcal{L}(\pi_q)$, but $\rho_{A_q}(w) \notin \mathcal{L}(\pi_q')$. Now $\rho_{A_q}(w) \in \mathcal{L}(\pi_q)$ implies that there is some complete path $(n_1, \ldots, n_k)$ in $G(\pi_q)$ that represents this string. Since $(n_1, \ldots, n_k)$ is also a complete path in $G(\pi_q')$ and $\rho_{A_q}(w) \notin \mathcal{L}(\pi_q')$, we conclude that, in $G(\pi_q')$, this path represents the empty set. But the node labeling functions for $G(\pi_q)$ and $G(\pi_q')$ are identical, except on the node n. We therefore conclude that $n = n_i$, for some $1 \leq i \leq k$. By hypothesis, however, n in unattainable in $G(\pi_q)$. Hence, $u = \iota w \notin \mathcal{L}(\epsilon)|_{\widehat{C}}$, as desired.

Having shown that $\mathcal{L}(\epsilon)|_{\widehat{C}} = \mathcal{L}(\epsilon')|_{\widehat{C}}$ and having observed that $\mathcal{L}(\epsilon') \subseteq \mathcal{L}(\epsilon)$, we reason that $P(\epsilon')|_{\widehat{C}} \subseteq P(\epsilon)|_{\widehat{C}} = \mathcal{L}(\epsilon)|_{\widehat{C}} = \mathcal{L}(\epsilon')|_{\widehat{C}} \subseteq P(\epsilon')|_{\widehat{C}}$, which completes the proof. ∎

§3. Algorithms for detecting unattainable nodes

We now turn our attention to the detection of unattainable nodes. Two message flow analysis algorithms are described. Both algorithms can be routinely applied to the flow graphs $G(\pi_q)$ of the process expressions π_q in the system expression of a reduced constrained expression representation for a system Σ to

detect certain classes of unattainable nodes. The second algorithm is more powerful than the first, as the unattainable nodes detected using the first algorithm are all detected using the second algorithm, and, in general, the second algorithm detects unattainable nodes that are not detected by applying the first algorithm. The first algorithm, however, is easier to describe and apply than the second.

The first of the message flow analysis algorithms is applied to the process expressions π_q in the system expression of a reduced constrained expression representation for a system on an individual basis. The underlying idea behind this algorithm can be explained as follows. For a given process $q \in Q$, the constraint $\kappa_2(q)$ assures that the buffer of the process q is used and modified in a consistent fashion by filtering out prefixes of the system expression in which a $d(q,m')$ symbol is followed by a $u(q,m)$ symbol, with no intervening $d(q,m)$ symbols, for some message types $m' \neq m$. If, for every rooted, first-arrival path to some $u(q,m)$-node, the last node representing a modification of the process' buffer along the path is labeled $d(q,m')$, where $m' \neq m$, the constraint $\kappa_2(q)$ thus assures that this $u(q,m)$-node is unattainable. Any node that can only be reached by passing through such nodes, of course, is also unattainable.

The first message flow analysis algorithm detects nodes in a flow graph for a process expression π_q that are unattainable for the reasons described above. This algorithm labels the edges of the flow graph with subsets of M so that, for each node n in the flow graph, if there is a rooted, first-arrival path to n that does not pass through any unattainable nodes and in which the last node representing a modification of the process' buffer is labeled $d(q,m')$, then the label on the last edge in this path (i.e., the edge entering n) contains the message type m'. If the message type m, for some $m \in M$, is not contained in any of the labels of the edges entering a $u(q,m)$-node, therefore, the node is unattainable.

The first message flow analysis algorithm is defined in figure 7. An initial labeling for all the edges of the graph is provided in step (1). The remainder of the

algorithm consists of repeatedly modifying this labeling, by performing any of the moves (a), (b), or (c), until no further moves can be made or no move changes the label of any edge in the graph. It is clear that this algorithm terminates, since a move always adds messages to the label of an edge and there are a finite number of edges in the graph and a finite number of message types in M. The order in which moves are performed, furthermore, does not affect the final labeling. This is because, if it is possible to perform some move that results in a message of type m being added to a particular edge, the move remains possible and results in m being added to the edge even if another move is performed first.

The edge labeling obtained by applying the first message flow analysis algorithm to the flow graph of figure 6 is shown in figure 8. A particular sequence of moves that produces this labeling is summarized in figure 9.

If no series of moves results in the message type m being added to the label of an edge entering a $u(q,m)$-node, for some $m \in M$, then clearly no move involving this node can ever be performed. Hence, all edges leaving the node are labeled with $\emptyset$ by the first message flow analysis algorithm. Similarly, the edges leaving nodes that can only be reached by first passing through this node are all labeled with $\emptyset$. We have argued informally that such nodes are unattainable. The next proposition formally establishes this result.

(3.1) **Proposition.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be a reduced constrained expression representation for an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\Big(\underset{q \in Q}{\triangle} \pi_q\Big)$, ι *is the initial expression derived from the design of* Σ, *and the process expressions* $\pi_q \in \mathcal{RE}(A_q)$, *for* $q \in Q$, *do not involve the shuffle operator, and let* $G = (N, E)$, *with node labeling function* l, *be a flow graph for the process expression* π_q, *for some* $q \in Q$. *If* $\kappa_2(q) \in \widehat{C}$ *and if, after applying the first message flow analysis algorithm to* G, *every edge leaving a node* $n \in N$ *is labeled with* $\emptyset$, *then* n *is unattainable.*

Input: A flow graph G for a process expression π_q.

Algorithm:

(1) Label all edges leaving the root node of G with $\{\Omega\}$ and label all the remaining edges in G with $\emptyset$.

(2) Perform legal moves (described below) until either no more moves can be made or no legal move changes the label of any edge in G.

Legal moves: Given a node n in G, an edge entering n labeled L and an edge leaving n labeled L', the following are the legal moves.

(a) If $L \neq \emptyset$ and n is a $d(q,m)$-node, then add m to L' (i.e., change the label on the edge leaving n to $L' \cup \{m\}$).

(b) If $L \neq \emptyset$, n is a $u(q,m)$-node, and $m \in L$, then add m to L'.

(c) If $L \neq \emptyset$, n is not a $d(q,m)$-node or a $u(q,m)$-node, for any $m \in M$, and n is not an $\emptyset$-node, then add all the elements of L to L' (i.e., change the label on the edge leaving n to $L \cup L'$).

Figure 7

The first message flow analysis algorithm

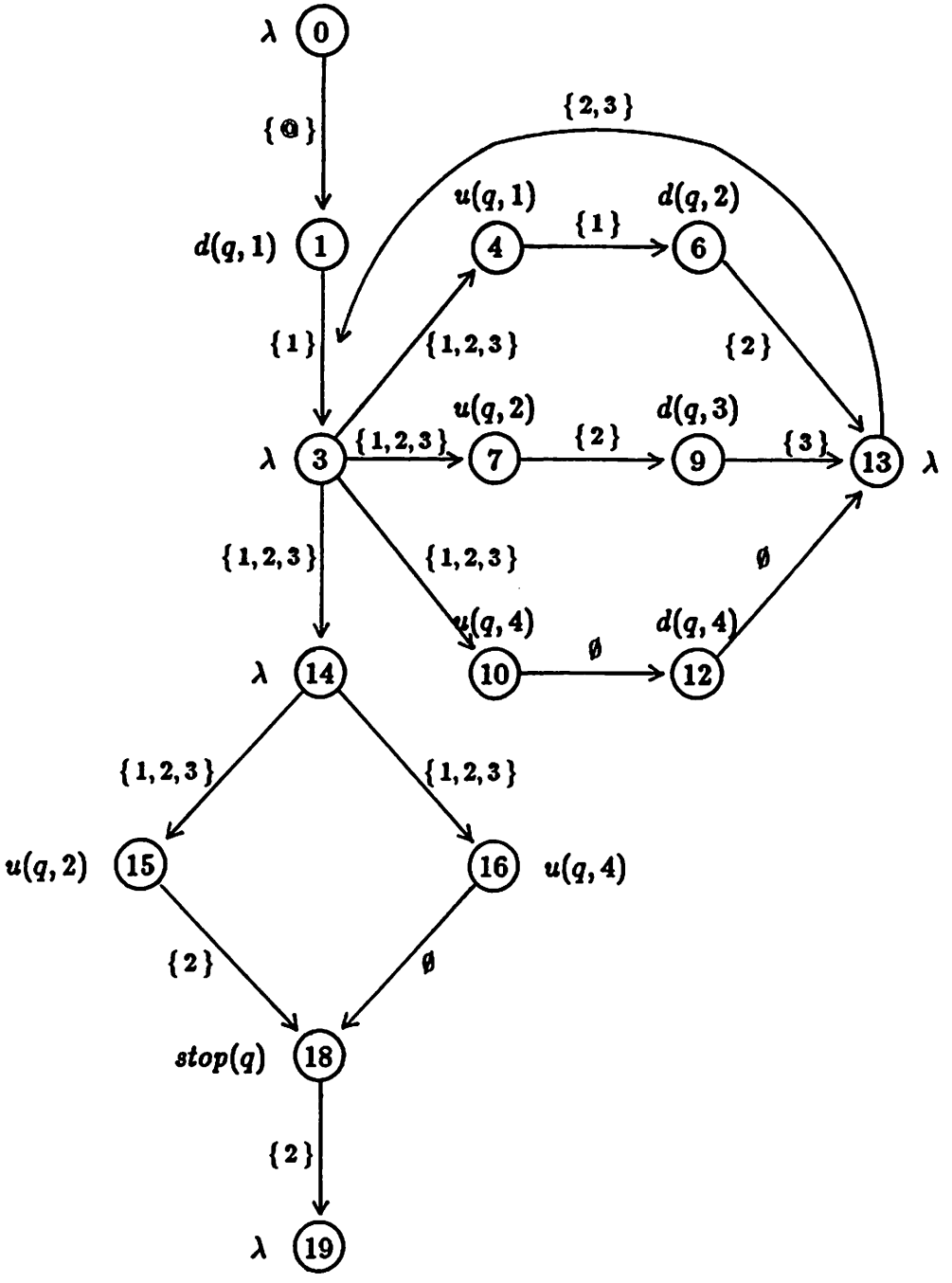


Figure 8

Edge labeling obtained using the flow graph of figure 6 and the first message flow analysis algorithm

edge	moves													
		a	c	b	a	c	c	b	a	c	c	c	b	c
(0,1)	@													
(1,3)		1												
(3,4)			1				2				3			
(4,6)				1										
(6,13)					2									
(3,7)			1				2				3			
(7,9)								2						
(9,13)									3					
(3,10)			1				2				3			
(10,12)														
(12,13)														
(13,3)						2				3				
(3,14)			1				2				3			
(14,15)												1,2,3		
(14,16)												1,2,3		
(15,18)													2	
(16,18)														
(18,19)														2

Figure 9

Message types added to the edges of the flow graph of figure 5 by a particular sequence of moves for the first message flow analysis algorithm

Proof. Observe that $\iota w \in \mathcal{L}(\epsilon)|_{\widehat{G}}$ implies that the string $v = \rho_{A_q}(w)$ belongs to the language of π_q and that the string $d(q, @)\, v$ satisfies the constraint $\kappa_2(q)$. We show that if a string $v \in \mathcal{L}(\pi_q)$ is represented by the complete path $(n_0, \ldots, n_k)$ in G and if the string $d(q, @)\, v$ satisfies the constraint $\kappa_2(q)$, then the edges (n_i, n_{i+1}), for $0 \leq i \leq k-1$ are all labeled with non-empty subsets of M. Hence, the path does not pass through the hypothesized node n, and so n is unattainable, as desired.

Assume, therefore, that $(n_0, \ldots, n_k)$ is a complete path in G, that $v = l(n_0) \cdots l(n_k)$, and that $d(q, @)\, v$ satisfies the constraint $\kappa_2(q)$. Let $0 \leq i \leq k-1$. If none of the nodes n_j, for $0 \leq j \leq i$, represent a modification of the buffer of process q, we show that the label for the edge (n_i, n_{i+1}) contains the message type $@$. Otherwise, there is a largest index $j \leq i$ such that n_j represents a modification of the buffer of process q. In this case, we show that the label for the edge (n_i, n_{i+1}) contains the message type m, where n_j is a $d(q, m)$-node.

This is easily proven by induction on the index i. The basis for the induction is trivial since the message type $@$ is added to the labels of all the edges leaving the root of G in step (1) of the first message flow analysis algorithm. If the node n_i is not a $d(q, m)$-node or a $u(q, m)$-node, for any $m \in M$, the conclusion then follows from the inductive hypothesis (the label for the edge (n_{i-1}, n_i) contains the required message type) and the move (c) of figure 7. If n_i is a $d(q, m)$-node, for some $m \in M$, the conclusion then follows from the inductive hypothesis (the label for the edge (n_{i-1}, n_i) is non-empty) and move (a) of figure 7. Otherwise, n_i is a $u(q, m)$-node, for some $m \in M$. If no previous node represents a modification of the buffer of process q, then since $d(q, @)\, v$ satisfies $\kappa_2(q)$, we have $m = @$. In this case, furthermore, the inductive hypothesis implies that the label for the edge (n_{i-1}, n_i) contains the message type $@$, and so move (b) of figure 7 applies to give the desired result. On the other hand, if some previous node represents the modification of the buffer of process q, then, since $d(q, @)\, v$ satisfies $\kappa_2(q)$, the

node with the largest index $j < i$ that represents such an event is a $d(q,m)$-node. By the inductive hypothesis, therefore, the label for the edge (n_{i-1}, n_i) contains the message type m, and so move (b) of figure 7 applies to give the desired result in this case also. ∎

The nodes 10, 12, and 16, in the flow graph of figure 8 are, therefore, unattainable. The results of the previous section indicate that the subexpressions of the process expressions in (2.3) corresponding to these nodes can be replaced with the empty regular expression. Thus, the process expression

$$\pi_q = d(q,1)\left(u(q,1)\,d(q,2) \vee u(q,2)\,d(q,3)\right)^* u(q,2)\,stop(q),$$

can be used in place of the process expression of (2.3) in an appropriate constrained expression representation of a system.

The first message flow analysis algorithm can be used to simplify the reduced process expressions for the fork processes f_i, for $0 \leq i \leq 4$, shown in figure VI.4. When this algorithm is applied to a flow graph for one of these processes, the nodes corresponding to the components of the last two terms in the iterated subexpressions (given by γ) of the process expression are seen to be unattainable, as are the nodes corresponding to the last two components of the second term of the process expression. These components, therefore, can be replaced with the empty regular expression, producing the process expression shown in figure 10.

The second message flow analysis algorithm is similar to the first one. Using the second message flow analysis algorithm, however, edge labeling functions for the flow graphs of a number of process expressions are constructed simultaneously, so that the message flow between processes can also be taken into account. The rationale behind the algorithm is as follows. If a process $q \in Q$ contains a link $l \in L(q)$, and if no $s(l,m)$-nodes in a flow graph for the process expression π_q are attainable, then the process q does not send a message of type m to the link l in any legal behavior of Σ. If a process $q' \in Q$ contains a port $p' \in P(q')$ connected

$$\pi_{f_i} = d(f_i, ok)\ \Big(u(f_i, ok)\, s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\, d(f_i, ok)$$
$$\vee\, u(f_i, ok)\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\, d(f_i, ok)\Big)^*$$
$$u(f_i, ok)\, s(f_i.u, ok)\, w(f_i.d)$$

Figure 10

Simplified process expression for a fork process obtained by applying the first message flow analysis algorithm to the process expression in figure VI.4

to l, therefore, it does not receive a message of type m from l through p' in any legal behavior. Every $r(l, p', m)$-node in a flow graph for the process expression $\pi_{q'}$ is thus unattainable. (This is formally assured by the constraint $\kappa_5(l, m)$.)

The algorithm is determined, in part, by the communications that are to be taken into account when constructing the labeling function. Before applying the algorithm, a designer identifies an appropriate set of reception events R to focus on (determined by the application). If $r(l, p, m) \in R$, for some $l \in L$, $p \in P$, and $m \neq \Omega$, then edge labeling functions for flow graphs for the process expressions of the processes that contain the link l and the port p are constructed simultaneously. The set R, therefore, determines the flow graphs whose edges are labeled by the second message flow analysis algorithm.

The most powerful version of the algorithm is achieved by taking R equal to all possible receptions events (i.e., $R = \{ r(l, p, m) \}_{l \in L, p \in T(l), m \neq \Omega}$). This, however, requires that labeling functions for flow graphs for every process containing a link, or containing a port that is connected to a link (usually, every process in the system) be constructed simultaneously. If the system is large, this may not be feasible. A designer might therefore decide to focus, for example, on the communications that

can be received through a particular set of ports $P' \subseteq P$. In this case, the set R would be defined as $R = \{ r(l,p,m) \}_{p \in P', l \in F(p), m \neq \Omega}$, and the algorithm would be applied to flow graphs for the process expressions of every process containing a port from P' and every process containing a link that is connected to a port from P'.

The second message flow analysis algorithm is defined in figure 11. Notice that the legal moves (a) and (b) are the same for both the first and the second message flow analysis algorithms. In the second message flow analysis algorithm, however, a move can only be made at an $r(l,p,m)$-node, for $r(l,p,m) \in R$, if the initial expression contains an $s(l,m)$ symbol or if an $s(l,m)$-node in some flow graph might be attainable. An $r(l,p,m)$-node represents a reception of a message of type m, but not the modification of the process' buffer resulting from the reception event, which is represented by the adjacent $d(q,m)$-node. In move (c), therefore, the elements of L, rather than the message type m, are added to the edge leaving n.

If all the edges leaving a node are labeled with $\emptyset$ by the second message flow analysis algorithm, the node is unattainable. This is assured by the constraints $\kappa_2(q)$ and $\kappa_5(l,m)$, for $q \in Q$, $l \in L$ and $m \neq \Omega$, in the constrained expression representation of an SDYMOL system. For completeness, we state this result as a proposition.

(3.2) **Proposition.** *Let* $\mathbf{D} = (\mathbf{F}, \epsilon)$ *be a reduced constrained expression representation for an* SDYMOL *system* Σ, *where* $\mathbf{F} = (A, E, \widehat{S}, \widehat{C})$, $\epsilon = \iota\left(\underset{q \in Q}{\triangle} \pi_q \right)$, ι *is the initial expression derived from the design of* Σ, *and the process expressions* $\pi_q \in \mathcal{RE}(A_q)$, *for* $q \in Q$, *do not involve the shuffle operator, and let* $G = (N, E)$, *with node labeling function* l, *be a flow graph for the process expression* π_q, *for some* $q \in Q$. *If* $\kappa_2(q), \kappa_5(l,m) \in \widehat{C}$, *for* $q \in Q$, $l \in L$, *and* $m \neq \Omega$, *and if, after applying the second message flow analysis algorithm to* G, *every edge leaving a node* $n \in N$ *is labeled with* $\emptyset$, *then* n *is unattainable.*

Input:

A set R of reception events.

A flow graph G_q for the process expression π_q, for every $q \in Q$ such that, for some $r(l,p,m) \in R$, $l \in L(q)$ or $p \in P(q)$.

The initial expression ι derived from the design of Σ.

Algorithm:

(i) Label all edges leaving the root nodes of the graphs G_q with $\{\Omega\}$ and all other edges with $\emptyset$.

(ii) Perform legal moves until no further moves can be made or no move that can be made changes the label of any of the edges in any of the graphs G_q.

Legal moves: Given a node n in a flow graph G_q, an edge entering n labeled L and an edge leaving n labeled L', the following are legal moves.

(a) If $L \neq \emptyset$ and n is a $d(q,m)$-node, then add m to L'.

(b) If $L \neq \emptyset$, n is a $u(q,m)$-node, and $m \in L$, then add m to L'.

(c) If $L \neq \emptyset$, n is an $r(l,p,m)$-node, for some $r(l,p,m) \in R$, then if ι contains an $s(l,m)$ symbol or if some edge leaving an $s(l,m)$-node in $G_{q'}$, where $l \in L(q')$, is labeled with a non-empty set, add the elements of L to L'.

(d) If $L \neq \emptyset$, n is not a $d(q,m)$-node or a $u(q,m)$-node, for any $m \in M$, n is not an $r(l,p,m)$-node, for any $r(l,p,m) \in R$, and n is not an $\emptyset$-node, then add the elements of L to L'.

Figure 11

The second message flow analysis algorithm

The proof of this proposition is similar to the proof of the corresponding proposition for the first message flow analysis algorithm (proposition (3.1)).

The second message flow analysis algorithm subsumes the first, since it detects all the unattainable nodes that are detected by applying the first algorithm to the appropriate process expressions. When applying the algorithms manually, we usually simplify large process expressions using the first message flow analysis algorithm, and then proceed to simplify different groups of process expressions using the second message flow analysis algorithm. This reduces the complexity of individual applications of the algorithms. If the algorithms were automated, this conservative approach to simplification might be unnecessary.

While the message flow analysis algorithms detect many unattainable nodes in flow graphs for the process expressions in a constrained expression representation of an SDYMOL system, they do not necessarily detect all unattainable nodes. Consider, for example, a system consisting of two processes q_1 and q_2, whose process expressions are given by

$$\pi_{q_1} = d(q_1, m)\, s(l_1, m)\, stop(q_1), \text{ and}$$

$$\pi_{q_2} = r(l_1, p_2, m)\, d(q_2, m)\, r(l_1, p_2, m)\, d(q_2, m)\, stop(q_2) \vee r(l_1, p_2, m)\, d(q_2, m)\, stop(q_2),$$

where l_1 is a link belonging to the process q_1, p_2 is a port belonging to the process q_2 and there is a channel connecting the link l_1 to the port p_2. Although the nodes corresponding to the last three components in the first term of π_{q_2} are unattainable, neither of the message flow analysis algorithms reveal this fact. This is because the message flow analysis algorithms do not take into account the number and order of event occurences in legal behavioral traces of a system. For this, more powerful techniques are required. We thus consider algebraic techniques for analyzing a reduced constrained expression representation of an SDYMOL system. A routine application of these techniques eliminates the first alternative of π_{q_2}.

CHAPTER VIII

ANALYSIS OF SDYMOL CONSTRAINED EXPRESSION REPRESENTATIONS

In this chapter we show how a reduced constrained expression representation can be analyzed to establish specific behavioral properties of the system it models. Our approach to analyzing a constrained expression representation of a system is based on determining whether specific patterns of event symbols can occur in strings from the interpreted language of the constrained expression. The patterns in question might correspond to some desirable property of the system or might represent certain pathological behaviors, such as deadlocks, which we would like to assure do not occur. To support this approach to reasoning about constrained expressions, we use the modularization techniques presented in chapter V and the algebraic techniques described in chapter I. The propositions developed in §V.3 are used to focus the analysis on specific constraints, essentially reducing the analysis of the original constrained expression to the analysis of a simpler one. The results of §V.4 permit us to use the analysis of a system in the subsequent analysis of a more detailed design for the same (or a similar) system. As explained in §I.1, the algebraic techniques begin with the assumption that a certain pattern of events occurs in a string representing a behavior of the system. They then use the constrained expression to iteratively generate inequalities involving the numbers of occurrences of particular symbols appearing in various segments of the hypothesized string. If the assumption leads, at any stage of the iterative process, to an inconsistent system of inequalities, a contradiction has been reached. The assumption is thus incorrect and the given pattern does not occur in a behavior. Otherwise, inequalities are generated until enough information is obtained to construct an example behavior containing the given pattern.

We demonstrate our approach to analysis by applying it to the constrained

expression representation of an SDYMOL system. Our starting point is the reduced constrained expression representation of the SDYMOL solution to the dining philosophers problem obtained in chapters VI and VII. By analyzing this constrained expression representation, we are able to characterize the behaviors in which the philosopher processes starve and demonstrate that the fork processes synchronize the philosopher processes as intended. We then modify the solution to the dining philosophers problem to eliminate the potential deadlock revealed by this analysis. Finally, we show how analysis of the modified system demonstrates that a philosopher process in this new system never starves and that the required synchronization of the philosopher processes is also achieved. Both of these examples illustrate general techniques for analyzing a reduced constrained expression representation of a system. The second example also shows how the analysis of a system might be modularized in a top-down development process.

§1. Analysis of the dining philosophers problem

We begin by analyzing a reduced constrained expression representation of the SDYMOL solution to the dining philosophers problem presented in figures III.1 and III.3. In particular, we examine the behaviors in which a philosopher process waits indefinitely for a communication through one of its inbound ports. If all the philosopher processes in the SDYMOL system execute a **receive** *right-up* command before any has executed the corresponding **receive** *left-up* command, the system deadlocks with all the philosopher processes waiting to receive a message through their *left-up* ports. (This models the unhappy situation described in [HOAR78] in which each philosopher picks up the fork on his/her right before any has picked up the fork on his/her left, so that they all sit forever, each holding onto one fork.) By analyzing the constrained expression representation of the system, we show that this is, in fact, the only way in which a philosopher process can starve. We also

show that the fork processes synchronize the philosopher processes as intended, which assures that the philosophers, as modeled in this system, use the forks in a consistent fashion. These are the types of properties that designers of concurrent systems must typically establish to determine if the systems they design behave as intended.

Preliminary notation and comments

For the analysis in this chapter, we require a notation to represent the strings obtained by concatenating a fixed number of strings from the language of a regular expression.

(1.1) *Notation.* For $\alpha \in \mathcal{RE}(A)$ and $n \geq 0$, we define α^n as follows:

$$\alpha^n = \begin{cases} \lambda, & \text{if } n = 0; \\ \alpha\alpha^{n-1}, & \text{if } n > 0. \end{cases}$$

We also require a notation to represent strings obtained by concatenating, in any arbitrary order, specific numbers of strings from the languages of each of the alternatives of a disjunction.

(1.2) *Notation.* If $\alpha, \beta \in \mathcal{RE}(A)$ and $m, n \geq 0$, we define $(\alpha \vee \beta)^{n \oplus m}$ as follows:

$$(\alpha \vee \beta)^{n \oplus m} = \begin{cases} \alpha^n, & \text{if } m = 0; \\ \beta^m, & \text{if } n = 0; \\ \alpha(\alpha \vee \beta)^{(n-1) \oplus m} \vee \beta(\alpha \vee \beta)^{n \oplus (m-1)}, & \text{otherwise.} \end{cases}$$

Thus, for example, $(\alpha \vee \beta)^{2 \oplus 1} = \alpha\alpha\beta \vee \alpha\beta\alpha \vee \beta\alpha\alpha$ represents the subset of strings from the language of $(\alpha \vee \beta)^*$ formed by selecting two strings from the language of the first alternative of $(\alpha \vee \beta)$ and one string from the language of the second alternative of $(\alpha \vee \beta)$, and then concatenating the selected strings in any arbitrary order. For different applications this notation could easily be generalized to apply to regular expressions that are expressed as the disjunction of any finite number of alternatives.

Finally, we require a notation to refer to the number of times an event symbol appears in a given event string.

(1.3) *Notation.* If $u \in A^*$ and $a \in A$, then $|a|_u$ denotes the number of occurrences of the event symbol a in u.

In applications where the event string u is fixed, we usually omit the subscript, writing $|a|$ instead of $|a|_u$.

We use Σ in this section to denote the SDYMOL system described in figures III.1 and III.3. The notation established in (III.2.1) is used when referring to the components of Σ. The analysis performed in this chapter is based on the reduced constrained expression representation for Σ that is obtained using the constraints shown in figure VI.2, and the system expression produced from the initial expression shown in figure VI.4, the process expressions for the philosopher processes also shown in this figure, and the simplified process expressions for the fork processes shown in figure VII.10. (Because the process expressions for the philosopher processes were simplified before reduction in chapter VI, the methods of chapter VII lead to no further simplifications of these process expressions.) Instead of analyzing this reduced constrained expression representation for Σ directly, however, we project the system expression on a subset of the augmented alphabet and focus on the constraints required to establish the desired properties, as described below.

The question regarding the starvation of philosopher processes is interpreted as asking if there are any constrained prefixes containing a $w(p)$ symbol, for $p \in P(p_i)$, $0 \leq i \leq 4$. The question regarding the intended synchronization of philosopher processes is interpreted as asking if there are any constrained prefixes containing an $r(f_i.u, p_i.lu, ok)$ symbol followed by an $r(f_i.u, p_{i+1}.ru, ok)$ symbol with no intervening $r(p_i.ld, f_i.d, ok)$ symbols, or an $r(f_{i-1}.u, p_i.ru, ok)$ symbol followed by an $r(f_{i-1}.u, p_{i-1}.lu, ok)$ symbol with no intervening $r(p_i.rd, f_{i-1}.d, ok)$ symbols. The constraints that (directly) restrict the order and number of these

symbols in constrained prefixes of the constrained expression representation for Σ are the constraints that relate to interprocess communication. It is natural, therefore, to focus the analysis on the constraints $\kappa_5(l, ok)$ and $\kappa_6(l, p, ok)$, for $l \in L$ and $p \in T(l)$.

Projecting the process expressions and initial expression described above on the alphabets of these constraints, we obtain the "process expressions" shown in figure 1 (the projection of the initial expression is the empty string, and so is omitted). The interleave of these process expressions produces the "system expression", also shown in this figure, which is used for the analysis that follows. For easy reference, the constraints $\kappa_5(l, ok)$ and $\kappa_6(l, p, ok)$, for $l \in L$ and $p \in T(l)$, are shown in figure 2. In this chapter, therefore, we let A denote the augmented alphabet derived from the design of Σ, ϵ denote the system expression obtained by interleaving the process expressions π_q, for $q \in Q$, given in figure 1, $\widehat{C}$ denote the collection of constraints given in figure 2, and $\widehat{S}$ denote the corresponding collection of constraint alphabets. We then analyze strings from $\mathcal{L}(\epsilon)|_{\widehat{C}}$. Proposition (V.3.2) implies that if a particular pattern of $w(p)$ and $r(l, p, ok)$ symbols, for $l \in L$ and $p \in T(l)$, does not appear in strings from $\mathcal{L}(\epsilon)|_{\widehat{C}}$, then this same pattern of symbols does not appear in constrained prefixes of the reduced constrained expression representation of Σ, justifying this approach to analyzing the number and order of symbols that appear in legal behavioral traces of Σ.

Our approach to the analysis of strings from $\mathcal{L}(\epsilon)|_{\widehat{C}}$ can be described as follows. We choose an abitrary string $u \in \mathcal{L}(\epsilon)|_{\widehat{C}}$ and consider the strings $\rho_{A_q}(u)$ obtained by projecting this string on the (derived) alphabets A_q of the processes $q \in Q$. Because these alphabets partition the augmented alphabet, we know that u is obtained by interleaving the strings $\rho_{A_q}(u)$, for $q \in Q$. We also know that $\rho_{A_q}(u)$, for each $q \in Q$, belongs to the language of the process expression π_q shown in figure 1. Based on the form of the process expression π_q, therefore, we obtain a representation for the string $\rho_{A_q}(u)$ in which the number of repetitions

$$\pi_{p_i} =$$
$$\Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^*$$
$$\vee\ \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^*$$
$$w(p_i.ru)$$
$$\vee \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^*$$
$$r(f_{i-1}.u, p_i.ru, ok)\, w(p_i.lu)$$

$$\pi_{f_i} =$$
$$\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^*$$
$$s(f_i.u, ok)\, w(f_i.d)$$

$$\epsilon = \Big(\Big(\underset{0\le i\le 4}{\triangle} \pi_{p_i}\Big) \triangle \Big(\underset{0\le i\le 4}{\triangle} \pi_{f_i}\Big)\Big)$$

Figure 1

Projected process expressions and system expression used for the analysis of the behaviors of Σ

$$\kappa_5(f_i.u, ok) =$$

$$s(f_i.u, ok)^* \triangle \Big[s(f_i.u, ok)\Big(r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)\Big]^\dagger$$

$$\kappa_5(p_i.ld, ok) = s(p_i.ld, ok)^* \triangle \Big(s(p_i.ld, ok)\, r(p_i.ld, f_i.d, ok)\Big)^\dagger$$

$$\kappa_5(p_i.rd, ok) = s(p_i.rd, ok)^* \triangle \Big(s(p_i.rd, ok)\, r(p_i.rd, f_{i-1}.d, ok)\Big)^\dagger$$

$$\kappa_6(f_i.u, p_i.lu, ok) =$$

$$\Big[s(f_i.u, ok)\Big(r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)\Big]^\dagger w(p_i.lu)$$

$$\Big(s(f_i.u, ok)\, r(f_i.u, p_{i+1}.ru, ok)\Big)^\dagger$$

$$\vee \Big(s(f_i.u, ok) \vee r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)^*$$

Figure 2

Constraints used for the analysis of the behaviors of Σ

$$\kappa_6(f_i.u, p_{i+1}.ru, ok) =$$
$$\Big[s(f_i.u, ok)\Big(r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)\Big]^{\dagger} w(p_{i+1}.ru)$$
$$\Big(s(f_i.u, ok)\, r(f_i.u, p_i.lu, ok)\Big)^{\dagger}$$
$$\vee \Big(s(f_i.u, ok) \vee r(f_i.u, p_{i+1}.ru, ok) \vee r(f_i.u, p_i.lu, ok)\Big)^{*}$$

$$\kappa_6(p_i.rd, f_{i-1}.d, ok) = \Big(s(p_i.rd, ok)\, r(p_i.rd, f_{i-1}.d, ok)\Big)^{\dagger} w(f_{i-1}.d)$$
$$\vee \Big(s(p_i.rd, ok) \vee r(p_i.rd, f_{i-1}.d, ok)\Big)^{*}$$

$$\kappa_6(p_i.ld, f_i.d, ok) = \Big(s(p_i.ld, ok)\, r(p_i.ld, f_i.d, ok)\Big)^{\dagger} w(f_i.d)$$
$$\vee \Big(s(p_i.ld, ok) \vee r(p_i.ld, f_i.d, ok)\Big)^{*}$$

Figure 2 (continued)

of each iterated subexpression of π_q is represented with a symbolic value using the notation of (1.1) and (1.2). By focusing on specific constraints in the set $\widehat{C}$, we then obtain inequalities relating these symbolic values, which we use to reason about the form of the strings $\rho_{A_q}(u)$, and the symbols that appear in these strings.

We also characterize the manner in which the symbols in the strings $\rho_{A_q}(u)$, for $q \in Q$, are interleaved to produce the string u. For this, we use the constraints of $\widehat{C}$ to reason about the form of the prefixes of $\rho_{A_q}(u)$ that are interleaved to produce a prefix of the string u. Given a prefix w of u, we project w on the alphabets of the processes to obtain the strings $\rho_{A_q}(w)$, for $q \in Q$, that are interleaved to form w. As when reasoning about the strings $\rho_{A_q}(u)$, we then examine the form of the process expressions π_q and specific constraints of $\widehat{C}$ to develop inequalities relating the numbers of different symbols that appear in the strings $\rho_{A_q}(w)$.

Starvation of philosopher processes

To characterize the behaviors of Σ in which a philosopher process starves, we chose $u \in \mathcal{L}(\epsilon)|_{\widehat{\mathcal{O}}}$ and examine the form of the strings $\rho_{A_{p_i}}(u)$, for $0 \leq i \leq 4$, showing that these strings do not contain any $w(p_i.ru)$ symbols, for $0 \leq i \leq 4$, and that if one of them contains a $w(p_i.lu)$ symbol, for $0 \leq i \leq 4$, then they all do.

Using the notation established in the previous section, we first observe that the string $\rho_{A_{p_i}}(u)$, for $0 \leq i \leq 4$, belongs to the language of

$$
\begin{aligned}
&\Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^{n_i} \\
&\quad \vee \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^{n_i} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad w(f_i.ru) \\
&\quad \vee \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^{n_i} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad r(f_{i-1}.u, p_i.ru, ok)\, w(p_i.lu),
\end{aligned}
\tag{1.4}
$$

for some $n_i \geq 0$, and that the string $\rho_{A_{f_i}}(u)$, for $0 \leq i \leq 4$, belongs to the language of

$$(1.5) \qquad \Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(l_i \oplus m_i)} \; s(f_i.u, ok)\, w(f_i.d),$$

for some $l_i, m_i \geq 0$. (Clearly, n_i is the number of times the philosopher process p_i completes a full loop of the **while internal test** statement, l_i is the number of times the fork process f_i receives a message from the link $p_i.ld$ while executing a full loop of the **do forever** statement, and m_i is the number of times the fork process f_i receives a message from the link $p_{i+1}.rd$ while executing a full loop of the **do forever** statement.)

Because the fork process f_i cycles "forever", first receiving a message through its *down* port and then sending a message to its *up* link, it must eventually starve waiting for a communication through the port $f_i.d$. (Recall that we only consider behaviors in the system runs to completion.) All messages deposited in the links connected to this port are thus eventually received. This is formally assured by the constraints $\kappa_6(p_i.ld, f_i.d, ok)$ and $\kappa_6(p_{i+1}.rd, f_i.d, ok)$ and permits us to establish an important relationship between the numbers n_i, l_i, and m_i of (1.4) and (1.5).

(1.6) **Proposition.** *If n_i, l_i, and m_i, for $0 \leq i \leq 4$, are defined as in (1.4) and (1.5), then $n_i = l_i$ and $n_{i+1} = m_i$, for $0 \leq i \leq 4$.*

Proof. For a given i, $0 \leq i \leq 4$, the constraint $\kappa_6(p_i.ld, f_i.d, ok)$ and the fact that the process expression π_{f_i} contains a $w(f_i.d)$ symbol assure that the messages sent to the link $p_i.ld$ are all received through the port $f_i.d$. We therefore focus on this constraint to show that $n_i = l_i$. For this, we define an alphabet $S = S_6(p_i.ld, f_i.d, ok)$ and consider $\rho_S(u)$.

To make use of the representations for the strings $\rho_{A_{P_i}}(u)$ and $\rho_{A_{f_i}}(u)$ shown in (1.4) and (1.5), we partition S into the sets $S_q = S \cap A_q$, for $q \in Q$. As

the only non-empty sets in this partition are the sets S_{p_i} and S_{f_i}, we observe that $\rho_S(u)$ is obtained by interleaving the strings $\rho_{S_{p_i}}(u)$ and $\rho_{S_{f_i}}(u)$. Examining the expressions for $\rho_{A_{p_i}}(u)$ and $\rho_{A_{f_i}}(u)$ shown in (1.4) and (1.5), we then reason that $\rho_{S_{p_i}}(u)$ belongs to the language of $s(p_i.ld, ok)^{n_i}$, and that $\rho_{S_{f_i}}(u)$ belongs to the language of $r(p_i.ld, f_i.d, ok)^{l_i}\, w(f_i.d)$.

The string $\rho_S(u)$, therefore, contains a $w(f_i.d)$ symbol. As it also satisfies the constraint $\kappa_6(p_i.ld, f_i.d, ok)$, we conclude that it belongs to the language of the first alternative of $\kappa_6(p_i.ld, f_i.d, ok)$ (see figure 2). As every string in this language contains the same number of $s(p_i.ld, ok)$ symbols as $r(p_i.ld, f_i.d, ok)$ symbols, we have $n_i = |s(p_i.ld, ok)| = |r(p_i.ld, f_i.d, ok)| = l_i$, where $|a|$ denotes $|a|_u$ here, and in the remainder of the proof.

Similarly, to show $n_{i+1} = m_i$, we focus on the constraint $\kappa_6(p_{i+1}.rd, f_i.d, ok)$. We therefore define $S = S_6(p_{i+1}.rd, f_i.d, ok)$, partition S into the sets $S_q = S \cap A_q$, for $q \in Q$, and observe that $\rho_S(u)$ is obtained by interleaving the string $\rho_{S_{p_{i+1}}}(u)$, which belongs to the language of $s(p_{i+1}.rd, ok)^{n_{i+1}}$, and the string $\rho_{S_{f_i}}(u)$, which belongs to the language of $r(p_{i+1}.rd, f_i.d, ok)^{n_i}\, w(f_i.d)$. As $\rho_S(u)$ also satisfies the constraint $\kappa_6(p_{i+1}.rd, f_i.d, ok)$, we have $n_{i+1} = |s(p_{i+1}.rd, ok)| = |r(p_{i+1}.rd, f_i.d, ok)| = m_i$. ∎

This proposition implies that the fork process f_i completes as many iterations of its loop as the philosopher processes p_i and p_{i+1}, together, complete of their loops, i.e., that

$$l_i + m_i = n_i + n_{i+1} \qquad \text{for } 0 \leq i \leq 4. \tag{1.7}$$

Using this relationship, we now show that a philosopher process never waits indefinitely for a communication through its *right-up* port.

(1.8) **Proposition.** *If* $u \in \mathcal{L}(\epsilon)|_{\widehat{G}}$ *then* u *does not contain any* $w(p_i.ru)$ *symbols, for* $0 \leq i \leq 4$.

Proof. Defining n_i, l_i, and m_i, for $0 \le i \le 4$, as in (1.4) and (1.5), we show that the result (1.7) is contradicted if u is assumed to contain a $w(p_i.ru)$ symbol.

For this, we focus on the constraint $\kappa_6(f_{i-1}.u, p_i.ru, ok)$, which relates to the starvation event $w(p_i.ru)$. We therefore define $S = S_6(f_{i-1}.u, p_i.ru, ok))$, partition S into the sets $S_q = S \cap A_q$, for $q \in Q$, and observe that $\rho_{S_{p_{i-1}}}(u)$ then lies in the language of $r(f_{i-1}.u, p_{i-1}.lu, ok)^{n_{i-1}}$, that $\rho_{S_{f_{i-1}}}(u)$ lies in the language of $s(f_{i-1}.u, ok)^{(l_{i-1}+m_{i-1})}s(f_{i-1}.u, ok)$, and that $\rho_{S_{p_i}}(u)$, if u is assumed to contain a $w(p_i.ru)$ symbol, lies in the language of $r(f_{i-1}.u, p_i.ru, ok)^{n_i} w(p_i.ru)$. Since $\rho_S(u)$ is obtained by interleaving these strings and it also satisfies the constraint $\kappa_6(f_{i-1}.u, p_i.ru, ok)$, which assures that it contains the same number of $s(f_{i-1}.u, ok)$ symbols as $r(f_{i-1}.u, p, ok)$ symbols, for $p \in \{p_i.ru, p_{i-1}.lu\}$, we conclude that $n_i + n_{i-1} = l_{i-1} + m_{i-1} + 1$, contradicting (1.7). ∎

This proposition shows that the second term of π_{p_i} in figure 1 can be eliminated. In the remainder of this section, therefore, we take

$$\begin{aligned}\pi_{p_i} = {} & \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^* \\ \vee {} & \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^* \\ & r(f_{i-1}.u, p_i.ru, ok)\, w(p_i.lu),\end{aligned} \tag{1.9}$$

when forming the system expression $\epsilon = \Big(\underset{0\le i\le 4}{\triangle} \pi_{p_i}\Big) \triangle \Big(\underset{0\le i\le 4}{\triangle} \pi_{f_i}\Big)$. The string $\rho_{A_{p_i}}(u)$, obtained by projecting a string $u \in \mathcal{L}(\epsilon)|_{\widehat{O}}$ on the alphabet of p_i, is thus assumed to belong to the language of

$$\begin{aligned}& \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^{n_i} \\ \vee {} & \Big(r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\Big)^{n_i} \\ & r(f_{i-1}.u, p_i.ru, ok)\, w(p_i.lu),\end{aligned} \tag{1.10}$$

and the string $\rho_{A_{f_i}}(u)$, to belong to the language of (1.5), for some $l_i, m_i, n_i \ge 0$ satisfying (1.7).

We now show that if the philosopher process p_i starves waiting for a communication through its *left-up* port, then so does the philosopher process p_{i+1}. (Recall that addition of subscripts in this example is performed modulo 5.) This implies that all the philosopher processes starve waiting for a communication through their *left-up* ports or all the philosopher processes terminate normally.

(1.11) **Proposition.** *If* $u \in \mathcal{L}(\epsilon)|_{\widehat{\sigma}}$ *contains a* $w(p_i.lu)$ *symbol, where* $0 \leq i \leq 4$, *then it also contains a* $w(p_{i+1}.lu)$ *symbol.*

Proof. Defining l_i and m_i, for $0 \leq i \leq 4$, as in (1.5), and n_i, for $0 \leq i \leq 4$, as in (1.10), we show the result (1.7) is contradicted if u is assumed to contain a $w(p_i.lu)$ symbol and not to contain any $w(p_{i+1}.lu)$ symbols, for some $0 \leq i \leq 4$.

We obtain the contradiction using the constraint $\kappa_6(f_i.u, p_i.lu, ok)$, which relates to the starvation event $w(p_i.lu)$. Defining S to be the alphabet of this constraint, and partitioning S into the sets $S_q = S \cap A_q$, for $q \in Q$, we note that $\rho_{S_{f_i}}(u)$ belongs to the language of $s(f_i.u, ok)^{(l_i+m_i)}s(f_i.u, ok)$, and, if u contains a $w(p_i.lu)$ symbol but does not contain a $w(p_{i+1}.lu)$ symbol, that $\rho_{S_{p_i}}(u)$ belongs to the language of $r(f_i.u, p_i.lu, ok)^{n_i}w(p_i.lu)$, while $\rho_{S_{p_{i+1}}}(u)$ belongs to the language of $r(f_i.u, p_{i+1}.ru, ok)^{n_{i+1}}$. As $\rho_S(u)$ is obtained by interleaving these strings and it also satisfies $\kappa_6(f_i.u, p_i.lu, ok)$, which assures that it contains the same number of $s(f_i.u, ok)$ symbols as $r(f_i.u, p, ok)$ symbols, for $p \in \{p_i.lu, p_{i+1}.ru\}$, we conclude that $n_i + n_{i+1} = l_i + m_i + 1$, contradicting (1.7). ∎

The above arguments, which are typical of the algebraic arguments required for reasoning about constrained expressions, are really much easier and more natural than they first appear. We have chosen to express them in the style of mathematical proofs. While this style of writing facilitates the statement of the arguments, it does not reveal the natural reasoning process that led to their formulation. For example, to eliminate the alternative containing the $w(p_i.ru)$ symbol in the process expression π_{p_i} of figure 1, it is natural to assume that there is a constrained prefix

containing a $w(p_i.ru)$ symbol, and then try to use the constraints to arrive at a contradiction. Initially, we focus on the contraints that are relevant to showing that the philosopher process p_i cannot starve waiting to receive a message through its *right-up* port. Clearly, the constraint $\kappa_6(f_{i-1}.u, p_i.ru, ok)$, which is the only constraint containing a $w(p_i.ru)$ symbol, is relevant. It tells us that

$$|s(f_{i-1}.u, ok)| = |r(f_{i-1}.u, p_i.ru, ok)| + |r(f_{i-1}.u, p_{i-1}.lu, ok)|,$$

where $|a|$, for an event symbol a, denotes the number of occurrences of a in a constrained prefix that is assumed to contain a $w(p_i.ru)$ symbol. By itself, this constraint does not lead to a contradiction. We therefore look for other constraints that might also be relevant.

Intuitively, we reason that if the philosopher process p_i starves waiting for a message through its *right-up* port, then, after some point in the behavior, the fork process f_{i-1} must never send another *ok* message to its *up* link. But this can only happen if the fork process f_{i-1} starves waiting for a message through its *down* port. The constraints $\kappa_6(p_{i-1}.ld, f_{i-1}.d, ok)$ and $\kappa_6(p_i.rd, f_{i-1}.d, ok)$, which pertain to the starvation event $w(f_{i-1}.d)$, are thus seen to be (indirectly) relevant. Since every constrained prefix contains a $w(f_{i-1}.d)$ symbol, these two constraints tells us that

$$|s(p_i.rd, ok)| = |r(p_i.rd, f_{i-1}.d, ok)|$$

and

$$|s(p_{i-1}.ld, ok) = |r(p_{i-1}.ld, f_{i-1}.d, ok)|.$$

We now look at the implications of the above facts and the form of the process expression (see figure 1) to try to obtain a contradiction. The form of the process expression reveals that

$$|s(f_{i-1}.u, ok)| = |r(p_{i-1}.ld, f_{i-1}.d, ok)| + |r(p_i.rd, f_{i-1}.d, ok)| + 1,$$

$$|s(p_{i-1}.ld, ok)| = |r(f_{i-1}.u, p_{i-1}.lu, ok)|,$$

and

$$|s(p_i.rd, ok)| = |r(f_{i-1}.u, p_i.ru, ok)|.$$

Together, these six equalties are clearly contradictory.

Similarly, to show that every constrained prefix containing a $w(p_i.lu)$ symbol also contains a $w(p_{i+1}.lu)$ symbol, it is natural to assume that there is some constrained prefix containing a $w(p_i.lu)$ symbol, but not containing a $w(p_{i+1}.lu)$ symbol, and try to use the constraints to obtain a contradiction. We first consider the constraint $\kappa_6(f_i.u, p_i.lu, ok)$, which relates to the starvation event $w(p_i.lu)$. When this constraint does not result in a contradiction, we consider the constraints $\kappa_6(p_i.ld, f_i.d, ok)$ and $\kappa_6(p_{i+1}.rd, f_i.d, ok)$, which relate to the starvation event $w(f_i.d)$, and hence (indirectly) to the starvation event $w(p_i.lu)$ as well. As in the above argument, equalities implied by these constraints and the form of the system expression are easily seen to be contradictory. When writing down these arguments, a common argument (proposition (1.6)), which is used in both of them, is presented separately so that it does not have to be repeated.

It is easy to produce a legitimate event sequence in which all the philosopher processes starve waiting for a communication through their *left-up* ports. For example, the constrained prefix

$$\iota\Big(\prod_{0\le i\le 4} d(f_i, ok)\, u(f_i, ok)\, s(f_i.u, ok)\, w(f_i.d)$$
$$think_{i+1}\, r(f_i.u, p_{i+1}.ru, ok)\, d(p_{i+1}, ok)\, w(p_{i+1}.lu)\Big),$$

where ι denotes the initial expression (shown in figure VI.4), is such a sequence. When projected on the terminal alphabet E (defined in §III.2), it produces the

string,

$$r(f_0.u, p_1.ru, ok)\, w(p_1.lu)\, r(f_1.u, p_2.ru, ok)\, w(p_2.lu)\, r(f_2.u, p_3.ru, ok)$$
$$w(p_3.lu)\, r(f_3.u, p_4.ru, ok)\, w(p_4.lu)\, r(f_4.u, p_0.ru, ok)\, w(p_0.lu).$$

A philosopher process in this solution to the dining philosophers problem can starve, therefore, but only if all the philosopher processes starve waiting for a communication through their *left-up* ports.

Synchronization of philosopher processes

In the arguments presented above, we examine the form of the process expressions and restrictions imposed by certain constraints on the symbols that appear in strings from $\mathcal{L}(\epsilon)|_{\widehat{C}}$ to reason about such things as the number of times an iterated subexpression of a particular process expression is repeated or as to which alternative of a disjunction in a process expression is selected when forming a string $u \in \mathcal{L}(\epsilon)|_{\widehat{C}}$. This gives us information about the form of the strings $\rho_{A_q}(u)$, for $q \in Q$, but not about how these strings are interleaved to produce the string u. To demonstrate that the fork process f_i, for a given $0 \leq i \leq 4$, synchronizes the philosopher processes p_i and p_{i+1} as intended, it is necessary to show that between every pair of symbols from $\{ r(f_i.u, p_i.lu, ok), r(f_i.u, p_{i+1}.ru, ok) \}$ in u, there is an intervening $r(p_i.ld, f_i.d, ok)$ or $r(p_{i+1}.rd, f_i.d, ok)$ symbol. We are thus required to reason not only about the form of the strings $\rho_{A_q}(u)$, for $q \in Q$, but also about the relative order of the symbols from the strings $\rho_{A_{f_i}}(u)$, $\rho_{A_{p_i}}(u)$, and $\rho_{A_{p_{i+1}}}(u)$ in the string u.

We therefore examine prefixes of a string $u \in \mathcal{L}(\epsilon)|_{\widehat{C}}$ and the restrictions imposed by the constraints of $\widehat{C}$ on the numbers of different event symbols that appear in these prefixes. The constraint $\kappa_5(l, ok)$, for a link $l \in L$, for example, assures that there are at least as many $s(l, ok)$ symbols as $r(l, p, ok)$ symbols, for $p \in T(l)$, in every prefix of u. Thus, if the string $\rho_{A_q}(u)$, for some $q \in Q$, contains

an $r(l,p,ok)$ symbol, for some $p \in P(q)$ and $l \in F(p)$, the constraint $\kappa_5(l,ok)$ assures that the $s(l,ok)$ symbol representing the transmission of the ok message that is received from l through p appears in the string $\rho_{A_{q'}}(u)$, where q' is the process containing the link l, and that this $s(l,ok)$ symbol precedes the $r(l,p,ok)$ symbol in u. Of course, any symbols in $\rho_{A_q}(u)$ that precede this $s(l,ok)$ symbol also precede the given $r(l,p,ok)$ symbol in u.

If u contains a $w(p)$ symbol, the constraint $\kappa_6(l,p,ok)$, for $l \in F(p)$, provides additional information about the manner in which the symbols from the strings $\rho_{A_q}(u)$, for $q \in Q$, are interleaved to produce u. In this case, the strings $\rho_{A_q}(u)$, for $q \in Q$, are interleaved so that the same number of $s(l,ok)$ symbols as $r(l,p',ok)$ symbols, for $p' \in T(l)$, precede the $w(p)$ symbol in u, and so that the same number of $s(l,ok)$ symbols as $r(l,p',ok)$ symbols, for $p' \in T(l)$ and $p' \neq p$, follow the $w(p)$ symbol in u.

By communicating with the philosopher processes p_i and p_{i+1}, the fork process f_i, for a given $0 \leq i \leq 4$, is intended to assure that the philosopher processes are properly synchronized with respect to one another (i.e., that two philosophers never simultaneously hold the same fork). By looking at the restrictions that the constraints $\kappa_5(l,ok)$, for the links l through which these processes communicate (i.e., for $l \in \{f_i.u, p_i.ld, p_{i+1}.rd\}$), impose on prefixes of a string $u \in \mathcal{L}(\epsilon)|_{\widehat{G}}$, we obtain a useful characterization of the patterns of symbols relating to the flow of messages through these links that appear in the string u.

(1.12) **Proposition.** *Given $0 \leq i \leq 4$, let S be the union of the constraint alphabets for the constraints $\kappa_5(l,ok)$, for $l \in \{f_i.u, p_i.lu, p_{i+1}.rd\}$. Then, if $v\,r(f_i.u,p,ok)$ is a prefix of a string $u \in \mathcal{L}(\epsilon)|_{\widehat{G}}$, there exist integers $n_i, n_{i+1} \geq 0$ such that*

$$\rho_{S_{p_i}}(v) \in \mathcal{L}\left(\left(r(f_i.u,p_i.lu,ok)\,s(p_i.ld,ok)\right)^{n_i}\right),$$

$$\rho_{S_{p_{i+1}}}(v) \in \mathcal{L}\left(\left(r(f_i.u, p_{i+1}.ru, ok)\, s(p_{i+1}.rd, ok)\right)^{n_{i+1}}\right), \quad \text{and}$$

$$\rho_{S_{f_i}}(v) \in \mathcal{L}\Big(\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(n_i \oplus n_{i+1})} s(f_i.u, ok)\Big),$$

where $S_{p_i} = S \cap A_{p_i}$, $S_{p_{i+1}} = S \cap A_{p_{i+1}}$, and $S_{f_i} = S \cap A_{f_i}$.

Proof. We give the proof for $p = p_i.lu$. The proof for $p = p_{i+1}.ru$ is similar.

We first consider $\rho_{S_{p_i}}(v)$. Since $\rho_{S_{p_i}}\left(v\, r(f_i.u, p_i.lu, ok)\right)$ is a prefix of $\rho_S(\pi_{p_i})$, we see from the form of the process expression π_{p_i} (see (1.9)) that $\rho_{S_{p_i}}(v)$ lies in the language of $\left(r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\right)^{n_i}$, for some $n_i \geq 0$.

Next we consider $\rho_{S_{p_{i+1}}}(v)$ and $\rho_{S_{f_i}}(v)$. We see from the form of the process expressions that there are integers $n_{i+1}, l_i, m_i \geq 0$ such that either

(i) $\rho_{S_{p_{i+1}}}(v) \in \mathcal{L}\left(\left(r(f_i.u, p_{i+1}.ru, ok)\, s(p_{i+1}.rd, ok)\right)^{n_{i+1}}\right)$, or

(ii) $\rho_{S_{p_{i+1}}}(v) \in \mathcal{L}\left(\left(r(f_i.u, p_{i+1}.ru, ok)\, s(p_{i+1}.rd, ok)\right)^{n_{i+1}} r(f_i.u, p_{i+1}.ru, ok)\right)$,

and either

(iii) $\rho_{S_{f_i}}(v) \in \mathcal{L}\Big(\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(l_i \oplus m_i)}\Big)$, or

(iv) $\rho_{S_{f_i}}(v) \in \mathcal{L}\Big(\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(l_i \oplus m_i)} s(f_i.u, ok)\Big)$.

We rule out (ii) and (iii) above, by showing that (i) and (iv) is the only combination of possibilities that does not result in a contradiction. For this argument,

we use the following inequalities:

$$|s(f_i.u, ok)| \geq |r(f_i.u, p_i.lu, ok)| + |r(f_i.u, p_{i+1}.ru, ok)|,$$
$$|s(p_i.ld, ok)| \geq |r(p_i.ld, f_i.d, ok)|, \quad \text{and}$$
$$|s(p_{i+1}.rd, ok)| \geq |r(p_{i+1}.rd, f_i.d, ok)|,$$

where $|a|$ denotes $|a|_{\rho_S(v\, r(f_i.u, p_i.lu, ok))} = |a|_{v\, r(f_i.u, p_i.lu, ok)}$, for $a \in S$. These inequalities follow from the observations that $\rho_S\left(v\, r(f_i.u, p_i.lu, ok)\right)$ is a prefix of the string $\rho_S(u)$, and that $\rho_S(u)$ satisfies the constraints $\kappa_5(f_i.u, ok)$, $\kappa_5(p_i.ld, ok)$, and $\kappa_5(p_{i+1}.rd, ok)$.

To see how these inequalities produce the desired contradictions, first note that the second and third inequalities, together with the characterizations of the strings $\rho_{S_{p_i}}(v)$, $\rho_{S_{p_{i+1}}}(v)$, and $\rho_{S_{f_i}}(v)$ obtained above, imply that $n_i \geq l_i$ and $n_{i+1} \geq m_i$ (since $|s(p_i.ld, ok)| = n_i$, $|r(p_i.ld, f_i.d, ok)| = l_i$, $|s(p_{i+1}.rd, ok)| = n_{i+1}$, and $|r(p_{i+1}.rd, f_i.d, ok)| = m_i$ no matter what combination of the conditions (i)–(iv) is satisfied). Now, if (i) and (iii) are both satisfied, we have $l_i + m_i \geq (n_i + 1) + n_{i+1}$ by the first inequality. But this clearly contradicts the fact that $n_i \geq l_i$ and $n_{i+1} \geq m_i$. Similarly, if (ii) and (iii) are satisfied, the first inequality implies $l_i + m_i \geq (n_i + 1) + (n_{i+1} + 1)$, contradicting the fact that $n_i \geq l_i$ and $n_{i+1} \geq m_i$, and if (ii) and (iv) are satisfied, the first inequality implies $l_i + m_i + 1 \geq (n_i + 1) + (n_{i+1} + 1)$, which also contradicts this fact.

By the process of elimination, therefore, we conclude (i) and (iv) are satisfied. Hence, using the first inequality, we have $l_i + (m_i + 1) \geq (n_i + 1) + n_{i+1}$. Since we also have $n_i \geq l_i$ and $n_{i+1} \geq m_i$, we conclude that $n_i = l_i$ and $m_i = n_{i+1}$, as desired. ∎

Using this characterization of the behavior of the processes f_i, p_i, and p_{i+1}, we now show that the intended synchronization of the philosopher processes is realized in the system Σ.

(1.13) Proposition. *If* $u \in \mathcal{L}(\epsilon)|_{\widehat{\mathcal{O}}}$ *contains an* $r(f_i.u, p, ok)$ *symbol that is followed by an* $r(f_i.u, p', ok)$ *symbol, for ports* $p, p' \in \{p_i.lu, p_{i+1}.ru\}$ *and* $0 \leq i \leq 4$, *then, if* $p = p_i.lu$, *there is an intervening* $r(p_i.ld, f_i.d, ok)$ *symbol, and, if* $p = p_{i+1}.ru$, *there is an intervening* $r(p_{i+1}.rd, f_i.d, ok)$ *symbol.*

Proof. Without loss of generality we may assume that there are no intervening $r(f_i.u, p'', ok)$ symbols, for $p'' \in \{p_i.lu, p_{i+1}.ru\}$, between the given $r(f_i.u, p, ok)$ and $r(f_i.u, p', ok)$ symbols in u.

We therefore write $u = x\, r(f_i.u, p, ok)\, y\, r(f_i.u, p', ok)\, z$, where $x, y, z \in A^*$ and y does not contain any $r(f_i.u, p_i.lu, ok)$ or $r(f_i.u, p_{i+1}.ru, ok)$ symbols. Applying proposition (1.12), first with $v = x$, and then with $v = x\, r(f_i.u, p, ok)\, y$, we conclude that there exist integers $n_i, n_{i+1}, n_i', n_{i+1}' \geq 0$ such that

(i) $\rho_{S_{p_i}}(x) \in \mathcal{L}\left(\left(r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\right)^{n_i}\right)$,

(ii) $\rho_{S_{p_{i+1}}}(x) \in \mathcal{L}\left(\left(r(f_i.u, p_{i+1}.ru, ok)\, s(p_{i+1}.rd, ok)\right)^{n_{i+1}}\right)$,

(iii) $\rho_{S_{f_i}}(x) \in \mathcal{L}\Big(\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)$
$\vee\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(n_i \oplus n_{i+1})} s(f_i.u, ok)\Big)$,

(iv) $\rho_{S_{p_i}}\left(x\, r(f_i.u, p, ok)\, y\right) \in \mathcal{L}\left(\left(r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\right)^{n_i'}\right)$,

(v) $\rho_{S_{p_{i+1}}}\left(x\, r(f_i.u, p, ok)\, y\right) \in \mathcal{L}\left(\left(r(f_i.u, p_{i+1}.ru, ok)\, s(p_{i+1}.rd, ok)\right)^{n_{i+1}'}\right)$, and

(vi) $\rho_{S_{f_i}}\left(x\, r(f_i.u, p, ok)\, y\right) \in \mathcal{L}\Big(\Big(s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)$
$\vee\, s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok)\Big)^{(n_i' \oplus n_{i+1}')} s(f_i.u, ok)\Big)$.

Now, if $p = p_i.lu$, then since y does not contain any $r(f_i.u, p'', ok)$ symbols, for $p'' \in \{p_i.lu, p_{i+1}.ru\}$, (i) and (iv) imply that $n_i' = n_i + 1$, and (ii) and (v) imply that $n_{i+1}' = n_{i+1}$. Hence, we conclude, using (iii) and (vi), that y contains a $r(p_i.ld, f_i.d, ok)$ symbol. Similarly, if $p = p_i.ru$ then $n_i' = n_i$ and $n_{i+1}' = n_{i+1} + 1$, and so y contains a $r(p_i.rd, f_i.d, ok)$ symbol. ∎

§2. A modified solution and its constrained expression representation

The analysis of the constrained expression representation for the system Σ performed in the previous section shows that the philosophers, as modeled in Σ, use the forks in a consistent fashion, and that a philosopher who is ready to eat can be denied access to a fork indefinitely only if all the philosophers are holding their right forks, and so are waiting for their left forks to become available. One way to eliminate such undesirable behaviors is to limit the number of philosophers that can be holding their right forks at any particular point in time to four. As suggested in [HOAR78], this can be accomplished by never allowing more than four philosophers to be in the dining room at the same time.

We therefore modify the solution to the dining philosophers problem presented in figures III.1 and III.3, adding two new processes, a *front door* and *back door* process, and various ports, links, channels, and message types. SDYMOL programs for the processes in this modified system are shown in figure 3. Using these programs, the abbreviations of figure III.2, and the abbreviations of figure 4, the system, which we denote by Σ', is defined in figure 5. We use a primed notation when refering to the components of Σ'. Thus, the set Q' in figure 5 identifies the processes that make up Σ' and the set M' identifies the message types that are used for communication in Σ'. The set of (inbound) ports of Σ' is represented by P' and the set of links by L'. The SDYMOL code, the set of ports, and the set of links (outbound ports) for a process $q \in Q'$ are denoted by $\mathbf{code}'_q$, $P'(q)$, and $L'(q)$, respectively. The set of links connected to a port $p \in P'$ is written $F'(p)$, and the set of ports connected to a link $l \in L'$ is written $T'(l)$.

The front door and back door processes model attendants who maintain a count of the number of philosophers in the room. (This count is represented by the single integer message type residing in the *ct-out* links, which is initialized to 0 in statements BD1–2.) Upon becoming hungry, a philosopher knocks at the entrance

```
        front-door:
FD1       do forever
            begin
FD2         receive knock;
FD3         receive ct-in;
FD4         if buffer < 4 then
              begin
FD5           set buffer := buffer + 1;
FD6           send ct-out;
FD7           set buffer := go;
FD8           send signal;
              end;
            else
FD9           begin
FD10          send ct-out;
FD11          set buffer := stay;
FD12          send signal;
              end;
          end.
```

```
        back-door:
BD1       set buffer := 0;
BD2       send ct-out;
BD3       do forever
            begin
BD4         receive exit;
BD5         receive ct-in;
BD6         set buffer := buffer - 1;
BD7         send ct-out;
          end.
```

```
        phil:
P1        while internal test do
            begin
P2          think;
P3          set buffer := tap;
P4          send knock;
P5          receive signal;
P6          if buffer = go then
              begin
P7            receive right-up;
P8            receive left-up;
P9            eat;
P10           send left-down;
P11           send right-down;
P12           set buffer := tap;
P13           send exit;
              end;
          end.
```

```
        fork:
F1        set buffer := ok;
F2        do forever
            begin
F3          send up;
F4          receive down;
          end.
```

Figure 3

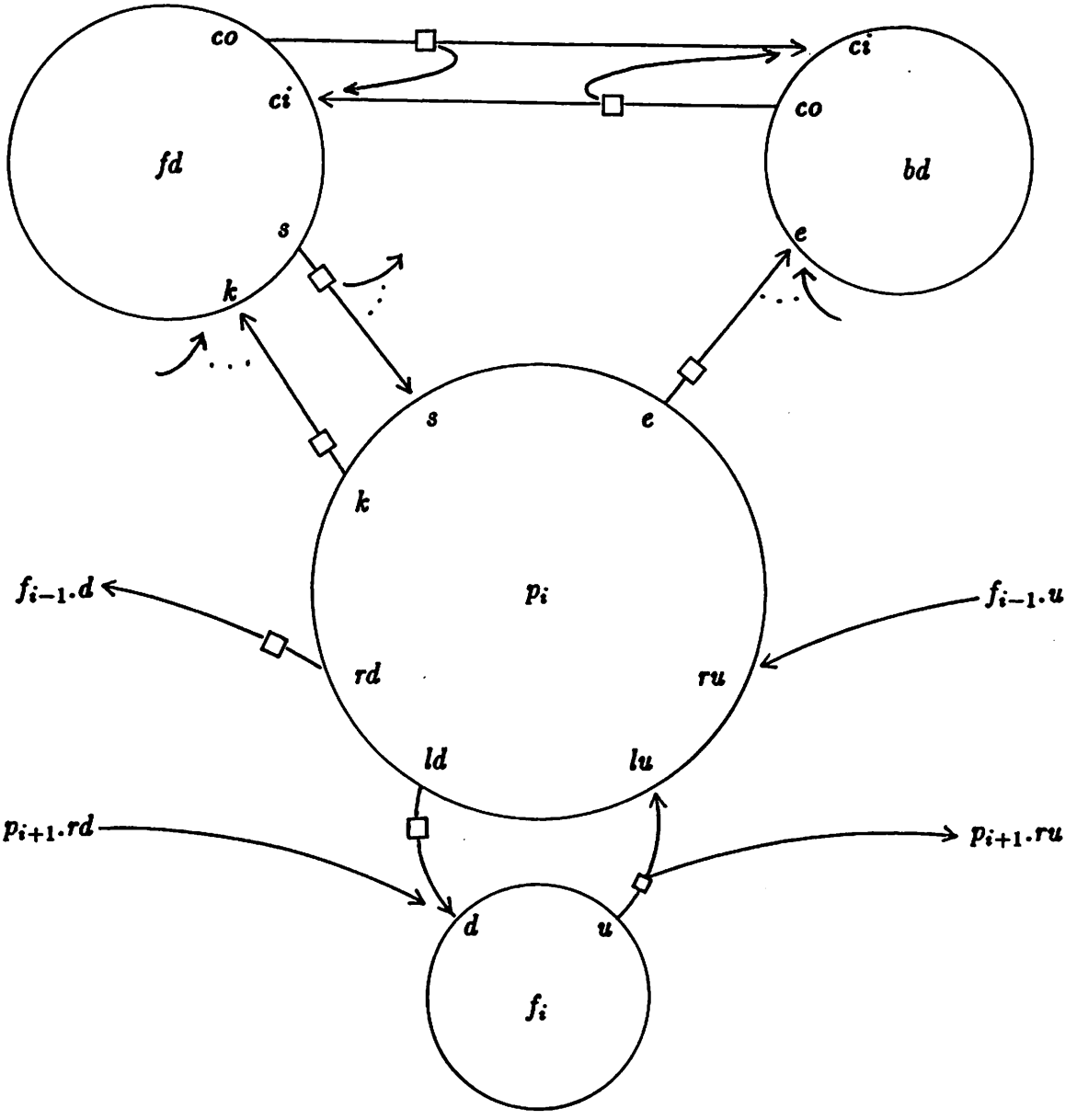
SDYMOL programs for the
modified solution to the dining philosophers problem

Process/port name	Abbreviation
back-door	*bd*
front-door	*fd*
signal	*s*
knock	*k*
exit	*e*
ct-in	*ci*
ct-out	*co*

Figure 4

Abbreviations for new process and port names

to the dining room (modeled by statements P3–4). A knock at the front door of the dining room causes the front door attendant to check the count of philosophers currently in the dining room (statements FD2–4). If the count is less than four, the attendant increments it and signals one of the hungry philosophers to enter (FD5–8). Otherwise, the attendant signals one of these philosophers, indicating that the dining room is full (FD11–12). Note that in this case the count of the philosophers in the room is not modified (FD10). A philosopher waits to enter the dining room until he/she receives a signal from the front door attendant (P5) indicating that he/she may enter (checked in P6). Once in the dining room, the activity of a philosopher is modeled exactly as in the original system (P7–P11, synchronized by F3–4), except that after eating and replacing the forks, a philosopher notifies the back door attendant (P12–13), who then proceeds to decrement the count of philosophers in the room (BD5–7). If instead of being admitted to the dining room, a philosopher receives a signal from the front door attendant indicating the dining room is full, he/she begins another cycle of thinking, knocking, and, perhaps, eating. (Nothing in this solution assures that the front door attendant is "fair". One philosopher can be repeatedly told to stay in order to admit other philosophers.)



$$Q' = \{ fd, bd, p_0, p_1, p_2, p_3, p_4, f_0, f_1, f_2, f_3, f_4 \}$$

$$M' = \{ tap, go, stay, ok, 0, 1, 2, 3, 4 \}$$

Figure 5

The modified solution to the dining philosophers problem

For this example, we have extended the SDYMOL design notation described in §III.1 to include certain elementary arithmetic operations. The extension of the design notation, of course, must be accompanied by a corresponding extension of the procedure for deriving SDYMOL constrained expression representations for systems expressed in the notation. When deriving a constrained expression representation for such systems, we require that the integer message types consist of a finite range of integers and that they be distinguished from the other message types in a system. For this discussion, we let I denote the finite range of integer message types of a system and N denote the set consisting of the remaining message types. The least element of I is written $i_\perp$, and largest element is written $i_\top$. For the system Σ', for example, we have $I = \{0,1,2,3,4\}$, $i_\perp = 0$, $i_\top = 4$, and $N = \{\, tap, go, stay, ok, @ \,\}$. Because the arithmetic operations are only defined for integer message types and because the results of these operations are required to lie within the range I of integer message types, we also require event symbols to represent events in which a process tries to use a message type in a fashion that is not appropriate for messages of that type. In this example we use the symbols $ill(q)$, for $q \in Q'$, for this purpose.

In keeping with the primed notation for the components of the modified system Σ', we use a primed notation when refering to the components of a constrained expression representation for Σ'. In this way we distinguish these components from the corresponding components of a constrained expression representation for the original system Σ. The augmented and terminal alphabets derived from Σ' are thus written A' and E', respectively. The system expression is written ϵ', and consists of an initial expression ι', followed by the interleave of process expressions π'_q, for $q \in Q'$. The set of constraints is written $\widehat{C}'$ and the set of constraint alphabets is written $\widehat{S}'$.

The augmented alphabet A' derived from Σ' consists of the illegal operation symbols $ill(q)$, for $q \in Q'$, and the symbols described in (ii) of definition (III.2.5),

where q, l, p, and m, of course, range over Q', L', P', and M', respectively. As usual, the terminal alphabet E' is determined by the questions to be addressed by analysis of the constrained expression representation of Σ'.

The translation rules shown in figure III.5 are used for deriving a process expression π'_q from the code for a process $q \in Q'$. Additional translation rules are required, however, for statements that increment or decrement a process' buffer and for conditional and iterative statements in which the value of a process' buffer is compared to an integer constant. The rules used for the above example are shown in figure 6. The interpretation of these rules is obvious. Notice that every $ill(q)$ symbol in these rules is immediately followed by an $ne(q)$ symbol. This is because an attempt to use the value of a process's buffer in a fashion that is not consistent with the type of the value terminates execution of the process. (A complete analysis of a system, of course, should establish that illegal operations are never attempted. We address this in the example below.)

We derive constraints and constraint alphabets from Σ' as described in (iii) and (iv) of definition (III.2.5), except that q, l, p, and m range over Q', L', P', and M', and that the third type of constraints and constraint alphabets is modified to reflect the fact that a process can terminate because of attempting to perform an illegal operation, as described above. When refering to the constraints and constraint alphabets derived from the design of Σ', we use a primed notation, distinguishing them from the corresponding constraints and constraint alphabets derived from the design of the original system Σ. The modified constraints,

$$\kappa'_3(q) = \Big(\bigvee_{p \in P'(q)} w(p) \Big) \vee stop(q) \vee ill(q),$$

and constraint alphabets, $S'_3(q) = \{ stop(q), ill(q), w(p) \}$, where p ranges over $P'(q)$, thus assure that the processes in the system all run to completion.

We reduce the constrained expression representation derived from the design of Σ' as described in chapter VI. The preliminary simplifications described in §VI.1

set buffer $:=$ **buffer** $+ 1 \quad \rightarrow$

$$\Big(\bigvee_{m \in I - \{ i_\top \}} u(q,m)\, d(q, m+1) \Big) \vee \Big(\bigvee_{m \in N \cup \{ i_\top \}} u(q,m)\, ill(q)\, ne(q) \Big)$$

set buffer $:=$ **buffer** $- 1 \quad \rightarrow$

$$\Big(\bigvee_{m \in I - \{ i_\bot \}} u(q,m)\, d(q, m-1) \Big) \vee \Big(\bigvee_{m \in N \cup \{ i_\bot \}} u(q,m)\, ill(q)\, ne(q) \Big)$$

if buffer $< j$ **then** $\langle s \rangle \quad \rightarrow$

$$\Big(\bigvee_{i_\bot \le m < j} u(q,m)\, \{ \langle s \rangle \} \Big) \vee \Big(\bigvee_{j \le m \le i_\top} u(q,m) \Big) \vee \Big(\bigvee_{m \in N} u(q,m)\, ill(q)\, ne(q) \Big)$$

if buffer $< j$ **then** $\langle ss \rangle$ **else** $\langle s \rangle \quad \rightarrow$

$$\Big(\bigvee_{i_\bot \le m < j} u(q,m)\, \{ \langle ss \rangle \} \Big) \vee \Big(\bigvee_{j \le m \le i_\top} u(q,m)\, \{ \langle s \rangle \} \Big) \vee \Big(\bigvee_{m \in N} u(q,m)\, ill(q)\, ne(q) \Big)$$

Figure 6

Additional translation rules required for

the modified solution to the dining philosophers problem

are performed first, after which the process expressions in the system expression of the resulting constrained expression representation for the system are simplified further, using the results of §VI.2, and then reduced, as described in §VI.3. When reducing the process expressions, the $ill(q)$ symbols are treated in the same manner as $w(p)$ symbols.

Figure 7 shows the simplified system expression for the constrained expression representation of the system Σ' that is described in theorem (VI.1.11) (using the extended set of translation rules to define the map t'). Because we are reducing the representation of this system manually, we simplify the process expressions in this system expression further, before reducing them. Using proposition (VI.2.5), for example, we eliminate all the alternatives, except the $u(fd, go)\, s(fd.s, go)$ alternative, in the subexpression of π'_{fd} immediately following the $d(fd, go)$ symbol. Once these alternatives have been eliminated from π'_{fd}, we use proposition (VI.2.6) to eliminate all alternatives in π'_{p_i} involving a $r(fd.s, p_i.s, m)$ symbol, for $m \notin \{ stay, go \}$. When putting a process expression in disjunctive form, we also truncate all subterms beginning with the first component that follows a basic component consisting of a non-event symbol (justified by proposition (VI.2.4)). After performing these and similar simplifications, the process expressions are reduced using the procedure described in §VI.3, and producing the process expressions shown in figure 8.

We then apply the message flow analysis algorithms to eliminate many impossible subterms of these process expressions. Applying the first message flow analysis algorithm to the process expression π'_{f_i} of figure 8, for example, we eliminate all subterms of π'_{f_i} containing a $u(f_i, @)$ symbol. Applying the second message flow analysis algorithm to the process expressions π'_{f_i} and π'_{p_i}, for $0 \leq i \leq 4$, using the set of reception events involving the links through which these processes communicate (i.e., defining $R = \{ r(l, p, m) \}$, where l assumes the values $f_i.u$, $p_i.ld$, and $p_i.rd$, for $0 \leq i \leq 4$, p ranges over $T(l)$, and $m \neq @$, in figure VII.6), we eliminate all the subterms of the process expressions π'_{p_i} and π'_{f_i} that contain an

$$\iota' = \Big(\prod_{0\le i\le 4} ch(p_i.ld, f_i.d)\, ch(p_i.rd, f_{i-1}.d)\, ch(f_i.u, p_i.lu)\, ch(f_i.u, p_{i+1}.ru)$$

$$ch(p_i.k, fd.k)\, ch(fd.s, p_i.s)\, ch(p_i.e, bd.e)\Big)$$

$$ch(fd.co, bd.ci)\, ch(bd.co, bd.ci)\, ch(fd.co, fd.ci)\, ch(bd.co, fd.ci)$$

$$\Big(\prod_{0\le i\le 4} d(p_i, @)\, d(f_i, @)\Big)\, d(bd, @)\, d(fd, @)$$

$$\pi'_{fd} = \Big[\Big(\bigvee_{\substack{m\neq @\\ 0\le i\le 4}} r(p_i.k, fd.k, m)\, d(fd, m) \vee w(fd.k)\, ne(fd)\Big)$$

$$\Big(\bigvee_{m\neq @} r(fd.co, fd.ci, m)\, d(fd, m) \vee \bigvee_{m\neq @} r(bd.co, fd.ci, m)\, d(fd, m) \vee w(fd.ci)\, ne(fd)\Big)$$

$$\Big(\bigvee_{m\in N} u(fd, m)\, \ddot{u}l(fd)\, ne(fd)$$

$$\vee \bigvee_{0\le m\le 3} u(fd, m)\Big(\bigvee_{0\le m\le 3} u(fd, m)\, d(fd, m+1) \vee \bigvee_{m\in N\cup\{4\}} u(fd, m)\, \ddot{u}l(fd)\, ne(fd)\Big)$$

$$\Big(\bigvee_{m\neq @} u(fd, m)\, s(fd.co, m) \vee u(fd, @)\Big)\, d(fd, go)$$

$$\Big(\bigvee_{m\neq @} u(fd, m)\, s(fd.s, m) \vee u(fd, @)\Big)$$

$$\vee\ u(fd, 4)\Big(\bigvee_{m\neq @} u(fd, m)\, s(fd.co, m) \vee u(fd, @)\Big)\, d(fd, stay)$$

$$\Big(\bigvee_{m\neq @} u(fd, m)\, s(fd.s, m) \vee u(fd, @)\Big)\Big)\Big]^*\, ne(fd)\, stop(fd)$$

Figure 7

Initial expression, process expressions and corresponding system expression obtained using the procedure described in theorem (VI.1.11)

$$\pi'_{bd} = d(bd,0)\left(\bigvee_{m\neq@} u(bd,m)\, s(bd.co,m) \vee u(bd,@)\right)$$
$$\left[\left(\bigvee_{\substack{m\neq@\\0\leq i\leq 4}} r(p_i.e,bd.e,m)\, d(bd,m) \vee w(bd.e)\, ne(bd)\right)\right.$$
$$\left(\bigvee_{m\neq@} r(bd.co,bd.ci,m)\, d(bd,m) \vee \bigvee_{m\neq@} r(fd.co,bd.ci,m)\, d(bd,m) \vee w(bd.ci)\, ne(bd)\right)$$
$$\left(\bigvee_{1\leq m\leq 4} u(bd,m)\, d(bd,m-1) \vee \bigvee_{m\in N\cup\{0\}} u(bd,m)\, ill(bd)\, ne(bd)\right)$$
$$\left.\left(\bigvee_{m\neq@} u(bd,m)\, s(bd.co,m) \vee u(bd,@)\right)\right]^{*} ne(bd)\, stop(bd)$$

$$\pi'_{p_i} = \left[think_i\, d(p_i,tap)\left(\bigvee_{m\neq@} u(p_i,m)\, s(p_i.k,m) \vee u(p_i,@)\right)\right.$$
$$\left(\bigvee_{m\neq@} r(fd.s,p_i.s,m)\, d(p_i,m) \vee w(p_i.s)\, ne(p_i)\right)$$
$$\left(u(p_i,go)\left(\bigvee_{m\neq@} r(f_{i-1}.u,p_i.ru,m)\, d(p_i,m) \vee w(p_i.ru)\, ne(p_i)\right)\right.$$
$$\left(\bigvee_{m\neq@} r(f_i.u,p_i.lu,m)\, d(p_i,m) \vee w(p_i.lu)\, ne(p_i)\right) eat_i$$
$$\left(\bigvee_{m\neq@} u(p_i,m)\, s(p_i.ld,m) \vee u(p_i,@)\right)$$
$$\left(\bigvee_{m\neq@} u(p_i,m)\, s(p_i.ld,m) \vee u(p_i,@)\right) d(p_i,tap)$$
$$\left.\left(\bigvee_{m\neq@} u(p_i,m)\, s(p_i.e,m) \vee u(p_i,@)\right)\right.$$
$$\left.\left. \vee \bigvee_{m\neq go} u(p_i,m)\right)\right]^{*} stop(p_i)$$

Figure 7 (continued)

$$\pi'_{f_i} = d(f_i, ok)$$
$$\left[\left(\bigvee_{m \neq @} u(f_i, m)\, s(f_i.u, m) \vee u(f_i, @) \right) \right.$$
$$\left. \left(\bigvee_{m \neq @} r(p_{i+1}.rd, f_i.d, m)\, d(f_i, m) \vee r(p_i.ld, f_i.d, m)\, d(f_i, m) \vee w(f_i.d)\, ne(f_i) \right) \right]^*$$
$$ne(f_i)\, stop(f_i)$$

$$\epsilon' = \iota' \left(\pi'_{fd} \triangle \pi'_{bd} \triangle \left(\bigtriangleup_{0 \leq i \leq 4} \pi'_{p_i} \right) \triangle \left(\bigtriangleup_{0 \leq i \leq 4} \pi'_{f_i} \right) \right)$$

Figure 7 (continued)

$$\pi'_{fd} = \alpha^* w(fd.k) \vee \bigvee_{0 \le j \le 4} \alpha^* r(p_j.k, fd.k, tap)\, d(fd, tap)\, w(fd.ci)$$

$$\vee \bigvee_{\substack{0 \le j \le 4 \\ n \in N - \{0\}}} \alpha^* r(p_j.k, fd.k, tap)\, d(fd, tap)\, r(fd.co, fd.ci, n)\, d(fd, n)\, u(fd, n)\, ill(fd)$$

$$\vee \bigvee_{\substack{0 \le j \le 4 \\ n \in N - \{0\}}} \alpha^* r(p_j.k, fd.k, tap)\, d(fd, tap)\, r(bd.co, fd.ci, n)\, d(fd, n)\, u(fd, n)\, ill(fd)\text{, where}$$

$$\alpha = \bigvee_{\substack{0 \le i \le 4 \\ 1 \le m \le s}} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(fd.co, fd.ci, m)\, d(fd, m)\, u(fd, m)\, u(fd, m)$$
$$d(fd, m+1)\, u(fd, m+1)\, s(fd.co, m+1)\, d(fd, go)\, u(fd, go)\, s(fd.s, go)$$

$$\vee \bigvee_{\substack{0 \le i \le 4 \\ 0 \le m \le s}} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(bd.co, fd.ci, m)\, d(fd, m)\, u(fd, m)\, u(fd, m)$$
$$d(fd, m+1)\, u(fd, m+1)\, s(fd.co, m+1)\, d(fd, go)\, u(fd, go)\, s(fd.s, go)$$

$$\vee \bigvee_{0 \le i \le 4} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(fd.co, fd.ci, 4)\, d(fd, 4)\, u(fd, 4)\, u(fd, 4)$$
$$s(fd.co, 4)\, u(fd, stay)\, s(fd.s, stay)$$

$$\vee \bigvee_{0 \le i \le 4} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(bd.co, fd.ci, 4)\, d(fd, 4)\, u(fd, 4)\, u(fd, 4)$$
$$s(fd.co, 4)\, u(fd, stay)\, s(fd.s, stay)$$

Figure 8

Reduced process expressions for the modified solution to the dining philosophers problem

$$\pi'_{bd} = d(bd,0)\,u(bd,0)\,s(bd.co,0)\,\beta^*\,w(bd.e)$$

$$\vee \bigvee_{0\le j\le 4} d(bd,0)\,u(bd,0)\,s(bd.co,0)\,\beta^*\,r(p_j.e,bd.e,tap)\,d(bd,tap)\,w(p.ci)$$

$$\vee \bigvee_{\substack{n\in N\cup\{0\}\\ n\neq 0,\,0\le j\le 4}} d(bd,0)\,u(bd,0)\,s(bd.co,0)\,\beta^*\,r(p_j.e,bd.e,tap)\,d(bd,tap)$$
$$r(fd.co,bd.ci,n)\,d(bd,n)\,u(bd,n)\,\overline{ill}(bd)$$

$$\vee \bigvee_{\substack{n\in N\cup\{0\}\\ n\neq 0,\,0\le j\le 4}} d(bd,0)\,u(bd,0)\,s(bd.co,0)\,\beta^*\,r(p_j.e,bd.e,tap)\,d(bd,tap)$$
$$r(bd.co,bd.ci,n)\,d(bd,n)\,u(bd,n)\,\overline{ill}(bd)\,,$$

where

$$\beta = \bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e,bd.e,tap)\,d(bd,tap)\,r(fd.co,bd.ci,m)\,d(bd,m)\,u(bd,m)\,d(bd,m-1)$$
$$u(bd,m-1)\,s(bd.co,m-1)$$

$$\bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e,bd.e,tap)\,d(bd,tap)\,r(bd.co,bd.ci,m)\,d(bd,m)\,u(bd,m)\,d(bd,m-1)$$
$$u(bd,m-1)\,s(bd.co,m-1)$$

Figure 8 (continued)

$$\pi'_{p_i} = \gamma^* \, stop(p_i) \vee \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, w(p_i.s)$$

$$\vee \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go) \, u(p_i, go) \, w(p_i.ru)$$

$$\vee \bigvee_{n \neq @} \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go)$$
$$u(p_i, go) \, r(f_{i-1}.u, p_i.ru, n) \, d(p_i, n) \, w(p_i.lu) \, ,$$

where

$$\gamma = \bigvee_{m,m' \neq @} think_i \; d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go) \, u(p_i, go)$$
$$r(f_{1-1}.u, p_i.ru, m) \, d(p_i, m) \, r(f_i.u, p_i.lu, m') \, d(p_i, m') \, eat_i \, u(p_i, m')$$
$$s(p_i.ld, m') \, u(p_i, m') \, s(p_i.rd, m') \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.e, tap)$$

$$\vee \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, stay) \, d(p_i, stay) \, u(p_i, stay)$$

$$\pi'_{f_i} = d(f_i, ok) \, \delta^* \, u(f_i, @) \, w(f_i.d)$$

$$\vee \bigvee_{n \neq @} d(f_i, ok) \, \delta^* \, u(f_i, n) \, s(f_i.u, n) \, w(f_i.d) \, , \text{ where}$$

$$\delta = \bigvee_{m,m' \neq @} u(f_i, m) \, s(f_i.u, m) \, r(p_{i+1}.rd, f_i.d, m') \, d(f_i, m')$$

$$\vee \bigvee_{m,m' \neq @} u(f_i, m) \, s(f_i.u, m) \, r(p_i.ld, f_i.d, m') \, d(f_i, m')$$

$$\vee \bigvee_{m \neq @} u(f_i, @) \, r(p_{i+1}.rd, f_i.d, m) \, d(f_i, m)$$

$$\vee \bigvee_{m \neq @} u(f_i, @) \, r(p_i.ld, f_i.d, m) \, d(f_i, m)$$

Figure 8 (continued)

event symbol representing the reception of a message from one of these links where the type of the message received is not ok. Similarly, a number of subterms of the process expressions π'_{bd} and π'_{fd} are eliminated by applying the second message flow analysis algorithm using the reception events involving the links through which these processes communicate (i.e., defining $R = \{ r(l,p,m) \}$, where l ranges over $\{ fd.co, bd.co \}$, p ranges over $\{ fd.ci, bd.ci \}$, and $m \neq @$). Specifically, this eliminates all subterms of π'_{fd} containing an $ill(fd)$ symbol, all the subterms of π'_{bd} containing an $ill(bd)$ symbol and not containing an $r(bd.co, bd.ci, 0)$ symbol, all the subterms of both π'_{fd} and π'_{bd} containing an $r(bd.co, p, 4)$ symbol, for $p \in \{ fd.ci, bd.ci \}$, and all the subterms of π'_{fd} containing an $r(fd.co, fd.ci, 0)$ symbol. This produces the process expressions shown in figure 9.

The system expression for a reduced constrained expression representation for Σ' is thus obtained using the process expressions shown in figure 9 and the initial expression shown in figure 7. The constraints are obtained as described in theorem (VI.1.11), except that the constraints which assure the processes run to completion are modified as described above. It is this constrained expression representation for Σ' that we analyze below.

§3. Analysis of the modified system

To identify questions that the analysis of a constrained expression representation for the system Σ' should address, we consider the behavior that we would like the system to exhibit. The philosopher processes are intended to loop, executing statements P2–13 (see figure 3) some finite number of times, and then terminate normally. When analyzing a constrained expression representation of Σ', therefore, we would like to verify that a philosopher process never waits indefinitely for a communication through one of its inbound ports. The fork processes are designed to synchronize the philosopher processes, assuring that, once in the dining room, the

$$\pi'_{fd} = \alpha^* w(fd.k) \vee \bigvee_{0\le j\le 4} \alpha^* r(p_j.k, fd.k, tap)\, d(fd, tap)\, w(fd.ci)\text{, where}$$

$$\alpha = \bigvee_{\substack{0\le i\le 4\\ 1\le m\le 8}} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(fd.co, fd.ci, m)\, d(fd, m)\, u(fd, m)\, u(fd, m)$$
$$d(fd, m+1)\, u(fd, m+1)\, s(fd.co, m+1)\, d(fd, go)\, u(fd, go)\, s(fd.s, go)$$
$$\vee \bigvee_{\substack{0\le i\le 4\\ 0\le m\le 8}} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(bd.co, fd.ci, m)\, d(fd, m)\, u(fd, m)\, u(fd, m)$$
$$d(fd, m+1)\, u(fd, m+1)\, s(fd.co, m+1)\, d(fd, go)\, u(fd, go)\, s(fd.s, go)$$
$$\vee \bigvee_{0\le i\le 4} r(p_i.k, fd.k, tap)\, d(fd, tap)\, r(fd.co, fd.ci, 4)\, d(fd, 4)\, u(fd, 4)\, u(fd, 4)\, s(fd.co, 4)$$
$$u(fd, stay)\, s(fd.s, stay)$$

$$\pi'_{bd} = d(bd, 0)\, u(bd, 0)\, s(bd.co, 0)\, \beta^*\, w(bd.e)$$
$$\vee \bigvee_{0\le j\le 4} d(bd, 0)\, u(bd, 0)\, s(bd.co, 0)\, \beta^*\, r(p_j.e, bd.e, tap)\, d(bd, tap)\, w(bd.ci)$$
$$\vee \bigvee_{0\le j\le 4} d(bd, 0)\, u(bd, 0)\, s(bd.co, 0)\, \beta^*\, r(p_j.e, bd.e, tap)\, d(bd, tap)$$
$$r(bd.co, bd.ci, 0)\, d(bd, 0)\, u(bd, 0)\, ill(bd)\text{,}$$

where

$$\beta = \bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, d(bd, tap)\, r(fd.co, bd.ci, m)\, d(bd, m)\, u(bd, m)\, d(bd, m-1)$$
$$u(bd, m-1)\, s(bd.co, m-1)$$
$$\bigvee_{\substack{1\le m\le 8\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, d(bd, tap)\, r(bd.co, bd.ci, m)\, d(bd, m)\, u(bd, m)\, d(bd, m-1)$$
$$u(bd, m-1)\, s(bd.co, m-1)$$

Figure 9

Process expressions obtained from the modified solution to the dining philosophers problem after message flow analysis

$$\pi'_{p_i} = \gamma^* \, stop(p_i) \vee \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, w(p_i.s)$$

$$\vee \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go) \, u(p_i, go) \, w(p_i.ru)$$

$$\vee \gamma^* \, think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go) \, u(p_i, go)$$
$$r(f_{i-1}.u, p_i.ru, ok) \, d(p_i, ok) \, w(p_i.lu) \, ,$$

where

$$\gamma = think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, go) \, d(p_i, go) \, u(p_i, go)$$
$$r(f_{i-1}.u, p_i.ru, ok) \, d(p_i, ok) \, r(f_i.u, p_i.lu, ok) \, d(p_i, ok) \, eat_i \, u(p_i, ok) \, s(p_i.ld, ok)$$
$$u(p_i, ok) \, s(p_i.rd, ok) \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.e, tap)$$

$$\vee think_i \, d(p_i, tap) \, u(p_i, tap) \, s(p_i.k, tap) \, r(fd.s, p_i.s, stay) \, d(p_i, stay) \, u(p_i, stay)$$

$$\pi'_{f_i} = d(f_i, ok) \, \delta^* \, u(f_i, ok) \, s(f_i.u, ok) \, w(f_i.d) \, , \text{ where}$$

$$\delta = u(f_i, ok) \, s(f_i.u, ok) \, r(p_{i+1}.rd, f_i.d, ok) \, d(f_i, ok)$$

$$\vee u(f_i, ok) \, s(f_i.u, ok) \, r(p_i.ld, f_i.d, ok) \, d(f_i, ok)$$

Figure 9 (continued)

philosophers, as modeled in Σ', use the forks in a consistent fashion. There is no difference, of course, between the desired synchronization of philosopher processes in the system Σ', and the desired synchronization of philosopher processes in the original system Σ. We would like, therefore, to establish the same characterization of communication between the fork and philosopher processes for behaviors of Σ' as we established for behaviors of Σ. Together, the front door and back door processes are supposed to maintain a count of the number of philosopher processes within their *critical sections*, which we define by statements P7–13 of their code, and assure that no more than four philosopher processes are ever executing their critical sections simultaneously. This count is represented in a legal behavior of the system by the message residing the *ct-out* links. A philosopher process should be permited to enter its critical section only if this count is less than four. We would like to demonstrate, therefore, that there is never more than a single message residing in one of the *ct-out* links, which are thus used by the front door and back door processes in a mutually exclusive fashion, and that there is always (eventually) a message to be received from these links, so that the message can be updated as required. Additionally, we would like to verify that this message is updated as intended, so that the value of the message residing in the *ct-out* links at any point in a legal behavior of Σ' provides an "accurate" count of the number of philosopher processes executing their critical sections.

These behavioral properties of Σ' all relate to the manner in which the processes in Σ' communicate with one another. It is natural, therefore, to focus the analysis on the constraints and symbols that pertain to interprocess communication in Σ'. As we did when analyzing a reduced constrained expression representation of the original system Σ, therefore, we project the system expression on the alphabet of these constraints, producing a simpler "system expression", and examine the strings from the language of this new system expression satisfying this restricted set of constraints.

Projecting the process expressions of figure 9 on the alphabets of the constraints $\kappa_5'(l,m)$ and $\kappa_6'(l,p,m)$, for $l \in L'$, $p \in T'(l)$, and $m \neq @$, we obtain the "process expressions" shown in figure 10. The projection of the initial expression derived from the design of Σ' on these alphabets is null. We therefore use the system expression ϵ', obtained by interleaving these process expressions, as the system expression for the following analysis. The constraint set $\widehat{C}'$ used in this analysis consists of the constraints $\kappa_5'(l,m)$ and $\kappa_6'(l,p,m)$, for $l \in L'$, $p \in T'(l)$, and $m \neq @$. We examine strings from $\mathcal{L}(\epsilon')|_{\widehat{C}'}$, showing that these strings possess the desired properties. Proposition (V.3.2) assures that the properties established below by the analysis of such strings apply to strings from the constrained language of the reduced constrained expression representation of Σ' obtained in the previous section.

We first analyze the flow of messages through the links *fd.co* and *bd.co*, showing that these links are used in a mutually exclusive fashion and that the message residing in these links is updated as desired. Although this is the hardest part of the analysis, it is a necessary prerequisite to subsequent analysis. It is then easy to show that a message is always (eventually) available to be received from one of the *ct-out* links and also from the *signal* link. Finally, we show that the results pertaining to the behavior of Σ obtained in section §1 apply to the behavior of Σ'. This demonstrates that the philosopher processes are properly synchronized and that a philosopher process never starves waiting for a message through its *right-up* port. Using the characterization of the behaviors in which a philosopher process in Σ, and hence in Σ', waits indefinitely for a communication through its *left-up* port and the characterization of the flow of messages through the *ct-out* links, we then show that a philosopher process in Σ' never starves waiting for a message through its *left-up* port.

$$\pi'_{fd} = \alpha^* w(fd.k) \vee \alpha^* r(p_j.k, fd.k, tap)\, w(fd.ci)\text{, where}$$

$$\alpha = \bigvee_{\substack{0\le i\le 4\\ 1\le m\le 8}} r(p_i.k, fd.k, tap)\, r(fd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee \bigvee_{\substack{0\le i\le 4\\ 0\le m\le 8}} r(p_i.k, fd.k, tap)\, r(bd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee \bigvee_{0\le i\le 4} r(p_i.k, fd.k, tap)\, r(fd.co, fd.ci, 4)\, s(fd.co, 4)\, s(fd.s, stay)$$

$$\pi'_{bd} = s(bd.co, 0)\, \beta^*\, w(bd.e)$$

$$\vee \bigvee_{0\le j\le 4} s(bd.co, 0)\, \beta^*\, r(p_j.e, bd.e, tap)\, w(bd.ci)$$

$$\vee \bigvee_{0\le j\le 4} s(bd.co, 0)\, \beta^*\, r(p_j.e, bd.e, tap)\, r(bd.co, bd.ci, 0)\, ill(bd)\text{, where}$$

$$\beta = \bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(fd.co, bd.ci, m)\, s(bd.co, m-1)$$

$$\vee \bigvee_{\substack{1\le m\le 8\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(bd.co, bd.ci, m)\, s(bd.co, m-1)$$

Figure 10

Projected process expressions and system expression for analysis of modified system

$$\pi'_{p_i} = \gamma^* \vee \gamma^* s(p_i.k, tap)\, w(p_i.s)$$

$$\vee \gamma^* s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, w(p_i.ru)$$

$$\vee \gamma^* s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, r(f_{i-1}.u, p_i.ru, ok)\, w(p_i.lu)\text{, where}$$

$$\gamma = s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, r(f_{i-1}.u, p_i.ru, ok)\, r(f_i.u, p_i.lu, ok)\, s(p_i.ld, ok)\, s(p_i.rd, ok)\, s(p_i.c, tap)$$

$$\vee s(p_i.k, tap)\, r(fd.s, p_i.s, stay)$$

$$\pi'_{f_i} = \Big(s(f_i.u, ok)\, r(p_{i+1}.rd, f_i.d, ok) \vee s(f_i.u, ok)\, r(p_i.ld, f_i.d, ok)\Big)^* s(f_i.u, ok)\, w(f_i.d)$$

$$\epsilon' = \pi'_{fd} \triangle \pi'_{bd} \triangle \left(\underset{0 \le i \le 4}{\triangle} \pi'_{p_i}\right) \triangle \left(\underset{0 \le i \le 4}{\triangle} \pi'_{f_i}\right)$$

Figure 10 (continued)

Message flow through the *ct-out* links

To see why we first focus the analysis on the flow of messages through the links *fd.co* and *bd.co*, observe that if we try to use proposition (V.4.1) to demonstrate that the results obtained in §1, which pertain to the behavior of Σ, also describe the behavior of Σ', we find that in order to satisfy the hypotheses of this proposition we must eliminate the alternatives in the process expressions for the philosopher processes p_i, for $0 \leq i \leq 4$, that contain a $w(p_i.s)$ symbol. When focusing on the flow of messages through the links *fd.s* and $p_i.k$, for $0 \leq i \leq 4$, however, it quickly becomes evident that if the front door process can wait indefinitely to receive a message through its *ct-in* port, then a philosopher process can wait indefinitely to receive a message through its *signal* port. Before eliminating alternatives in the process expression for the philosopher processes p_i, for $0 \leq i \leq 4$, containing a $w(p_i.s)$ symbol, therefore, we must eliminate alternatives in the process expression for the front door process containing a $w(fd.ci)$ symbol. We are thus led to consider the flow of messages through the *ct-out* links.

Clearly, the front door process can wait indefinitely to receive a message through its *ct-in* port if execution of the back door process is terminated because of an attempt to decrement a message of type 0. Thus, we first show that the back door process never receives a message of type 0 through its *ct-in* port. Informally, the idea is that there is never more than a single message residing in the *ct-out* links, which assures that the front door and back door processes use these links in a mutually exclusive fashion. The front door process increments the single message residing in these links each time a philosopher process enters its critical section and the back door process decrements it each time a philosopher process leaves its critical section. Since a philosopher process always enters its critical section before leaving it, this assures that the message the back door process decrements is always of type greater than 0. This argument relies on characterizations of the manner in which the front door and back door processes use and modify the message residing

in their *ct-out* links and of the relationship between the value of this message and the state of the system.

To obtain the required characterization of the flow of messages through the *ct-out* links, we focus on the constraints $\kappa_5'(l,m)$, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$. We therefore define an alphabet,

$$S = \{ s(l,m), r(l,p,m) \}_{\substack{l \in \{bd.co, fd.co\} \\ p \in \{fd.ci, bd.ci\}, 0 \leq m \leq 4}}, \tag{3.1}$$

and partition S into the sets $S_{fd} = S \cap A'_{fd}$ and $S_{bd} = S \cap A'_{bd}$ determined by the alphabets A'_q of the processes $q \in Q'$. The projected process expressions, whose interleave produces the projected system expression $\rho_S(\epsilon')$, are shown in figure 11.

$$\rho_S(\pi'_{fd}) = \alpha^* \vee \bigvee_{0 \leq j \leq 4} \alpha^* w(fd.ci), \text{ where}$$

$$\alpha = \bigvee_{1 \leq m \leq 3} r(fd.co, fd.ci, m)\, s(fd.co, m+1) \vee \bigvee_{1 \leq m \leq 3} r(bd.co, fd.ci, m)\, s(fd.co, m+1)$$

$$\vee\, r(fd.co, fd.ci, 4)\, s(fd.co, 4)$$

$$\rho_S(\pi'_{bd}) = s(bd.co, 0)\, \beta^* \vee s(bd.co, 0)\, \beta^*\, w(bd.ci)$$

$$\vee\, s(bd.co, 0)\, \beta^*\, r(bd.co, bd.ci, 0), \text{ where}$$

$$\beta = \bigvee_{1 \leq m \leq 4} r(fd.co, bd.ci, m)\, s(bd.co, m-1) \vee \bigvee_{1 \leq m \leq 3} r(bd.co, bd.ci, m)\, s(bd.co, m-1)$$

Figure 11

Projection of the process expressions on the alphabet defined in (3.1)

Our first result regarding the flow of messages through the *ct-out* links is stated as follows.

(3.2) Proposition. *Let v be a prefix of a string $u \in \mathcal{L}(\epsilon')|_{\widehat{C}'}$. If S is partitioned into the sets S_{fd} and S_{bd} as defined in (3.1), then*

$$\sum |r(l,p,m)| = \begin{cases} \sum |s(p,m)|, & \text{or} \\ \sum |s(p,m)| - 1, \end{cases}$$

where the sums range over $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$, and $0 \le m \le 4$, and $|a|$ denotes $|a|_{\rho_S(v)} = |a|_v$, for $a \in S$. Moreover, the first (top) equality holds if the last symbol in $\rho_{S_{fd}}(v)$ or in $\rho_{S_{bd}}(v)$ is not an $s(l,m)$ symbol, where $l \in \{fd.co, bd.co\}$ and $0 \le m \le 4$, and the second (bottom) equality holds otherwise (i.e., if the last symbol in both $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$ is an $s(l,m)$ symbol, where $l \in \{fd.co, bd.co\}$ and $0 \le m \le 4$.)

Proof. From the form of the projected process expressions (see figure 11) we see that there is some $k \ge 0$ such that $\rho_{S_{fd}}(v)$ lies in the language of either

$$\Big(\bigvee_{\substack{l \in \{fd.co, bd.co\} \\ 0 \le m, m' \le 4}} r(l, fd.ci, m)\, s(fd.co, m') \Big)^k$$

or

$$\Big(\bigvee_{\substack{l \in \{fd.co, bd.co\} \\ 0 \le m, m' \le 4}} r(l, fd.ci, m)\, s(fd.co, m') \Big)^k r(l', fd.ci, m''),$$

where $l' \in \{fd.co, bd.co\}$, and $0 \le m'' \le 4$, and that there is some $j \ge 0$ such that $\rho_{S_{bd}}(v)$ lies in the language of either

$$s(bd.co, 0) \Big(\bigvee_{\substack{l \in \{fd.co, bd.co\} \\ 0 \le m, m' \le 4}} r(l, bd.ci, m)\, s(bd.co, m') \Big)^j$$

or

$$s(bd.co, 0) \Big(\bigvee_{\substack{l \in \{fd.co, bd.co\} \\ 0 \le m, m' \le 4}} r(l, bd.ci, m)\, s(bd.co, m') \Big)^j r(l', bd.ci, m''),$$

where $l' \in \{fd.co, bd.co\}$ and $0 \le m'' \le 4$.

Now, $\rho_S(v)$ is obtained by interleaving the strings $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$. If the last symbol in both $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$ is an $s(l,m)$ symbol, where $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, therefore, we have $\sum|r(l,p,m)| = k+j$ and $\sum|s(l,m)| = k+j+1$, and so $\sum|r(l,p,m)| = \sum|s(l,m)| - 1$. Similarly, if the last symbol in one of $\rho_{S_{fd}}(v)$ or $\rho_{S_{bd}}(v)$ is an $s(l,m)$ symbol, where $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, and the last symbol in the other is an $r(l,p,m)$ symbol, where $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$ and $0 \leq m \leq 4$, then we have $\sum|r(l,p,m)| = k+j+1 = \sum|s(l,m)|$.

We complete the proof by showing that it is not possible that the last symbol in both $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$ is an $r(l,p,m)$ symbol. This is because $\rho_S(v)$ is a prefix of $\rho_S(u)$, which satisfies the constraints $\kappa_5'(l,m)$, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, which implies that $\sum|s(l,m)| \geq \sum|r(l,p,m)|$. But if the last symbol in both $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$ is an $r(l,p,m)$ symbol, we have $\sum|s(l,m)| = k+j+1 < k+j+2 = \sum|r(l,p,m)|$, which is a contradiction. ∎

This proposition implies that there is never more than a single message residing in one of the *ct-out* links, and so the symbols from the set S representing the transmission of messages and the symbols from the set S representing the reception of messages alternate within strings from $\mathcal{L}(\epsilon')|_{\widehat{C}}$. Inspection of the process expressions then shows that the symbols representing the tranmission of messages and the symbols representing the reception of messages alternate within each of the processes, so that one of the front door or back door processes cannot send a message to its *ct-out* link if the other process has just received a message through its *ct-in* port and not yet sent a message to its *ct-out* link. This establishes the mutually exclusive use of the message residing in the *ct-out* links, as intended.

Using this proposition, we now establish the desired characterization of the flow of messages through the links *fd.co* and *bd.co*.

(3.3) **Proposition.** *If S is the alphabet defined in (3.1) and $u \in \mathcal{L}(\epsilon')|_{\widehat{C}'}$, then*

(i) if u contains an $s(l_1,m_1)$ symbol, where $l_1 \in \{fd.co, bd.co\}$ and $0 \leq$

$m_1 \leq 4$, then the next symbol from S in u is a $r(l_1, p_2, m_1)$ symbol, where $p_2 \in \{fd.ci, bd.ci\}$,

(ii) if u contains an $r(l_1, fd.ci, m_1)$ symbol, where $l_1 \in \{fd.co, bd.co\}$ and $0 \leq m_1 \leq 3$, then the next symbol from S in u is a $s(fd.co, m_1 + 1)$ symbol,

(iii) if u contains an $r(l_1, fd.ci, 4)$ symbol, where $l_1 \in \{fd.co, bd.co\}$, then the next symbol from S in u is a $s(fd.co, 4)$ symbol, and

(iv) if u contains an $r(l_1, bd.ci, m_1)$ symbol, where $l_1 \in \{fd.co, bd.co\}$ and $1 \leq m_1 \leq 4$, then the next symbol from S in u is a $s(bd.co, m_1 - 1)$ symbol.

The restricted patterns of symbols from the set S described by this proposition are shown pictorially in figure 12. The ellipses in this figure represent arbitrary sequences of event symbols. The proposition justifies the assumption that the sequences represented by the middle ellipses in each pattern do not contain any symbols from the set S.

Proof of proposition (3.3). The statements (i)–(iv) are vacuously true if the hypothesized symbol is the last symbol from S in u.

To establish (i), assume that u contains a $s(l_1, m_1)$ symbol, where $l_1 \in \{fd.co, bd.co\}$ and $0 \leq m_1 \leq 4$, and that this $s(l_1, m_1)$ symbol is not the last symbol from S in u. Then since the symbols from S representing the transmission of messages and the symbols from S representing the reception of messages alternate within u, the next symbol from S in u is a $r(l_2, p_2, m_2)$ symbol, for some $l_2 \in \{fd.co, bd.co\}$, $p_2 \in \{fd.ci, bd.ci\}$ and $0 \leq m_2 \leq 4$.

Now, if there are no symbols from S in u preceeding the $s(l_1, m_1)$ symbol, then applying proposition (3.2) to the prefix of u that ends with the $r(l_2, p_2, m_2)$ symbol, we conclude that $l_1 = l_2$ and $m_1 = m_2$, as desired.

On the other hand, if there is a symbol from S in u preceeding the $s(l_1, m_1)$ symbol, then the last such symbol is an $r(l_0, p_0, m_0)$ symbol, where l_0 is one

$$\cdots s(l_1, m_1) \cdots r(l_1, p_2, m_1) \cdots \qquad \text{for } 0 \leq m_1 \leq 4$$

$$\cdots r(l_1, fd.ci, m_1) \cdots s(fd.co, m_1 + 1) \cdots \qquad \text{for } 0 \leq m_1 \leq 3$$

$$\cdots r(l_1, fd.ci, 4) \cdots s(fd.co, 4) \cdots$$

$$\cdots r(l_1, bd.ci, m_1) \cdots s(bd.co, m_1 - 1) \cdots \qquad \text{for } 1 \leq m_1 \leq 4$$

where $l_1 \in \{ fd.co, bd.co \}$, $p_2 \in \{ fd.ci, bd.ci \}$, and the sequences of events represented by the middle ellipses do not contain any $r(l, p, m)$ or $s(l, m)$ symbols, for $l \in \{ fd.co, bd.co \}$, $p \in \{ fd.ci, bd.ci \}$, and $0 \leq m \leq 4$.

Figure 12

Characterization of the flow of messages through the *ct-out* links of the front door and back door processes

of $fd.co$ or $bd.co$, p_0 is one of $fd.ci$ or $bd.ci$, and $0 \leq m_0 \leq 4$. Applying proposition (3.2), first to the prefix of u ending with the $r(l_0, p_0, m_0)$ symbol, and then to the prefix of u ending with the $r(l_2, p_2, m_2)$ symbol, we again conclude that $l_1 = l_2$ and $m_1 = m_2$. Thus, (i) is established.

To establish (ii), assume that u contains a $r(l_1, fd.ci, m_1)$ symbol, where $l_1 \in \{ fd.co, bd.co \}$ and $0 \leq m_1 \leq 3$, and that this $r(l_1, fd.ci, m_1)$ symbol is not the last symbol from S in u. Let v denote the prefix of u that ends with the $r(l_1, fd.ci, m_1)$ symbol. Then the last symbol in $\rho_{S_{fd}}(v)$ is an $r(l_1, fd.ci, m_1)$ symbol. As it is not possible that the last symbol in both $\rho_{S_{fd}}(v)$ and $\rho_{S_{bd}}(v)$ is a reception symbol (by the proof of proposition (3.2)), the last symbol in $\rho_{S_{bd}}(v)$ is an $s(p_0, m_0)$ symbol, where $p_0 \in \{ fd.ci, bd.ci \}$ and $0 \leq m_0 \leq 4$. Examination of the projected process expressions (see figure 11), therefore, reveals that the next symbol from S in u (following the $r(l_1, fd.co, m_1)$ symbol) is either a $s(fd.co, m_1 + 1)$ symbol or a $r(l_2, bd.ci, m_2)$ symbol, where $l_2 \in \{ fd.co, bd.co \}$ and $0 \leq m_2 \leq 4$. But it is not a reception symbol, and so we conclude that the next symbol from S

in u is an $s(fd.co, m_1 + 1)$ symbol, as desired.

The results (iii) and (iv) are established in an analogous fashion to complete the proof. ∎

The value of the message residing in the *ct-out* links

Having shown that the desired mutually exclusive use of the links *fd.co* and *bd.co* is achieved and that the front door and back door processes update the single message residing in these links as intended, we now show how the value of the message transmitted to these links is related to the state of the system at the time of its transmission.

To motivate the characterization of this relationship, we consider how the system Σ' is intended to behave. The front door and back door attendants, which are modeled in Σ' by the front door and back door processes, are supposed to maintain a count of the number of philosophers in the dining room at any given point in time. To be completely accurate, of course, this count must be updated exactly as a philosopher enters or leaves the dining room. In Σ', however, each of these occurrences is modeled by a single atomic event, and so they cannot be represented as taking place simultaneously. To determine the precise relationship between the value of the count and the activities of the philosophers, as modeled in Σ', therefore, we examine the order in which these events occur more closely.

Since the front door attendant increments the count (modeled by statements FD5–6) before granting a philosopher permission to enter the dining room (FD8) and the back door attendant decrements the count (BD6–7) after observing that a philosopher has left the dining room (BD5), the count should provide an upper bound on the difference between the number of times the front door attendant has given a philosopher permission to enter the dining room and the number of times the back door attendant has observed a philosopher leave the dining room (and hence, on the number of philosophers in the dining room at any point in time, so

that the front door attendant never inadvertantly admits too many philosophers). After incrementing the count, furthermore, the front door process signals a philosopher permission to enter before incrementing the count again, and, after observing a philosopher leave the dining room, the back door process decrements the count before checking to see if another philosopher has left the room. The count, therefore, is as close to this difference as can be reasonably expected. It is off by one if a philosopher has not yet been signaled permission to enter since it was last incremented, and off by (another) one if it has not been decremented since a philosopher was last noticed leaving the dining room. This characterization of the relationship between the count (represented by the value of the last message sent to one of the *ct-out* links in a prefix of a legal behavioral trace of the system) and the difference between the number of times the front door attendant has given a philosopher permission to enter the dining room (represented by the number of $s(fd.s, go)$ symbols in such a prefix) and the number of times the back door attendant has observed a philosopher leaving the dining room (represented by the number of $r(p_i.e, bd.e, tap)$ symbols) is formalized in the next proposition.

For this proposition, we focus on the constraints $\kappa'_5(l, m)$ that relate to those communications through the *ct-out*, *signal* and *exit* links that model the events of interest. We therefore define an alphabet,

$$S = \{\, s(l,m), r(l,p,m) \,\}_{\substack{l \in \{fd.co, bd.co, fd.s, p_i.e\}_{0 \le i \le 4} \\ p \in T'(l), m \in I \cup \{go, tap\}}} . \tag{3.4}$$

We partition S into the sets $S_{fd} = S \cap A'_{fd}$, $S_{bd} = S \cap A'_{bd}$, and $S_{p_i} = S \cap A'_{p_i}$, for $0 \le i \le 4$, determined by the alphabets of the processes, and project the process expressions on S to obtain the "process expressions" for the projected system expression $\rho_S(\epsilon')$. For future reference, the projected process expressions are shown in figure 13.

To facilitate the statement and proof of the proposition, we introduce terminology for referring to prefixes of legal behavioral traces of Σ in which the front

$$\rho_S(\pi'_{fd}) = \Big(\bigvee_{1\le m\le 3} r(fd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee \bigvee_{0\le m\le 3} r(bd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee\, r(fd.co, fd.ci, 4)\, s(fd.co, 4)\Big)^*$$

$$\rho_S(\pi'_{bd}) = s(bd.co, 0)\,\beta^* \vee \bigvee_{0\le j\le 4} s(bd.co, 0)\,\beta^*\, r(p_j.e, bd.e, tap)$$

$$\vee \bigvee_{0\le j\le 4} s(bd.co, 0)\,\beta^* r(p_j.e, bd.e, tap)\, r(bd.co, bd.ci, 0)\text{, where}$$

$$\beta = \bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(fd.co, bd.ci, m)\, s(bd.co, m-1)$$

$$\vee \bigvee_{\substack{1\le m\le 3\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(bd.co, bd.ci, m)\, s(bd.co, m-1)$$

$$\rho_S(\pi'_{p_i}) = \Big(r(fd.s, p_i.s, go)\, s(p_i.e, tap)\Big)^* \vee \Big(r(fd.s, p_i.s, go)\, s(p_i.e, tap)\Big)^*\, r(fd.s, p_i.s, go)$$

Figure 13

Projection of the process expressions on the alphabet S defined in (3.4)

door attendant is modeled as having incremented the count but not yet given another philosopher permission to enter and (or) the back door attendant is modeled as having noticed that another philosopher has left the dining room but not yet decremented the count. After execution of the event sequence corresponding to a prefix of a legal behavioral trace of the system, therefore, we consider the front door process to be in a "stable state" if, for each time it has incremented the value of the message residing in the *ct-out* links, it has also sent a message of type *go* to its *signal* port. Clearly, this is the case as long as the prefix, when projected on the alphabet S_{fd}, does not have a tail of the form $r(l, fd.ci, m)\, s(fd.co, m+1)$, for any $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 3$. Similarly, we consider the back door process to be in a stable state if, for each time that it has received a message of type *tap* through its *exit* port, it has also decremented the value of the message residing in the *ct-out* links. The back door process is thus in a stable state as long as the prefix, when projected on the alphabet S_{bd}, does not end with a $r(p_i.e, bd.e, tap)$ symbol, for any $0 \leq i \leq 4$. This motivates the following definition.

(3.5) *Definition.* A prefix v of a string $u \in \mathcal{L}(\epsilon')|_{\widehat{C}'}$ is *fd-stable* if $\rho_{S_{fd}}(v)$ does not have a tail of the form $r(l, fd.ci, m)\, s(fd.co, m+1)$, for any $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 3$, and *bd-stable* if $\rho_{S_{bd}}(v)$ does not have a tail of the form $r(p_i.e, bd.e, tap)$, for any $0 \leq i \leq 4$.

(3.6) **Proposition.** *If v is a prefix of a string $u \in \mathcal{L}(\epsilon')|_{\widehat{C}'}$ and the last symbol in v is a $s(l_1, m_1)$ symbol, where $l_1 \in \{fd.co, bd.co\}$ and $0 \leq m_1 \leq 4$, then*

(i) $m_1 = |s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$, *if v is both fd-stable and bd-stable,*

(ii) $m_1 - 1 = |s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$, *if v is either fd-stable or bd-stable, but not both, and*

(iii) $m_1 - 2 = |s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$, *if v is neither fd-stable nor bd-stable.*

Proof. We induct on $\sum_{\substack{l\in\{fd.co,bd.co\}\\0\le m\le 4}} |s(l,m)|_v$.

For the basis, we observe that if $\sum_{\substack{l\in\{fd.co,bd.co\}\\0\le m\le 4}} |s(l,m)|_v = 1$ then $m_1 = 0$ and v is both *fd*-stable and *bd*-stable. Hence, we have

$$|s(fd.s, go)|_v - \sum_{0\le j\le 4} |r(p_j.e, bd.e, tap)|_v = 0 = m_1,$$

as desired.

For the inductive step, assume $\sum_{\substack{l\in\{fd.co,bd.co\}\\0\le m\le 4}} |s(l,m)|_v > 1$. Then proposition (3.3), which characterizes the flow of messages through the *ct-out* links (see figure 12) implies that v has one of the following three forms:

$$\cdots s(l_0, m_1+1)\cdots r(l_0, bd.ci, m_1+1)\cdots s(bd.co, m_1),$$

$$\cdots s(l_0, m_1-1)\cdots r(l_0, fd.ci, m_1-1)\cdots s(fd.co, m_1), \text{ or}$$

$$\cdots s(l_0, 4)\cdots r(l_0, fd.ci, 4)\cdots s(fd.co, 4),$$

where $l_0 \in \{fd.co, bd.co\}$ and the sequences of event symbols represented by the last two ellipses in each of the above three patterns do not contain any $s(l,m)$ or $r(l,p,m)$ symbols, for $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$, and $0 \le m \le 4$. We consider each of these cases separately.

Case I: v has the form

$$\frac{\underbrace{\cdots s(l_0, m_1+1)}_{v'}\ \underbrace{\cdots r(l_0, bd.ci, m_1+1)\cdots s(bd.co, m_1)}_{v''}}{v}$$

Let v' denote the prefix of v that ends with the explicit $s(l_0, m_1+1)$ symbol, and v'' denote the tail of v defined by $v = v'v''$. Observe that v is *bd*-stable. As v may or may not be *fd*-stable, however, we distinguish two subcases.

Case I.A: v is *fd*-stable. Since v is *fd*-stable and proposition (3.3) implies that $\rho_{S_{fd}}(v)$ does not end with a $r(l, fd.ci, m)$ symbol, for $l \in \{fd.co, bd.co\}$ and

$0 \leq m \leq 3$, we conclude that $\rho_{S_{fd}}(v)$ ends with a $s(fd.s, go)$ symbol or with the tail $r(fd.co, fd.ci, 4)\, s(fd.co, 4)$.

Case I.A.1: $\rho_{S_{fd}}(v)$ ends with a $s(fd.s, go)$ symbol. If the final $s(fd.s, go)$ symbol of $\rho_{S_{fd}}(v)$ appears in the prefix v' of v, then $\rho_{S_{fd}}(v) = \rho_{S_{fd}}(v')$, and so v' is fd-stable and the tail v'' of v does not contain any $s(fd.s, go)$ symbols. Also, $l_0 = bd.co$, and so v' is bd-stable and v'' contains a single $r(p_j.e, bd.e, tap)$ symbol (assured by the form of $\rho_S(\pi'_{bd})$, which is shown in figure 13, and the assumption that there are no $s(l, m)$ or $r(l, p, m)$ symbols, for $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$ and $0 \leq m \leq 4$, within the last two ellipses in the pattern above). We therefore conclude that

$$\begin{aligned} |s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v & \\ &= |s(fd.s, go)|_{v'} - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'} - 1 \\ &= (m_1 + 1) - 1 = m_1, \end{aligned}$$

where the first equality follows from the above observations and the second from the inductive hypothesis.

Otherwise, the final $s(fd.s, go)$ symbol of $\rho_{S_{fd}}(v)$ appears in the tail v'' of v, and so v' is not fd-stable (assured by the form of $\rho_S(\pi'_{fd})$ and the assumption that there are no $s(l, m)$ or $r(l, p, m)$ symbols, for $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$ and $0 \leq m \leq 4$, within the last two ellipses of the pattern above). Now, if v' is bd-stable, then $\rho_{S_{bd}}(v')$ ends with a $s(bd.co, m)$ symbol, for some $0 \leq m \leq 3$ (assured by proposition (3.3) and the form of $\rho_S(\pi'_{bd})$), and so v'' contains a single $r(p_i.e, bd.e, tap)$ symbol, for $0 \leq i \leq 4$ (assured by the form of $\rho_S(\pi'_{bd})$ and the assumption that v'' contains a $r(l_0, bd.ci, m_1 + 1)$ symbol). As the final $s(fd.s, go)$ symbol of $\rho_{S_{fd}}(v)$ is the only $s(fd.s, go)$ symbol in v'',

we then have

$$|s(fd.s, go)|_v - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_{v'}$$
$$= m_1 ,$$

where, again, the first equality follows from the above observations and the second from the inductive hypothesis. On the other hand, if v' is not bd-stable, then v'' does not contain an $r(p_i.e, bd.e, tap)$ symbol, for any $0 \le i \le 4$, and so we have

$$|s(fd.s, go)|_v - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_{v'} + 1$$
$$= (m_1 - 1) + 1 = m_1 ,$$

completing case I.A.1.

Case I.A.2: $\rho_{S_{fd}}(v)$ has a tail of the form $r(fd.co, fd.ci, 4)\, s(fd.co, 4)$. In this case we have $\rho_{S_{fd}}(v) = \rho_{S_{fd}}(v')$, and so v'' does not contain any $s(fd.s, go)$ symbols and v' is fd-stable. Now, if v' is also bd-stable, then $\rho_{S_{bd}}(v')$ ends with an $s(bd.co, m)$ symbol, for some $0 \le m \le 3$, and so v'' contains a single $r(p_i.e, bd.e, tap)$ symbol. We therefore conclude that

$$|s(fd.s, go)|_v - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \le j \le 4} |r(p_j.e, bd.e, tap)|_{v'} - 1$$
$$= (m_1 + 1) - 1 = m_1 ,$$

as desired. On the other hand, if v' is not bd-stable, then v'' does not contain any

$r(p_i.e, bd.e, tap)$ symbols, for $0 \leq i \leq 4$, and so we conclude that

$$|s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'}$$
$$= m_1,$$

as desired. This completes case I.A.2, which in turn completes case I.A.

Case I.B: v is not fd-stable. Then $\rho_{S_{fd}}(v)$ ends with an $s(fd.co, m)$ symbol, for some $1 \leq m \leq 4$, and so $\rho_{S_{fd}}(v) = \rho_{S_{fd}}(v')$. Hence, v' is not fd-stable and v'' does not contain any $s(fd.s, go)$ symbols. Now, if v' is bd-stable, then $\rho_{S_{bd}}(v')$ ends with an $s(bd.co, m)$ symbol, for some $0 \leq m \leq 3$, and so v'' contains a single $r(p_i.e, bd.e, tap)$ symbol, for $0 \leq i \leq 4$. We therefore have

$$|s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'} - 1$$
$$= m_1 - 1,$$

as desired. On the other hand, if v' is not bd-stable, then v'' does not contain any $r(p_i.e, bd.e, tap)$ symbols, for $0 \leq i \leq 4$, and we have

$$|s(fd.s, go)|_v - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_v$$
$$= |s(fd.s, go)|_{v'} - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'}$$
$$= m_1 - 1,$$

as desired. This completes case I.B, which in turn completes case I. The arguments for the remaining cases are similar. ∎

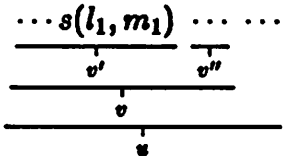
Using this characterization of the relationship between the value of a message transmitted to one of the *ct-out* links and the state of the system when the message

is transmitted, we now show that the value of the message residing in one of these links provides an upper bound on the number of philosopher processes executing their critical sections.

(3.7) **Proposition.** *If v is a prefix of a string $u \in \mathcal{L}(\epsilon')|_{\widehat{C}}$, and the last $s(l,m)$ symbol in v, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, is an $s(l_1,m_1)$ symbol, then*

$$\begin{aligned} m_1 &\geq |s(fd.s, go)|_v - \sum_{0\leq j\leq 4} |r(p_j.e, bd.e, tap)|_v \\ &\geq \sum_{0\leq j\leq 4} |r(fd.s, p_j.s, go)|_v - \sum_{0\leq j\leq 4} |s(p_j.e, tap)|_v \end{aligned}$$

Proof. Let v' denote the prefix of u that ends with the last $s(l_1,m_1)$ symbol in v and v'' denote the tail of v satisfying $v = v'v''$, as shown below.



Now, v'' does not contain any $s(fd.co, m)$ symbols, and so it contains, at most, a single $s(fd.s, go)$ symbol. If v'' does not contain any $s(fd.s, go)$ symbols, we therefore reason that

$$\begin{aligned} |s(fd.s, go)|_v - \sum_{0\leq j\leq 4} |r(p_j.e, bd.e, tap)|_v & \\ \leq |s(fd.s, go)|_{v'} - \sum_{0\leq j\leq 4} |r(p_j.e, bd.e, tap)|_{v'} &\leq m_1 , \end{aligned}$$

where the last inequality follows from the previous proposition. On the other hand, if v'' contains a $s(fd.s, go)$ symbol, we reason that

$$\begin{aligned} |s(fd.s, go)|_v - \sum_{0\leq j\leq 4} |r(p_j.e, bd.e, tap)|_v & \\ \leq |s(fd.s, go)|_{v'} - \sum_{0\leq j\leq 4} |r(p_j.e, bd.e, tap)|_{v'} + 1 & \\ \leq (m_1 - 1) + 1 , & \end{aligned}$$

where the last inequality follows from the previous proposition and the observation that v' is not fd-stable. This establishes the first of the desired inequalities.

The second inequality follows easily from the fact that u satisfies the constraints $\kappa_5'(fd.s, go)$ and $\kappa_5'(p_j.e, tap)$, for $0 \leq j \leq 4$. ∎

Clearly, the number of philosopher processes within their critical sections following execution of the sequence of events represented by the prefix w is given by the difference on the right hand side of the second inequality in the proposition above.

Pruning the process expressions

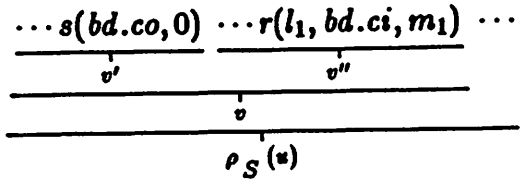
Having shown that the single message residing in either of the links $bd.co$ or $fd.co$ is updated as intended, and having established the characterizing relationship between the value of this message and the state of the system when the message is transmitted, we now show that the back door process never receives a message of type 0 through its *ct-in* port. This allows us to prune the alternative involving the $ill(bd)$ symbol from the process expression π_{bd}' (see figure 10).

(3.8) **Proposition.** *If* $u \in \mathcal{L}(\epsilon')|_{\widehat{\mathcal{O}}'}$, *then there are no* $r(bd.co, bd.ci, 0)$ *symbols in* u.

Proof. Because of the manner in which messages flow through the links $fd.co$ and $bd.co$ (see figure 12), we know that if u, and hence $\rho_S(u)$, where S is defined in (3.4), contains an $r(bd.co, bd.ci, 0)$ symbol, then the last $s(l, m)$ symbol in $\rho_S(u)$, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, preceding this $r(bd.co, bd.ci, 0)$ symbol is an $s(bd.co, 0)$ symbol. We prove the proposition, therefore, by showing that between every $s(bd.co, 0)$ symbol and the next $r(l_1, bd.ci, m_1)$ symbol in $\rho_S(u)$, there is an intervening $r(l_0, fd.ci, m_0)$ symbol, where $l_1, l_0 \in \{fd.co, bd.co\}$ and $0 \leq m_1, m_0 \leq 4$.

Let v' denote the prefix of $\rho_S(u)$ that ends with a given $s(bd.co, 0)$ symbol, v denote the prefix of $\rho_S(u)$ that ends with the next $r(l_1, bd.ci, m_1)$ symbol,

where $l_1 \in \{fd.co, bd.co\}$ and $0 \leq m_1 \leq 4$, and v'' denote the tail of v satisfying $v = v'v''$. We therefore have the following picture,



where there are no $r(l, bd.ci, m)$ symbols, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, within the middle ellipses. We show that v'' contains a $r(l_0, fd.ci, m_0)$ symbol, where $l_0 \in \{fd.co, bd.co\}$ and $0 \leq m_0 \leq 4$.

For this, we first observe that

$$\begin{aligned}\sum_{0 \leq j \leq 4} |s(p_j.e, tap)|_{v'} &\geq \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'} \\ &\geq |s(fd.s, go)|_{v'} \\ &\geq \sum_{0 \leq j \leq 4} |r(fd.s, p_j.s, go)|_{v'},\end{aligned}$$

where the first inequality follows from the assumption that $\rho_S(u)$ satisfies the constraints $\kappa_5'(p_j.e, tap)$, for $0 \leq j \leq 4$, the second inequality follows from proposition (3.7) (applied to the prefix v'), and the third inequality follows from the assumption that $\rho_S(u)$ satisfies the constraint $\kappa_5'(fd.s, go)$. From the form of the projected process expression $\rho_S(\pi'_{p_i})$ (see figure 13), however, we reason that $|r(fd.s, p_i.s, go)|_{v'} \geq |s(p_i.e, tap)|_{v'}$, for $0 \leq i \leq 4$. The quantities in the above string of inequalities must therefore be equal. From this, the assumption that $\rho_S(u)$ satisfies the constraints $\kappa_5'(p_j.e, tap)$ and the form of the projected process expressions $\rho_S(\pi'_{p_j})$, for $0 \leq j \leq 4$, we conclude that

(i) $|s(fd.s, go)|_{v'} = \sum_{0 \leq j \leq 4} |r(fd.s, p_j.s, go)|_{v'}$,

(ii) $|s(p_j.e, tap)|_{v'} = |r(p_j.e, bd.e, tap)|_{v'}$, for $0 \leq j \leq 4$,

(iii) $|s(p_j.e, tap)|_{v'} = |r(fd.s, p_j.s, go)|_{v'}$, for $0 \leq j \leq 4$, and

(iv) $|s(fd.s, go)|_{v'} = \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'}$.

We now consider the string v''. From the form of the projected process expression $\rho_S(\pi'_{bd})$, we see that v'' contains an $r(p_i.e, bd.e, tap)$ symbol, for some $0 \leq i \leq 4$. Since $|s(p_i.e, tap)|_{v'} = |r(p_i.e, bd.e, tap)|_{v'}$ (by (ii) above) and $\rho_S(u)$ satisfies the constraint $\kappa'_5(p_i.e, tap)$, this implies that v'' also contains an $s(p_i.e, tap)$ symbol. But $|s(p_i.e, tap)|_{v'} = |r(fd.s, p_i.s, go)|_{v'}$ (by (iii) above), and so, because of the form of the projected process expression $\rho_S(\pi'_{p_i})$, we conclude that v'' contains an $r(fd.s, p_i.s, go)$ symbol. From the result (i) above and the assumption that $\rho_S(u)$ satisfies the constraint $\kappa'_5(fd.s, go)$, we therefore conclude that v'' also contains an $s(fd.s, go)$ symbol. If v'' does not contain an $r(l_0, fd.ci, m_0)$ symbol, for any $l_0 \in \{fd.co, bd.co\}$ and $0 \leq m_0 \leq 4$, therefore, the string v' is not fd-stable, and so, by proposition (3.6), we have $0 - 1 = |s(fd.s, go)|_{v'} - \sum_{0 \leq j \leq 4} |r(p_j.e, bd.e, tap)|_{v'}$. But this contradicts observation (iv) above. Hence, v'' contains an $r(l_0, fd.ci, m_0)$ symbol, as desired. ∎

We now show that the subterms of π'_{bd} and π'_{fd} containing a $w(p)$ symbol, for $p \in \{fd.ci, bd.ci\}$, can also be eliminated.

(3.9) **Proposition.** *If $u \in \mathcal{L}(\epsilon')|_{\widehat{C}'}$, there are no $w(bd.ci)$ or $w(fd.ci)$ symbols in u.*

Proof. To prove this proposition, we focus on the constraints $\kappa'_6(l, p, m)$, for $l \in \{fd.co, bd.co\}$, $p \in \{fd.ci, bd.ci\}$, and $0 \leq m \leq 4$. We therefore define $S = \{s(l, m), r(l, p, m), w(p)\}$, where l takes on the values $fd.co$ and $bd.co$, p takes on the values $fd.ci$ and $bd.ci$, and $0 \leq m \leq 4$, and we partition S into the sets $S_{fd} = S \cap A'_{fd}$ and $S_{bd} = S \cap A'_{bd}$. Examining the form the of process expressions π'_{fd} and π'_{bd} (see figure 10), we then observe that $\rho_{S_{fd}}(u)$ lies in the

language of

$$\Big(\bigvee_{\substack{l\in\{fd.co,bd.co\}\\0\le m,m'\le 4}} r(l,fd.ci,m)\,s(fd.co,m')\Big)^{j}$$
$$\vee\Big(\bigvee_{\substack{l\in\{fd.co,bd.co\}\\0\le m,m'\le 4}} r(l,fd.ci,m)\,s(fd.co,m')\Big)^{j} w(fd.ci),$$

for some $j \geq 0$ and that $\rho_{S_{bd}}(u)$, since it does not contain any $r(bd.co, bd.ci, 0)$ symbols (by the previous proposition), lies in the language of

$$s(bd.co,0)\Big(\bigvee_{\substack{l\in\{fd.co,bd.co\}\\0\le m,m'\le 4}} r(l,bd.ci,m)\,s(bd.co,m')\Big)^{k}$$
$$\vee\, s(bd.co,0)\Big(\bigvee_{\substack{l\in\{fd.co,bd.co\}\\0\le m,m'\le 4}} r(l,bd.ci,m)\,s(bd.co,m')\Big)^{k} w(bd.ci),$$

for some $k \geq 0$.

Now, if u contains a $w(bd.ci)$ symbol, then since $\rho_S(u)$ satisfies the constraints $\kappa_6'(l, bd.ci, m)$, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$, we conclude that

$$j+k+1 = \sum_{0\le m\le 4} |s(fd.co,m)| + \sum_{0\le m\le 4} |s(bd.co,m)|$$
$$= \sum_{\substack{p\in\{fd.ci,bd.ci\}\\0\le m\le 4}} |r(fd.co,p,m)| + \sum_{\substack{p\in\{fd.ci,bd.ci\}\\0\le m\le 4}} |r(bd.co,p,m)| = j+k,$$

which is clearly impossible. Hence, u does not contain a $w(bd.ci)$ symbol.

If u is assumed to contain a $w(fd.ci)$ symbol, a contradiction is obtained in exactly the same manner. ∎

Finally, we show that the subterms of π'_{p_i}, for $0 \leq i \leq 4$, containing a $w(p_i.s)$ symbol can also be pruned.

(3.10) **Proposition.** *If $u \in \mathcal{L}(\epsilon')|_{\widehat{\mathcal{O}}}$, then u does not contain a $w(p_i.s)$ symbol, for $0 \leq i \leq 4$.*

Proof. To prove this proposition, we focus on the constraints $\kappa_6'(fd.s, p_i.s, m)$, for $0 \leq i \leq 4$ and $m \in \{stay, go\}$, and $\kappa_6'(p_i.k, fd.k, tap)$, for $0 \leq i \leq 4$. We therefore define

$$S = \{ s(fd.s, m), r(fd.s, p_i.s, m), w(p_i.s), s(p_i.k, tap), r(p_i.k, fd.k, tap) w(fd.k) \},$$

where m ranges over the set $\{stay, go\}$, and $0 \leq i \leq 4$, and partition S into the sets $S_{fd} = S \cap A'_{fd}$ and $S_{p_i} = S \cap A'_{p_i}$, for $0 \leq i \leq 4$. We then observe that $\rho_{S_{p_i}}(u)$, for $0 \leq i \leq 4$, lies in the language of

$$\Big(s(p_i.k, tap)\, r(fd.s, p_i.s, go) \vee s(p_i.k, tap)\, r(fd.s, p_i.s, stay)\Big)^{k_i \oplus l_i}$$
$$\vee \Big(s(p_i.k, tap)\, r(fd.s, p_i.s, go) \vee s(p_i.k, tap)\, r(fd.s, p_i.s, stay)\Big)^{k_i \oplus l_i}$$
$$s(p_i.k, tap)\, w(p_i.s),$$

for some $k_i, l_i \geq 0$, and that $\rho_{S_{fd}}(u)$, since it does not contain a $w(fd.ci)$ symbol (by the previous proposition), lies in the language of

$$\Big[\Big(\bigvee_{0 \leq j \leq 4} r(p_j.k, fd.k, tap)\, s(fd.s, go)\Big) \vee \Big(\bigvee_{0 \leq j \leq 4} r(p_j.k, fd.k, tap)\, s(fd.s, stay)\Big)\Big]^{h \oplus t}$$
$$w(fd.k),$$

for some $h, t \geq 0$.

The string $\rho_S(u)$, therefore, contains a $w(fd.k)$ symbol. As it also satisfies the constraints $\kappa_6'(p_i.k, fd.k, tap)$, for $0 \leq i \leq 4$, we have $\sum_{0 \leq i \leq 4} |s(p_i.k, tap)| = \sum_{0 \leq i \leq 4} |r(p_i.k, fd.k, tap)|$, where $|a|$ denotes $|a|_{\rho_S(u)} = |a|_u$, for $a \in S$. Now, $\sum_{0 \leq i \leq 4} |r(p_i.k, fd.k, tap)| = t + h$, and, for $0 \leq i \leq 4$, $|s(p_i.k, tap)| = k_i + l_i$, if $\rho_S(u)$ does not contain a $w(p_i.s)$ symbol, and $|s(p_i.k, tap)| = k_i + l_i + 1$, if $\rho_S(u)$

does contain a $w(p_i.s)$ symbol. We therefore reason that

$$\sum_{0\le i\le 4} k_i + \sum_{0\le i\le 4} l_i + \sum_{0\le i\le 4} |w(p_i.s)| = \sum_{0\le i\le 4} |s(p_i.k, tap)|$$
$$= \sum_{0\le i\le 4} |r(p_i.k, fd.k, tap)| = t + h.$$

Thus, if u contains a $w(p_j.s)$ symbol, for some $0 \le j \le 4$, we conclude that $\sum_{0\le i\le 4} k_i + \sum_{0\le i\le 4} l_i < t + h$.

If u, and hence $\rho_S(u)$, contains a $w(p_j.s)$ symbol, however, then the constraint $\kappa_6'(fd.s, p_j.s, go)$ implies that $h = |s(fd.s, go)| = \sum_{0\le i\le 4} |r(fd.s, p_i.s, go)| = \sum_{0\le i\le 4} k_i$, and the constraint $\kappa_6'(fd.s, p_j.s, stay)$ implies that $t = |s(fd.s, stay)| = \sum_{0\le i\le 4} |r(fd.s, p_i.s, stay)| = \sum_{0\le i\le 4} l_i$. As this contradicts the conclusion obtained above, the assumption that u contains a $w(p_j.s)$ symbol must be incorrect. ∎

The propositions in this subsection justify the use of the process expressions shown in figure 14 (in place of the process expressions shown in figure 10).

Using the analysis of the original system Σ

We now show how the results obtained in section §1, which pertain to behaviors of the original system Σ, can be used in the analysis of the system Σ'. For this, we take ϵ' to be the interleave of the process expressions shown in figure 14, $\hat{C}'$ to consist of the constraints $\kappa_5'(l, m)$ and $\kappa_6'(l, p, m)$, for $l \in L'$, $p \in T'(l)$, and $m \in M' - \{@\}$, and ϵ and $\hat{C}$ to be the system expression and constraints shown in figures 1 and 2, which were used for the analysis in section §1. Defining the alphabet S to be the union of the constraint alphabets of the constraints in $\hat{C}$, we then apply proposition (V.4.1) (with $B = S$ and $\hat{D} = \hat{C}$) to conclude that

$$\rho_S\left(\mathcal{L}(\epsilon')|_{\hat{C}'}\right) \subseteq \mathcal{L}(\epsilon)|_{\hat{C}}. \tag{3.11}$$

Together with the propositions (1.8), (1.11) and (1.13), this implies the following propositions.

$$\pi'_{fd} = \Big(\bigvee_{\substack{0\le i\le 4\\ 1\le m\le 8}} r(p_i.k, fd.k, tap)\, r(fd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee \bigvee_{\substack{0\le i\le 4\\ 0\le m\le 8}} r(p_i.k, fd.k, tap)\, r(bd.co, fd.ci, m)\, s(fd.co, m+1)\, s(fd.s, go)$$

$$\vee \bigvee_{0\le i\le 4} r(p_i.k, fd.k, tap)\, r(fd.co, fd.ci, 4)\, s(fd.co, 4)\, s(fd.s, stay) \Big)^* w(fd.k)$$

$$\pi'_{bd} = s(bd.co, 0) \Big(\bigvee_{\substack{1\le m\le 4\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(fd.co, bd.ci, m)\, s(bd.co, m-1)$$

$$\vee \bigvee_{\substack{1\le m\le 8\\ 0\le i\le 4}} r(p_i.e, bd.e, tap)\, r(bd.co, bd.ci, m)\, s(bd.co, m-1) \Big)^* w(bd.e)$$

$$\pi'_{p_i} = \gamma^* \vee \gamma^*\, s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, w(p_i.rx)$$

$$\vee \gamma^*\, s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, r(f_{i-1}.x, p_i.rx, ok)\, w(p_i.lx)\text{, where}$$

$$\gamma = s(p_i.k, tap)\, r(fd.s, p_i.s, go)\, r(f_{i-1}.x, p_i.rx, ok)\, r(f_i.x, p_i.lx, ok)$$

$$s(p_i.ld, ok)\, s(p_i.rd, ok)\, s(p_i.e, tap)$$

$$\vee\ s(p_i.k, tap)\, r(fd.s, p_i.s, stay)$$

$$\pi'_{f_i} = \Big(s(f_i.x, ok)\, r(p_{i+1}.rd, f_i.d, ok) \vee s(f_i.x, ok)\, r(p_i.ld, f_i.d, ok) \Big)^* s(f_i.x, ok)\, w(f_i.d)$$

Figure 14

Process expressions obtained when the process expressions of figure 10 are pruned

(3.12) **Proposition.** *If* $u \in \mathcal{L}(\epsilon')|_{\widehat{C'}}$ *then there are no* $w(p_i.ru)$ *symbols in* u, *for* $0 \leq i \leq 4$.

(3.13) **Proposition.** *If* $u \in \mathcal{L}(\epsilon')|_{\widehat{C'}}$ *contains a* $w(p_i.lu)$ *symbol, then it also contains a* $w(p_{i+1}.lu)$ *symbol, for* $0 \leq i \leq 4$.

(3.14) **Proposition.** *If* $u \in \mathcal{L}(\epsilon')|_{\widehat{C'}}$ *contains an* $r(f_i.u, p, ok)$ *symbol that is followed by an* $r(f_i.u, p', ok)$ *symbol, for ports* $p, p' \in \{p_i.lu, p_{i+1}.ru\}$, *then, if* $p = p_i.lu$, *there is an intervening* $r(p_i.lu, f_i.u, ok)$ *symbol, and, if* $p = p_{i+1}.ru$, *there is an intervening* $r(p_{i+1}.ru, f_i.u, ok)$ *symbol.*

Since every legal behavioral trace of Σ', when projected on the alphabets of the constraints $\kappa_5'(l, m)$ and $\kappa_6'(l, p, m)$, for $l \in L'$, $p \in T'(l)$, and $m \in M' - \{@\}$, belongs to $\mathcal{L}(\epsilon')|_{\widehat{C'}}$ (assured by proposition (V.3.2) and the results of the previous subsection), we conclude that a philosopher process in Σ' cannot wait indefinitely for a communication through its *right-up* port, and that the intended synchronization of philosopher processes is achieved in Σ'.

Finally, we show that a string from $\mathcal{L}(\epsilon')|_{\widehat{C'}}$ cannot contain a $w(p_i.lu)$ symbol, which implies that, in the modified system Σ', a philosopher process cannot wait indefinitely for a communication through its *left-up* port.

(3.15) **Proposition.** *If* $u \in \mathcal{L}(\epsilon')|_{\widehat{C'}}$, *then* u *does not contain a* $w(p_i.lu)$ *symbol, for* $0 \leq i \leq 4$.

Proof. Proposition (3.13) implies that if u contains a $w(p_i.lu)$ symbol, for some $0 \leq i \leq 4$, then it contains a $w(p_i.lu)$ symbol, for all $0 \leq i \leq 4$. Examining the form of the process expressions π'_{p_i}, for $0 \leq i \leq 4$, we reason that if u contains a $w(p_i.lu)$ symbol, for all $0 \leq i \leq 4$, then $\sum_{0 \leq j \leq 4} |r(fd.s, p_j.s, go)| - \sum_{0 \leq j \leq 4} |s(p_j.e, tap)| = 5$, where $|a|$ denotes $|a|_u$. Proposition (3.7), however, implies that $m_1 \geq \sum_{0 \leq j \leq 4} |r(fd.s, p_j.s, go)| - \sum_{0 \leq j \leq 4} |s(p_j.e, tap)|$, where m_1 is the value of the message type m in the last $s(l, m)$ symbol, for $l \in \{fd.co, bd.co\}$ and $0 \leq m \leq 4$,

that appears in u. (We know that u contains such a symbol as the initial expression contains a $s(bd.co, 0)$ symbol.) These later two results are contradictory, since they imply $4 \geq 5$. ∎

CHAPTER IX

SUMMARY AND CONCLUSIONS

§1. Summary

In this dissertation we have described the constrained expression framework and shown how it could be used to help developers of distributed software systems analyze the systems they create. We have explained how constrained expressions are used to represent the possible behaviors of a concurrent system, illustrating the generality of constrained expressions by using them with systems expressed in three fundamentally different notations. We have developed a theory for manipulating constrained expressions. Based on this theory, we have formulated a procedure for simplifying the constrained expression representations of SDYMOL systems. Using this procedure, a normal form for SDYMOL constrained expressions is obtained that facilitates subsequent analysis. Two algorithms, based on standard data flow analysis algorithms, have also been presented for simplifying SDYMOL constrained expressions. An example was presented to illustrate the above techniques. This example was also used to demonstrate analysis techniques, based on the constrained expression framework, that can be used to establish specific behavioral properties of an SDYMOL system. Finally, the example was extended to further illustrate these analysis techniques and show how the analysis of an SDYMOL system could be effectively modularized.

§2. Conclusions

The goal of the research for this dissertation was to investigate the representational power and analytic capability of the constrained expression framework. In particular, we sought to identify techniques, based on the constrained expression framework, that could be used to analyze the behaviors of distributed software systems. Our intention was to assess the potential for developing automated tools based on these techniques to aid the developers of distributed systems in reasoning about the behaviors of the systems they create.

Our results clearly demonstrate the power of the constrained expression framework for describing distributed software systems. It can be used with systems expressed in languages providing different communication primitives (e.g., synchronous or asynchronous message-passing primitives), and based on different underlying models of computation (e.g., Petri nets or state machines). By varying the level of granularity of events, it can be used to describe a single system at differing levels of abstraction and across a number of phases of software development. We did, of course, encounter some limitations in its use. We were not able, for example, to devise a means for representing a specific ordering of buffered messages (e.g., LIFO or FIFO) in asynchronous message-passing systems using constrained expressions. We have yet to investigate the possibility of using constrained expressions to describe performance constraints (e.g., real-time) of distributed software systems. In general, however, the expressive power of the constrained expression framework compares quite favorably with the expressive power of the descriptive notations associated with other techniques for analyzing distributed systems.

The feasibility of providing developers of distributed software systems with automated tools based on the constrained expression framework and supporting the analysis of SDYMOL systems is also demonstrated by the results of our research. Obtaining a reduced constrained expression representation for a system from its

SDYMOL design, as shown in chapters III and VI, is a purely mechanical procedure, and could be easily automated. Likewise, the message flow analysis algorithms described in chapter VII could be readily automated. Automating the analysis of SDYMOL designs would be less straightforward, but is apparently feasible. As illustrated in chapter VIII, the arguments required to establish specific behavioral properties of SDYMOL systems are long and tedious, but they are not particularly deep. The problems that would be encountered when automating this analysis are of a combinatorial nature. Heuristics will be required for directing the search for a proof of specific behavioral properties. Ideally, this knowledge would be built into the tool itself, which could recognize certain commonly occuring patterns of process interactions, such as the exchange of simple timing signals, and the constraints that restrict specific event occurrences. Alternately, the developer could interactively direct the search. For example, specific statements in the design for a system might be intended to enforce a particular restricted access to some shared resource. If these statements are identified by the designer, an automated tool could then focus on the constraints relating to these statements, as illustrated in chapter VIII.

Our research indicates that similar tools could be developed to support the analysis of systems described in many different development languages, in addition to SDYMOL. Clearly, the generation of constrained expression representations for systems expressed in a suitably formal development language is routine, once a procedure for doing so has been devised, and such generation could be easily automated. Many of the techniques for simplifying and analyzing SDYMOL constrained expressions could be applied directly to more general constrained expressions. Others, like the message flow analysis algorithms and certain of the algebraic techniques, which rely on specific SDYMOL constraints, generalize to the extent that other development languages can be expected to require similar constraints. The first message flow analysis algorithm, for example, can be applied to the constrained expressions generated by systems expressed in any language that provides variables that are

referenced and assigned values in the usual fashion. Similarly, the second message flow analysis algorithm should give valid results for most message-passing systems.

Until a prototype toolset is developed, of course, we can only guess at the usefulness of tools based on the constrained expression framework. Our research indicates, however, that such tools could provide important capabilities not often found in tools for analyzing distributed software systems. There are two reasons for this. One is the extreme generality of the constrained expression framework. As noted earlier, this generality would allow the same analysis techniques to be applied to systems described in a variety of development languages and to be extended to provide common analysis techniques across a number of phases in the development process. Moreover, it should be possible to produce tools in which the use of constrained expressions is completely transparent to the user. Software developers would thus be able to use these tools while continuing to work within the languages most appropriate to their tasks.

The second important property that should enhance the usefulness of the constrained expression framework is that it is expressly designed to highlight the most significant aspects of a distributed system's behavior and to limit the combinatorial explosion endemic in analyses of such behaviors. In particular, constrained expression representations provide a closed form, high level, abstract representation of behavior that focuses on concurrency. Thus, constrained expressions facilitate analysis by providing a finite representation of the potentially infinite set of possible behaviors of a distributed system. The analysis methods that exploit the structure of constrained expression representations, furthermore, can be used in a directed, rather than exhaustive manner, to aid developers in formulating arguments regarding the order and number of occurrences of events in possible behaviors of a distributed system.

§3. Future directions

There are a number of directions along which future research on the constrained expression framework could proceed. One of these would be to actually implement and experimentally evaluate a prototype toolset to aid developers in reasoning about the behaviors of SDYMOL systems, based on the techniques described in this dissertation. Important pieces of such a toolset are already in existence. A simulator for DYMOL, of which SDYMOL is a simple subset, has been developed independently of this research [WONG84]. The front end for a *deriver*, a tool that would produce constrained expression representations from SDYMOL designs, thus essentially exists. The simulator, furthermore, is a likely tool to include among a set of prototype tools for analyzing SDYMOL designs.

As part of this research, a senior undergraduate student has implemented a *behavior generator*, which interactively generates sample behaviors from constrained expressions [AVER84]. Such a tool would be essential to aid developers in using information about behaviors possessing specific behavioral properties (i.e., information produced by the analysis techniques) to actually produce example behaviors possessing these properties.

Other tools that would be included in such a toolset include a *simplifier* and *analyzer*. A simplifier would perform simplifications on SDYMOL constrained expressions, putting them into a reduced form, as described in chapter VI, and applying the message flow analysis algorithms described in chapter VII. After simplification, it would be substantially easier to determine and reason about the possible behaviors represented by a constrained expression. A simplifier based on our research should be fairly straightforward to implement.

The analyzer is a generic name for a collection of tools implementing specific analysis techniques applicable to constrained expressions. All of these tools would benefit from the constrained expression simplifications performed by the simplifier.

Among the tools composing the analyzer would be tools supporting the algebraic techniques illustrated in chapter VIII and tools exploiting modularization of designs to simplify analysis. Implementing these tools would be more complex than implementing the other three tools. Among the research issues that would be addressed as part of this effort are methods and heuristics for guiding the application of the rules for generating inequalities that are the basis of the algebraic techniques and extensions of our results on modularization of analysis. The analyzer, comprising these tools and others that might result from the ongoing investigation of constrained expression analysis techniques, would provide extremely valuable aid to developers of distributed systems.

The purpose of implementing a toolset for analyzing SDYMOL designs would be to conduct experimental evaluation of the constrained expression approach to analyzing the behaviors of distributed systems. Additionally, we would explore a variety of enhancements and extensions to this prototype toolset. Among the extensions of particular interest would be applications of the approach to systems with dynamic structure (i.e., systems in which processing units come into existance and disappear dynamically, as with Ada tasks, and/or systems in which interprocess communication pathways are dynamically altered, as in some computer networks) and to systems with real-time constraints. We would also plan to integrate constrained expression tools with other tools for design development and analysis.

Alternately, further research on the constrained expression framework might first be directed at adapting it for use with a more widely used development language than SDYMOL. The SDYMOL design notation was chosen for initial research on the constrained expression framework because it is expressly designed for describing interprocess communication and synchronization, issues of central concern for designers of distributed systems, and yet it possesses a relatively simple semantics when compared to other languages for describing distributed software systems. It was felt that the simplicity of the SDYMOL semantics would facilitate the analysis

of SDYMOL systems. The experimental value of a prototype toolset based on the constrained expression framework might be enhanced, however, if it were designed to be used with a better known and accepted development language. Given the anticipated widespread use of Ada as an implementation language for distributed systems, an Ada-based design language would be an appropriate choice. Designers of distributed software systems would be more receptive to tools supporting the analysis of distributed systems expressed in an appropriate PDL/Ada. Additionally, the value of constrained expression-based tools for analyzing systems expressed in a PDL/Ada could be significantly enhanced by integrating them with other tools for development and analysis of Ada systems. Finally, providing tools to analyze distributed systems expressed in a PDL/Ada would constitute a first step toward providing tools that would span several phases of the software development process. Because of the deliberate similarities between Ada and its various program design languages, tools for analyzing suitable PDL/Ada designs could likely be used with Ada implementations, as well.

The effort to adapt the constrained expression framework for use with systems expressed in an appropriate PDL/Ada would proceed in three directions. First, a procedure for deriving constrained expressions representing the possible behaviors of PDL/Ada designs would be devised. The formulations of SDYMOL and CSP presented in this dissertation should provide valuable guidelines for this task. Second, simplification and analysis techniques appropriate to PDL/Ada constrained expressions would be developed. Clearly, many of the techniques described in this dissertation could be applied to PDL/Ada constrained expressions directly. We expect that the others could be easily modified to provide techniques with equivalent capabilities for simplifying and analyzing PDL/Ada constrained expressions. Third, more powerful techniques for simplifying and analyzing PDL/Ada constrained expressions would be investigated. We expect that the constrained expressions produced by a more complex design language than SDYMOL will be harder to manipulate and reason

about. Stronger techniques may be required, therefore, to obtain useful tools for analyzing the PDL/Ada systems. In particular, we would try to extend our preliminary results on the modularization of analysis and also to develop heuristics to guide the choice of constraints for generating the inequalities used in the algebraic techniques. Since synchronized communication primitives result in more restrictive communication patterns, it may be possible to obtain stronger message flow analysis algorithms than those presented here and to generate stronger inequalities for the algebraic analysis of behaviors exhibiting specific properties.

As an interesting byproduct, results that might be obtained for factoring constrained expressions in ways that allow the analysis of individual factors to be combined to prove properties of the composite system could well have implications for modularization of designs. Factorizations of constrained expressions could indicate what sorts of modularizations are easy to analyze, and so are desirable.

To complete the formal foundation for the constrained expression framework presented in this dissertation, techniques for verifying the derivation of constrained expressions should be investigated. Analysis based on constrained expressions crucially depends on the correctness of procedures for deriving constrained expressions to represent the possible behaviors of distributed systems. Once a procedure for deriving a constrained expression representation of a system expressed in a particular development language is verified, using a suitably formal definition of the language semantics, developers would be assured that the constrained expressions derived from systems expressed in the language describe all and only the possible system behaviors.

A final direction along which we propose conducting further research on the constrained expression framework is towards using the constrained expression framework to compare and classify different development languages. As discussed in the introduction, existing development languages differ in their focus, in the types of objects they use to represent aspects of a distributed software system, and in

the assumptions they make about the general features of these systems. It is quite likely, furthermore, that these different foci are best suited for different applications and in different phases of the development process. Because constrained expressions provide a uniform framework for describing and analyzing systems expressed in a variety of development languages, it may be possible to make such determinations using the constrained expression framework.

SELECTED BIBLIOGRAPHY

AVER84 S. M. Avery. Development of a Behavior Generator for Constrained Expressions, SDLM/84-2, Department of Computer and Information Science, University of Massachusetts, Amherst (June 1984).

AVRU83a G. S. Avrunin and J. C. Wileden. Algebraic Techniques for the Analysis of Concurrent Systems, *Proc. Sixteenth Annual Hawaii International Conference on System Sciences* (1983), 51–57.

AVRU83b G. S. Avrunin and J. C. Wileden. Describing and Analyzing Distributed System Designs (to appear in *ACM Trans. on Programming Languages and Systems*).

BARR79 W. A. Barrett and J. D. Couch. *Compiler Construction: Theory and Practice*, Science Research Associates, Inc. (1979).

BOYE79 R. S. Boyer and J. S. Moore. *A Computational Logic*, Academic Press, New York (1979).

BRIN78 P. Brinch Hansen. Distributed Processes: A Concurrent Programming Language, *Communications of the ACM* (November 1978), 934–941.

CAMP74 R. H. Campbell and A. N. Haberman. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science 6*, Springer Verlag, Heidelberg (1974), 89–102.

CHEN82 B. Chen and R. T. Yeh. Formal specification and verification of distributed systems, *IEEE Trans. on Software Engineering* (November 1983), 710–722.

DILL83 L. K. Dillon. On the Equivalence of Two Constrained Expression Frameworks, SDLM/83-2, Department of Computer and Information Science, University of Massachusetts, Amherst (March 1983).

DOD83 United States Department of Defense. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A, Washington, D.C. (January 1983).

DOEP83 T. W. Doeppner, Jr., and A. Giacalone. A Formal Description of the UNIX Operating System, *Proc. Principles of Distributed Computing* (August 1983), 241–253.

FELD79 J. Feldman. High Level Programming for Distributed Computing, *Communications of the ACM* (June 1979), 353–368.

FOSD76 L. D. Fosdick and L. J. Osterweil. Data Flow Analysis in Software Reliability, *Computing Surveys* (September 1976), 305–330.

GREI77 I. Greif. A language for formal problem specification, *Communications of the ACM* (December 1977), 931–935.

HABE72 A. N. Habermann. Synchronization of Communicating Processes, *Communications of the ACM* (March 1972), 171–176.

HACK75 M. Hack. Petri Net Languages, Computation Structures Group Memo 124, Massachusetts Institute of Technology, Cambridge (June 1975).

HECH77 M. S. Hecht. *Flow Analysis of Computer Programs*, The Computer Science Library, Programming Languages Series, T. E. Cheatham (Ed.), Elsevier North-Holland, Inc., New York 10017 (1977).

HOAR69 C. A. R. Hoare. An Axiomatic Basis for Computer Programming, *Communications of the ACM* (October 1969), 576–583.

HOAR78 C. A. R. Hoare. Communicating Sequential Processes, *Communications of the ACM* (August 1978), 666–677.

HOAR81A C. A. R. Hoare. A Calculus of Total Correctness for Communicating Processes, Technical Monograph PRG-23, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (April 1981).

HOAR81B C. A. R. Hoare, S. D. Brookes and A. W. Roscoe. A Theory of Communicating Sequential Processes, Technical Monograph PRG-16, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (May 1981).

HOAR81C C. A. R. Hoare. A Model for Communicating Sequential Processes, Technical Monograph PRG-22, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (June 1981).

HOAR82 C. A. R. Hoare. Specifications, Programs and Implementations, Technical Monograph PRG-29, Oxford University Computing Laboratory, Programming Research Group, Oxford, England (June 1982).

HOLZ82 G. J. Holzmann. A Theory for Protocol Validation, *IEEE Trans. on Computers* (August 1982), 730–738.

KANE83 P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence, *Proc. Principles of Distributed Computing* (August 1983), 228–240.

KELL76 R. M. Keller. Formal Verification of Parallel Programs, *Communications of the ACM* (July 1976), 371–384.

KOYM83 R. Koymans, J. Vytopil, and W. P. de Roever. Real-Time Programming and Asynchronous Message Passing, *Proc. Principles of Distributed Computing* (August 1983), 187–197.

LAMP78 L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System, *Communications of the ACM* (July 1978), 558–565.

LAMP79 L. Lamport. A New Approach to Proving the Correctness of Multiprocess Programs, *ACM Trans. on Programming Languages and Systems* (July 1979), 84–97.

LANS83 A. L. Lansky and S. Owicki. GEM: A Tool for Concurrency Specification and Verification, *Proc. Principles of Distributed Computing* (August 1983), 198–212.

LAUE79 P. E. Lauer, P. R. Torrigiani and M. W. Shields. COSY: A System Specification Language Based on Paths and Processes, *Acta Informatica* (1979), 451–503.

MILN82 R. Milner. Four Combinators for Concurrency, *Proc. Principles of Distributed Computing* (August 1982), 104–110.

MISR81 J. Misra and K. M. Chandy. Proofs of Networks of Processes, *IEEE Trans. on Software Engineering* (July 1981), 417–426.

OWIC80 S. Owicki and L. Lamport. Proving Properties of Concurrent Programs, Computer Systems Laboratory, Stanford University (October 1980).

PNUE79 A. Pnueli. The Temporal Semantics of Concurrent Programs, *Semantics of Concurrent Computation*, Kahn (Ed.), Springer-Verlag (1979), 1–20.

RAMA80 K. Ramamritham and R. M. Keller. Specification and Synthesis of Synchronizers, *Proc. 1980 International Conference on Parallel Processing* (August 1980).

RAMA81 K. Ramamritham and R. M. Keller. Specifying and Proving Properties of Sentinel Processes, *Proc. Fifth International Conference on Software Engineering* (1981), 374–382.

RIDD72 W. E. Riddle. Modelling and analysis of supervisory systems. Ph. D. Thesis and Tech. Rep. STAN-CS-72-271, Computer Science Department, Stanford University, Stanford, Ca. (March 1972).

RIDD76 W. E. Riddle. An approach to software system modelling, behavior specification and analysis, RSSM/25, Department of Computer and Communication Sciences, University of Michigan, Ann Arbor (July 1976).

RIDD78 W. Riddle, J. Wileden, J. Sayler, A. Segal and A. Stavely. Behavior Modelling During Software Design, *IEEE Trans. on Software Engineering* (July 1978), 283–292.

SCHW83 R. L. Schwartz, P. M. Melliar–Smith, F. H. Vogt. An Interval Logic for Higher-level Temporal Reasoning, *Proc. Principles of Distributed Computing* (August 1983), 173–186.

SHAW79 A. C. Shaw. Software specification languages based on regular expressions, Institut für Informatik, Eidgenössische Technische Hochschule, Zürich (June 1979).

STAV83 A. M. Stavely, J. M. Newman, G. B. Titus, and S. Orr. User Documentation, Software Design Technology Project, Computer Science Department, New Mexico Tech, Socorro (March 1983).

TAYL83 R. N. Taylor. A General-Purpose Algorithm for Analyzing Concurrent Programs, *Communications of the ACM* (May 1983), 362–376.

WILE78 J. C. Wileden. Modelling parallel systems with dynamic structure, COINS Tech. Rep. 78-4, University of Massachussetts, Amherst (January 1978).

WILE79 J. Wileden. DREAM – An Approach to Designing Large Scale, Concurrent Software Systems, *Proc. 1979 National Conference of the ACM* (October 1979), 88–94.

WILE80 J. Wileden. Techniques for Modelling Parallel Systems with Dynamic Structure, *Journal of Digital Systems* (Summer 1980), 177–197.

WONG84 Y. Wong. A DYMOL Simulation system, Department of Computer and Information Science, University of Massachusetts, Amherst (in preparation).

APPENDIX

FORMAL DESCRIPTION OF SDYMOL PROCESS EXPRESSION REDUCTION PROCEDURE

In this appendix we formally define the procedure, described informally in chapter VI, whereby a reduced process expression is obtained from a process expression that meets the process expression reduction criteria (i)–(iv) of figure VI.3. Definitions required for the description of the procedure are presented first. The procedure is then described and shown to be correct.

In this appendix, let q represent an arbitrary, but fixed, process in an SDYMOL system Σ. As usual, we refer to the components of Σ using the notation established in (III.2.1). The alphabet A_q represents the (derived) alphabet of the process q, as defined in (VI.2.1).

§1. Preliminary definitions

The process expression π_q for the process q must meet the process expression reduction criteria (i)–(iv) shown in figure VI.3 before it can be reduced using the procedure described below. As indicated by these criteria, the form of the process expression is important. In the discussion that follows, therefore, it is necessary to distinguish regular expressions of $\mathcal{RE}(A_q)$ not only on the basis of the languages they represent, but also on the basis of their syntactic form. Specifically, we need to distiguish regular expressions that differ in more than just the order in which the operands in series of concatenations and disjunctions are associated, or the order in which the operands of disjunctions appear. We do not, however, need to consider all regular expressions in $\mathcal{RE}(A_q)$. The following definitions permit us to identify the subset of $\mathcal{RE}(A_q)$ that we do need to consider and to distinguish between the regular expressions in this subset as required.

We determine if a regular expression belongs to the subset of $\mathcal{RE}(A_q)$ that concerns us in this appendix by reassociating the operands in series of concatenations and disjunctions wherever necessary to produce a regular expression in which the operands in series of concatenations and disjunctions are all associated from left to right. The original regular expression belongs to this subset if the regular expression produced in this manner is in disjunctive form. This motivates the following definition.

(1.1) *Definition.* If A is an alphabet of event symbols, $\alpha \in \mathcal{RE}(A)$, and α does not involve the shuffle operator, we define $[\![\alpha]\!] \in \mathcal{RE}(A)$ as follows.

(i) If α is one of $\emptyset$, λ, or a, where $a \in A$, then $[\![\alpha]\!]$ is just α.

(ii) If α has the form $\beta \vee \gamma$, where $\beta, \gamma \in \mathcal{RE}(A)$, then

  (a) if γ has the form $\delta \vee \epsilon$, where $\delta, \epsilon \in \mathcal{RE}(A)$, then $[\![\alpha]\!]$ is $[\![\beta \vee \delta]\!] \vee [\![\epsilon]\!]$,

  (b) otherwise, $[\![\alpha]\!]$ is $[\![\beta]\!] \vee [\![\gamma]\!]$.

(iii) If α has the form $\beta\gamma$, where $\beta, \gamma \in \mathcal{RE}(A)$, then

  (a) if γ has the form $\delta\epsilon$, where $\delta, \epsilon \in \mathcal{RE}(A)$, then $[\![\alpha]\!]$ is $[\![\beta\delta]\!][\![\epsilon]\!]$,

  (b) otherwise, $[\![\alpha]\!]$ is $[\![\beta]\!][\![\gamma]\!]$.

(iv) If α has the form β^*, where $\beta \in \mathcal{RE}(A)$, then $[\![\alpha]\!]$ is $[\![\beta]\!]^*$.

If $\alpha \in \mathcal{RE}(A)$, α does not involve the shuffle operator, and $[\![\alpha]\!]$ is in disjunctive form, we say α can be *syntactically expressed* in disjunctive form. The subset of $\mathcal{RE}(A)$ consisting of those regular expressions that can be syntactically expressed in disjunctive form is denoted by $\mathcal{DE}(A)$. For example,

$$\begin{aligned}
[\![a \vee (bc^* \vee d)]\!] &= [\![a \vee bc^*]\!] \vee [\![d]\!] \\
&= ([\![a]\!] \vee [\![bc^*]\!]) \vee d \\
&= (a \vee [\![b]\!][\![c^*]\!]) \vee d \\
&= (a \vee b[\![c]\!]^*) \vee d \\
&= (a \vee bc^*) \vee d \\
&= a \vee bc^* \vee d,
\end{aligned}$$

and so we have $a \vee (bc^* \vee d) \in \mathcal{DE}(A)$.[1] Clearly, not all regular expressions can be syntactically expressed in disjunctive form. The regular expression $d\Big((a \vee b)c\Big)$ does not belong to $\mathcal{DE}(A)$, since $[\![d\Big((a \vee b)c\Big)]\!] = d(a \vee b)c$, which is not in disjunctive form.

If $\alpha \in \mathcal{RE}(A)$ does not involve the shuffle operator, then the regular expressions α and $[\![\alpha]\!]$ differ only in the order in which the operands in series of concatenations and disjunctions are associated, and in $[\![\alpha]\!]$, the operands in series of concatenations and disjunctions are all associated from left to right. If α is already in disjunctive form, then α and $[\![\alpha]\!]$ are identical. The regular expressions of $\mathcal{RE}(A)$ that are in disjunctive form, therefore, all belong to $\mathcal{DE}(A)$.

The process expression reduction criterion (ii) requires that the process expression π_q for the process q be expressed in disjunctive form. We therefore have $\pi_q \in \mathcal{DE}(A_q)$. As we show below, each stage of the reduction procedure produces a regular expression over A_q that can be syntactically expressed in disjunctive form. In this appendix, therefore, we restrict our attention to the subset $\mathcal{DE}(A_q)$ of $\mathcal{RE}(A_q)$.

To distinguish between the elements of $\mathcal{DE}(A_q)$ as required, we define the following equivalence relation.

(1.2) *Definition.* If A is an alphabet of event symbols, we first define the *syntactic equivalence* relation, $=_s$, on the regular expressions of $\mathcal{DE}(A)$ that are in disjunctive form as follows.

(i) If $\bigvee_{1 \le j \le m} \alpha_{1j} \dots \alpha_{n_j j}$ and $\bigvee_{1 \le j \le m'} \alpha'_{1j} \dots \alpha'_{n'_j j}$ are in disjunctive form, then $\bigvee_{1 \le j \le m} \alpha_{1j} \dots \alpha_{n_j j} =_s \bigvee_{1 \le j \le m'} \alpha'_{1j} \dots \alpha'_{n'_j j}$ if and only if $m = m'$ and there is a one-to-one correspondence between the terms of $\bigvee_{1 \le j \le m} \alpha_{1j} \dots \alpha_{n_j j}$ and the terms of $\bigvee_{1 \le j \le m'} \alpha'_{1j} \dots \alpha'_{n'_j j}$ that matches up syntactically equivalent terms.

---

[1] As usual, unless otherwise explicitly specified by parentheses, disjunction and concatenation are assumed to associate from left to right

(ii) If $\alpha_1 \cdots \alpha_n$ and $\alpha'_1 \cdots \alpha'_{n'}$ are in disjunctive form, then $\alpha_1 \cdots \alpha_n =_e \alpha'_1 \cdots \alpha'_{n'}$ if and only if $n = n'$ and $\alpha_i =_e \alpha'_i$, for $1 \leq i \leq n$.

(iii) If α and α' are one of $\emptyset$, λ, or a, where $a \in A$, then $\alpha =_e \alpha'$ if and only if α and α' are identical.

We then extend this definition to all of $\mathcal{DE}(A)$ in the obvious fashion: if $\alpha, \alpha' \in \mathcal{DE}(A)$, we define $\alpha =_e \alpha'$ if and only if $[\![\alpha]\!] =_e [\![\alpha']\!]$.

Since it is not necessary, when reducing a process expression, to distinguish between regular expressions of $\mathcal{DE}(A_q)$ that are syntactically equivalent, we identify the elements of $\mathcal{DE}(A_q)$ with their syntactic equivalence classes. Thus, for example, $ab \vee acd =_e a(cd) \vee ab$, and so we do not distinguish between the regular expressions $ab \vee acd$ and $a(cd) \vee ab$, but $a^* \neq_e \lambda a^*$, and so these regular expressions represent distinct syntactic equivalence classes.

Clearly, a representative in disjunctive form of a particular equivalence class is unique up to the order of alternatives within disjunctions. For this reason, we often refer to the terms and components of a regular expression belonging to $\mathcal{DE}(A_q)$ as if the regular expression, itself, were in disjunctive form. When we do this we are actually referring to the terms and components of a representative in disjunctive form of the syntactic equivalence class of the regular expression.

Using the above definitions, the process expression reduction criterion (ii) can be relaxed slightly. To apply the process expression reduction procedure described below to a process expression π_q, it is not actually necessary that π_q be expressed in disjunctive form. It must, however, be syntactically equivalent to a regular expression in disjunctive form, i.e., it must belong to $\mathcal{DE}(A_q)$.

We now define predicates on $\mathcal{DE}(A_q)$ that allow us to state the process expression reduction criteria (iii) and (iv) more precisely. The first predicate, $\text{ISIN}(\alpha, B)$, is used in the statement of the second. It determines if any event symbols from the set, B, appear in the regular expression, α. The second predicate, $\text{LASTCOMP}(\alpha, B)$, determines if the event symbols in the set, B, appear

(only) as last components of terms in the regular expression, α. The third predicate, FOLLOWS(α, a, b), determines if the event symbol, a, is always immediately followed by the event symbol, b, in a regular expression, α.

(1.3) *Definition.* If A is an alphabet of event symbols, then for $B \subseteq A$ and $\alpha =_e \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}$, where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j} \in \mathcal{DE}(A)$ is in disjunctive form, we define the predicate ISIN(α, B) by

ISIN$(\alpha, B) \equiv$
there exists $1 \le j \le m$, $1 \le i \le n_j$, and $b \in B$,
such that $\alpha_{ij} =_e b$
or there exists $1 \le j \le m$, $1 \le i \le n_j$, $b \in B$ and $\beta \in \mathcal{RE}(A)$
such that $\alpha_{ij} =_e \beta^*$ and ISIN(β, B).

If $b \in A$, we write ISIN(α, b) for ISIN$(\alpha, \{b\})$.

(1.4) *Definition.* If A is an alphabet of event symbols, then for $B \subseteq A$ and $\alpha =_e \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}$, where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j} \in \mathcal{DE}(A)$ is in disjunctive form, we define the predicate LASTCOMP(α, B) by,

LASTCOMP$(\alpha, B) \equiv$
for all $1 \le j \le m$, $1 \le i \le n_j$, and $b \in B$
if ISIN(α_{ij}, b)
then $i = n_j$ and $\alpha_{ij} =_e b$.

As before, we write LASTCOMP(α, b) for LASTCOMP$(\alpha, \{b\})$.

(1.5) *Definition.* If A is an alphabet of event symbols, then for $a, b \in A$ and $\alpha =_e \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}$, where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j} \in \mathcal{DE}(A)$ is in disjunctive form,

we define a predicate FOLLOWS(α,a,b) by,

$$\begin{aligned}&\text{FOLLOWS}(\alpha,a,b) \equiv\\ &\quad \text{for all } 1 \le j \le m \text{ and } 1 \le i \le n_j\\ &\qquad \text{if } \alpha_{ij} =_e a\\ &\qquad \text{then } i < n_j \text{ and } \alpha_{(i+1)j} =_e b\\ &\qquad \text{else if there exists } \beta \in \mathcal{RE}(A) \text{ such that } \alpha_{ij} =_e \beta^*\\ &\qquad\quad \text{then FOLLOWS}(\beta,a,b).\end{aligned}$$

These predicates are well-defined (i.e., they do not produce different values when interpreted using different representatives in disjunctive form of the same syntactic equivalence class) because a representative in disjunctive form of a syntactic equivalence class is unique up to the order of the alternatives within disjunctions.

(1.6) *Remarks.* If a and b are distinct event symbols and α and β belong to $\mathcal{DE}(A)$, then the following statements are easy consequences of the above definitions.

(i) If ISIN(α,a) is false, then FOLLOWS(α,a,b).

(ii) If FOLLOWS(α,a,b) and ISIN(α,a), then ISIN(α,b).

(iii) If α_{ij} is a component of α and $\alpha_{ij} =_e \beta^*$, then FOLLOWS(α,a,b) implies FOLLOWS(β,a,b).

(iv) If α and β are single terms, then FOLLOWS(α,a,b) and FOLLOWS(β,a,b) implies FOLLOWS$(\alpha\beta,a,b)$.

(v) If $\alpha =_e \alpha_1 \ldots \alpha_n$, where $\alpha_1 \ldots \alpha_n$ is in disjunctive form, and if $1 \le k \le n$, then FOLLOWS(α,a,b) implies FOLLOWS$(\alpha_k \ldots \alpha_n,a,b)$.

(vi) If $\alpha =_e \alpha_1 \ldots \alpha_n$, where $\alpha_1 \ldots \alpha_n$ is in disjunctive form, and if $1 \le k \le n$ satisfies $\alpha_k \neq_e b$, then FOLLOWS(α,a,b) implies FOLLOWS$(\alpha_1 \ldots \alpha_{k-1},a,b)$.

Using the pedicates defined above, the process expression reduction criteria

are restated as follows.

(1.7)
(i) $\pi_q \in \mathcal{DE}(A_q)$,
(ii) LASTCOMP$(\pi_q, stop(q))$, and
(iii) FOLLOWS$(\pi_q, w(p), ne(q))$.

Notice that the original process expression reduction criterion (ii) (see figure VI.3) has been relaxed and combined with the original process expression reduction criterion (i), producing the criterion (i) above.

§2. The process expression reduction procedure

A process expression π_q satisfying (1.7) is reduced in three stages. These stages are described informally in chapter VI. We describe them more formally below.

Stage one of the reduction procedure

In the first stage of the reduction procedure, the process expression π_q, which is assumed to satisfy (i)–(iii) of (1.7), is transformed into a process expression π'_q, with the following properties:

(2.1)
(i) $\pi'_q \in \mathcal{DE}(A_q)$,
(ii) $\mathcal{P}(\pi'_q) \subseteq \mathcal{P}(\pi_q)$,
(iii) $\mathcal{P}(\pi_q)|_{\kappa_4(q)} \subseteq \mathcal{P}(\pi'_q)$, and
(iv) the first component containing an $ne(q)$ symbol in every term of π'_q is basic,

where $\kappa_4(q) = \lambda$ is the SDYMOL constraint that filters out prefixes of the system expression containing an $ne(q)$ symbol.

We use the predicate defined below to state (iv) above more concisely.

(2.2) *Definition.* If A is an alphabet of event symbols, then for $a \in A$ and $\alpha =_e \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}$, where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j} \in \mathcal{DE}(A)$ is in disjunctive form, we define the predicate FIRSTOUT(α, a) by,

FIRSTOUT$(\alpha, a) \equiv$
  for all $1 \le j \le m$
    if i is the least index satisfying
      $1 \le i \le n_j$ and ISIN(α_{ij}, a)
    then $\alpha_{ij} =_e a$.

The invariance of representatives in disjunctive form of a given syntactic equivalence class assures that FIRSTOUT(α, a) is well-defined. Using this predicate, property (iv) of (2.1) is restated as FIRSTOUT$(\pi'_q, ne(q))$.

As explained informally in chapter VI, the process expression π_q is transformed into a regular expression π'_q satisfying (i)–(iv) of (2.1) by pulling terms containing $ne(q)$ symbols out of the scope of certain of the star operators in π_q. The algorithm for doing this, which we call PULL, is defined in figure 1 using an arbitrary event symbol, a (instead of $ne(q)$). For an informal explanation of this algorithm, the reader is referred to chapter VI.

At least on the inputs for which it terminates, the algorithm PULL returns syntactically equivalent results when interpreted using different representatives of the same syntactic equivalence class. We write PULL(α, a) for the regular expression (i.e., the syntactic equivalence class of the regular expression) obtained by applying the algorithm PULL using the inputs α and a.

The algorithm PULL can be shown to terminate using an inductive argument, where the induction is performed on the number of star operators that contain the event symbol a within their scope. We therefore make the following definition.

(2.3) *Definition.* If A is an alphabet of event symbols, then, for each $a \in A$, we define a map d_a, which associates every regular expression from $\mathcal{DE}(A)$ with a

Algorithm PULL:

Inputs:

$a \in A$ and

$\alpha =_e \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}$,

where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j} \in \mathcal{DE}(A)$ is in disjunctive form

Algorithm:

(1) if FIRSTOUT($\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{n_j j}, a$) then return α

(2) else if $m > 1$ then return $\bigvee_{1 \le j \le m}$ PULL($\alpha_{1j} \ldots \alpha_{n_j j}, a$)

(3) else write $\alpha =_e \alpha_1 \ldots \alpha_n$,

where $\alpha_1 \ldots \alpha_n$ is in disjunctive form

(4) find the least index k such that ISIN(α_k, a)

(5) choose β in disjunctive form such that $\alpha_k =_e \beta^*$

(6) calculate PULL(β, a)

(7) for some $t \ge 0$, let θ_j, for $1 \le j \le t$, denote the terms of PULL(β, a) satisfying ISIN(θ_j, a)

for some $l \ge 0$, let δ_j, for $1 \le j \le l$, denote the remaining terms of PULL(β, a)

and define $\delta =_e \bigvee_{1 \le j \le l} \delta_j$

(8) return

$$\text{PULL}(\alpha_1 \ldots \alpha_{k-1} \delta^* \alpha_{k+1} \ldots \alpha_n, a) \vee \left(\bigvee_{1 \le j \le t} \alpha_1 \ldots \alpha_{k-1} \delta^* \theta_j \alpha_{k+1} \ldots \alpha_n \right)$$

Figure 1

Algorithm for pulling the first occurences of an event symbol out of the scope of all star operators

non-negative integer, as follows.

(i) $d_a(\alpha) = 0$, if α is one of $\emptyset$, λ, or b, where $b \in A$,

(ii) $d_a(\alpha \vee \beta) = d_a(\alpha) + d_a(\beta)$,

(iii) $d_a(\alpha\beta) = d_a(\alpha) + d_a(\beta)$,

(iv) $d_a(\alpha^*) = \begin{cases} 1 + d_a(\alpha), & \text{if ISIN}(\alpha, a); \\ 0, & \text{otherwise,} \end{cases}$

where $\alpha, \beta \in \mathcal{DE}(A)$.

Clearly, the map d_a, for a given $a \in A$, is well-defined (i.e., it is constant on a syntactic equivalence classes). We now show that the algorithm PULL terminates for all legal inputs.

(2.4) **Proposition.** *If A is an alphabet of event symbols, $\alpha \in \mathcal{DE}(A)$, and $a \in A$, then the algorithm* PULL *terminates when applied with α and a as inputs.*

Proof. Write $\alpha =_s \bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{nj}$, where $\bigvee_{1 \le j \le m} \alpha_{1j} \ldots \alpha_{nj} \in \mathcal{DE}(A)$ is in disjunctive form. We induct on $d_a(\alpha)$ to prove the proposition.

Basis: $d_a(\alpha) = 0$ implies FIRSTOUT(α, a) and so PULL terminates trivially when applied to α and a (in step (1) of the algorithm PULL).

Inductive step: We assume that, as long as $d_a(\beta) < h$, the algorithm PULL terminates when applied with β and a as inputs, and show that, if $d_a(\alpha) = h$, then PULL terminates when applied with α and a as inputs.

Case 1: $m = 1$. Write $\alpha =_s \alpha_1 \ldots \alpha_n$, where $\alpha_1 \ldots \alpha_n$ is in disjunctive form. Now if FIRSTOUT(α, a), then PULL terminates when applied to α and a trivially. On the other hand, if FIRSTOUT(α, a) is false, then there is a least index k satisfying ISIN(α_k, a), and there is a regular expression β in disjunctive form satisfying $\alpha_k =_s \beta^*$, as required for steps (4) and (5) of the algorithm. Now, ISIN(α_k, a) implies ISIN(β, a), and so $d_a(\beta^*) > d_a(\beta)$. We therefore have $h = \sum_{1 \le i \le n} d_a(\alpha_i, a) \ge d_a(\alpha_k, a) = d_a(\beta^*, a) > d_a(\beta, a)$, and so the calculation of PULL(β, a) in step (6) terminates by the inductive hypothesis. Integers $t, l \ge 0$

and terms θ_j, for $1 \leq j \leq t$, and δ_j, for $1 \leq j \leq l$, can thus be obtained from this calculation, and δ defined as in required in step (7). (A disjunction over an empty set is, by convention, the empty regular expression, so if $l = 0$, then $\delta =_e \emptyset$.)

Finally, the assumption that ISIN(δ, a) is false implies that $d_a(\delta) = 0$, and so

$$\begin{aligned} h &= \sum_{1 \leq i \leq n} d_a(\alpha_i, a) \\ &> \sum_{1 \leq i \leq k-1} d_a(\alpha_i, a) + \sum_{k+1 \leq i \leq n} d_a(\alpha_i, a) \\ &= \sum_{1 \leq i \leq k-1} d_a(\alpha_i, a) + d_a(\delta^*, a) + \sum_{k+1 \leq i \leq n} d_a(\alpha_i, a) \\ &= d_a(\alpha_1 \ldots \alpha_{k-1} \delta^* \alpha_{k+1} \ldots \alpha_n, a). \end{aligned}$$

Thus, the calculation of PULL$(\alpha_1 \ldots \alpha_{k-1} \delta^* \alpha_{k+1} \ldots \alpha_n, a)$ in step (8) terminates. In this case, therefore, the calculation of PULL(α, a) terminates.

Case 2: $m > 1$. Then for each $1 \leq j \leq m$, we have $h = d_a(\alpha, a) \geq d_a(\alpha_{1j} \ldots \alpha_{n_j j}, a)$, and so by case 1 the calculation of PULL$(\alpha_{1j} \ldots \alpha_{n_j j}, a)$ in step (2) of the algorithm terminates. In this case, therefore, the calculation of PULL(α, a) terminates also. ∎

The algorithm used in the first stage of the process expression reduction procedure consists of applying the algorithm PULL to the process expression π_q and the event symbol $ne(q)$. For future reference, this algorithm, which we call STAGE1, is formally defined in figure 2. To prove that this algorithm is correct we need to show that if π'_q is obtained by applying the algorithm STAGE1 to a process expression π_q satisfying (i)–(iii) of figure (1.7), then π'_q satisfies (i)–(iv) of (2.1). For this proof, we use the following proposition.

(2.5) **Proposition.** *Given an alphabet A of event symbols, a regular expression $\alpha \in \mathcal{DE}(A)$, and an event symbol $a \in A$, if $\alpha' =_e$ PULL(α, a), then*

(i) $\alpha' \in \mathcal{DE}(A)$,

(ii) $\mathcal{L}(\alpha') \subseteq \mathcal{L}(\alpha)$,

Algorithm STAGE1:

Input:

π_q, where q satisfies (i)–(iii) of figure (1.7)

Algorithm:

(1) return PULL(π_q, $ne(q)$)

Figure 2

Algorithm for the first stage of the
process expression reduction procedure

(iii) FIRSTOUT(α', a),

(iv) $\{u \in \mathcal{L}(\alpha) \mid \rho_a(u) = \lambda\} \subseteq \mathcal{L}(\alpha')$, and

(v) $\{u \in \mathcal{P}(\alpha) \mid \rho_a(u) = \lambda\} \subseteq \mathcal{P}(\alpha')$.

Proof. The proof is a routine induction argument on $d_a(\alpha)$. Result (iv) is needed in order to prove (v). ∎

Correctness of the algorithm STAGE1 follows trivially from the definition of the algorithm and the above proposition. For future reference, we state this as a proposition below.

(2.6) **Proposition.** *If* $\pi'_q =_e$ STAGE1(π_q), *where* π_q *satisfies (i)–(iii) of (1.7), then*

(i) $\pi'_q \in \mathcal{DE}(A_q)$,

(ii) $\mathcal{P}(\pi'_q) \subseteq \mathcal{P}(\pi_q)$,

(iii) $\mathcal{P}(\pi_q)|_{\pi_4(q)} \subseteq \mathcal{P}(\pi'_q)$, and

(iv) FIRSTOUT(π'_q, $ne(q)$).

The next proposition is required for later stages of the process expression reduction procedure.

(2.7) **Proposition.** *If* $\pi'_q =_e \text{STAGE1}(\pi_q)$, *where* π_q *satisfies (i)–(iii) of (1.7), then*

(i) $\text{LASTCOMP}(\pi'_q, stop(q))$ *and*

(ii) $\text{FOLLOWS}(\pi'_q, w(p), ne(q))$, *for* $p \in P(q)$.

Proof. A straightforward induction argument shows that if A is an alphabet of event symbols, $\alpha \in \mathcal{DE}(A)$, $a, b \in A$, and $\alpha' =_e \text{PULL}(\alpha, a)$, then $\text{LASTCOMP}(\alpha, b)$ implies $\text{LASTCOMP}(\alpha', b)$. (Induct on $d_a(\alpha)$.) The desired result (i) then follows trivially from the assumption $\text{LASTCOMP}(\pi_q, stop(q))$, which is assumption (ii) of (1.7), and the definition of the algorithm STAGE1. Similarly, the desired result (ii) is established by showing that if A is an alphabet of event symbols, $\alpha \in \mathcal{DE}(A)$, $a, b \in A$ are distinct event symbols, and $\alpha' =_e \text{PULL}(\alpha, b)$, then $\text{FOLLOWS}(\alpha, a, b)$ implies $\text{FOLLOWS}(\alpha', a, b)$. (Induct on $d_b(\alpha)$.) ∎

Stage two of the reduction procedure

In the second stage of the procedure for reducing a process expression π_q, which is assumed to satisfy properties (i)–(iii) of (1.7), the regular expression π'_q obtained in the first stage is transformed into a process expression π''_q satisfying the properties

$$\text{(2.8)} \qquad \begin{aligned} &\text{(i)} \quad \pi''_q \in \mathcal{DE}(A), \\ &\text{(ii)} \quad P(\pi_q)|_{\kappa_4(q)} = P(\pi''_q), \text{ and} \\ &\text{(iii)} \quad \text{LASTCOMP}(\pi''_q, S_3(q)), \end{aligned}$$

where $S_3(q) = \{ stop(q), w(p) \}_{p \in P(q)}$ is the alphabet of the constraint $\kappa_3(q) = stop(q) \vee \left(\bigvee_{p \in P(q)} w(q) \right)$, which filters out prefixes of the system expression that do not contain a single $stop(q)$ of $w(p)$ symbol, for $p \in P(q)$. This is accomplished by simply dropping the "tail" of each term of π'_q beginning with the first component containing an $ne(q)$ symbol. The algorithm, which we call STAGE2, is formally defined in figure 3. The required process expression π''_q is obtained by applying the

Algorithm STAGE2:

Input:

π'_q, where $\pi'_q =_e$ STAGE1(π_q) and π_q satisfies (i)–(iii) of (1.7)

Algorithm:

(1) write $\pi'_q =_e \bigvee_{1 \le j \le m'} \pi'_{1j} \ldots \pi'_{n'_j j}$,
where $\bigvee_{1 \le j \le m'} \pi'_{1j} \ldots \pi'_{n'_j j}$ is in disjunctive form

(2) for $1 \le j \le m'$, find the least index k'_j such that
$1 \le k'_j < n'_j$ and ISIN$(\pi'_{(k'_j+1)j}, ne(q))$
or $k'_j = n'_j$

(3) return $\bigvee_{1 \le j \le m'} \pi'_{1j} \ldots \pi'_{k'_j j}$

Figure 3

Algorithm for the second stage of the
process expression reduction procedure

algorithm STAGE1 to π_q, to produce the process expression π'_q, and then applying the algorithm STAGE2 to π'_q.

From the definition of the algorithm STAGE2 it is clear that if π''_q is obtained in this fashion, it can be syntactically expressed in disjunctive form, so that (i) of (2.8) holds. The following propositions establish (ii) and (iii) of (2.8).

(2.9) **Proposition.** *If* $\pi'_q =_e$ STAGE1(π_q) *and* $\pi''_q =_e$ STAGE2(π'_q), *where* π_q *satisfies (i)–(iii) of (1.7), then* $P(\pi_q)|_{\kappa_4(q)} = P(\pi''_q)$.

Proof. We first show $P(\pi''_q) \subseteq P(\pi_q)|_{\kappa_4(q)}$.

Using the notation established in steps (1) and (2) of the algorithm STAGE2

(see figure 3), we have

$$\begin{aligned} P(\pi''_q) &= P\left(\bigvee_{1\le j\le m'} \pi'_{1j}\dots\pi'_{k'_j j}\right) \\ &\subseteq P\left(\bigvee_{1\le j\le m'} \pi'_{1j}\dots\pi'_{n'_j j}\right) \\ &= P(\pi'_q) \\ &\subseteq P(\pi_q), \end{aligned}$$

where the last inclusion is just (ii) of (2.6). By definition of k'_j, furthermore, we know that $\text{ISIN}(\pi'_{ij}, ne(q))$ is false for $1 \le j \le m'$ and $1 \le i \le k'_j$. If $u \in P(\pi''_q)$ we therefore conclude that $\rho_{ne(q)}(u) = \lambda$, and so every prefix of π''_q satisfies the constraint $\kappa_4(q) = \lambda$, over the constraint alphabet $S_4(q) = \{ ne(q) \}$. We therefore have $P(\pi''_q) = P(\pi''_q)|_{\kappa_4(q)} \subseteq P(\pi_q)|_{\kappa_4(q)}$, as desired.

To establish the reverse inclusion, note that, for $1 \le j \le m'$, if $k'_j < n'_j$ then $\text{FIRSTOUT}(\pi'_q, ne(q))$, which was established in (2.6), implies that $\pi'_{(k'_j+1)j} =_e ne(q)$. Thus, for all $1 \le j \le m'$, every prefix of $\pi'_{1j}\dots\pi'_{n'_j j}$ that is not also a prefix of $\pi'_{1j}\dots\pi'_{k'_j j}$ contains an $ne(q)$ symbol, which implies $P\left(\pi'_{1j}\dots\pi'_{n'_j j}\right)\Big|_{\kappa_4(q)} \subseteq P\left(\pi'_{1j}\dots\pi'_{k'_j j}\right)$. We therefore have

$$\begin{aligned} P(\pi_q)|_{\kappa_4(q)} &\subseteq P(\pi'_q)|_{\kappa_4(q)} \\ &= \bigcup_{1\le j\le m'} P\left(\pi'_{1j}\dots\pi'_{n'_j j}\right)\Big|_{\kappa_4(q)} \\ &\subseteq \bigcup_{1\le j\le m'} P\left(\pi'_{1j}\dots\pi'_{k'_j j}\right) \\ &= P(\pi''_q), \end{aligned}$$

where the first inclusion follows from (iii) of (2.6). ∎

(2.10) **Proposition.** *If* $\pi''_q =_e \text{STAGE2}(\pi'_q)$, *where* $\pi'_q =_e \text{STAGE1}(\pi_q)$ *and* π_q *satisfies (i)–(iii) of (1.7), then* $\text{LASTCOMP}(\pi''_q, S_3(q))$.

Proof. Write $\pi''_q =_e \bigvee_{1\le j\le m''} \pi''_{1j}\dots\pi''_{n''_j j}$, where $\bigvee_{1\le j\le m''} \pi''_{1j}\dots\pi''_{n''_j j}$ is in disjunctive form. We must show that, for all $1 \le j \le m''$, $1 \le i \le n''_j$, and $p \in P(q)$,

(i) if $\text{ISIN}(\pi''_{ij}, stop(q))$, then $i = n''_j$ and $\pi''_{n''_j j} =_e stop(q)$, and

(ii) if $\text{ISIN}(\pi''_{ij}, w(p))$, then $i = n''_j$ and $\pi''_{n''_j j} =_e w(p)$.

First observe that if $\bigvee_{1\le j\le m'} \pi'_{1j}\dots\pi'_{n'_j j}$ and k'_j, for $1 \le j \le m'$, are defined as in steps (1) and (2) of the algorithm STAGE2, so that $\bigvee_{1\le j\le m'} \pi'_{1j}\dots\pi'_{k'_j j} =_e \pi''_q =_e \bigvee_{1\le j\le m''} \pi''_{1j}\dots\pi''_{n''_j j}$, then the invariance of syntactically equivalent regular expressions in disjunctive form implies that $m' = m''$ and that there is a one-to-one correspondence between the terms of these expressions that matches up syntactically equivalent terms. Without loss of generality, therefore, we may assume that $\pi'_{1j}\dots\pi'_{k'_j j} =_e \pi''_{1j}\dots\pi''_{n''_j j}$, for all $1 \le j \le m''$, and thus that $k'_j = n''_j$ and $\pi'_{ij} =_e \pi''_{ij}$, for all $1 \le j \le m''$ and $1 \le i \le n''_j$. The desired result (i) follows easily from this observation, the assumption that $\pi'_q =_e \bigvee_{1\le j\le m'} \pi'_{1j}\dots\pi'_{n'_j j}$, and the observation that $\text{LASTCOMP}(\pi'_q, stop(q))$, which was established in (2.7).

To establish (ii), assume $p \in P(q)$, $1 \le j \le m''$, $1 \le i \le n''_j$, and $\text{ISIN}(\pi''_{ij}, w(p))$, and observe that then $\pi''_{ij} \neq_e \beta^*$, for any regular expression $\beta \in \mathcal{RE}(A)$. This is because $\pi'_{ij} =_e \pi''_{ij} =_e \beta^*$ and $\text{FOLLOWS}(\pi'_q, w(p), ne(q))$, which was noted in (2.7), imply $\text{FOLLOWS}(\beta, w(p), ne(q))$ (by (iii) of (1.6)). $\text{ISIN}(\beta, w(p))$ and $\text{FOLLOWS}(\beta, w(p), ne(q))$, however, imply $\text{ISIN}(\beta, ne(q))$ (by (ii) of (1.6)). Thus, if $\pi''_{ij} =_e \beta^*$, we have $\text{ISIN}(\pi'_{ij}, ne(q))$, and since $1 \le i \le n''_j = k'_j$, this contradicts the choice of k'_j.

Because $\pi''_{ij} \neq_e \beta^*$, for any $\beta \in \mathcal{RE}(A_q)$, and $\text{ISIN}(\pi''_{ij}, w(p))$, we conclude that $\pi'_{ij} =_e \pi''_{ij} =_e w(p)$, and, as $\text{FOLLOWS}(\pi'_q, w(p), ne(q))$, that $i < n'_j$ and $\pi'_{(i+1)j} =_e ne(q)$. By the choice of k'_j, therefore, we must have $i = k'_j = n''_j$, as desired. ∎

The algorithm STAGE2 is therefore correct with respect to the criteria stated in (2.8).

Stage three of the reduction procedure

In the final stage of the process expression reduction procedure, the process expression π_q'' obtained in the second stage is transformed into the reduced process expression π_q'''. The reduced process expression satisfies

$$\text{(2.11)} \quad \begin{array}{ll} \text{(i)} & \pi_q''' \in \mathcal{DE}(A_q) \\ \text{(ii)} & \mathcal{P}(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = \mathcal{P}(\pi_q''')|_{\{\kappa_3(q),\kappa_4(q)\}}, \text{ and} \\ \text{(iii)} & \mathcal{P}(\pi_q''')|_{\kappa_3(q)} = \mathcal{L}(\pi_q'''), \end{array}$$

and is obtained from π_q'' by dropping all terms of π_q'' whose final component does not consist of a single $stop(q)$ or $w(p)$ symbol, for some $p \in P(q)$. The algorithm for this third stage, which we call STAGE3, is defined formally in figure 4.

Algorithm STAGE3:

Input:

π_q'', where $\pi_q'' =_e \text{STAGE2}(\pi_q')$, $\pi_q' =_e \text{STAGE1}(\pi_q)$, and
π_q satisfies (i)–(iii) of (1.7)

Algorithm:

(1) write $\pi'' =_e \bigvee_{1 \le j \le m''} \pi''_{1j} \ldots \pi''_{n''_j j}$,
where $\bigvee_{1 \le j \le m''} \pi''_{1j} \ldots \pi''_{n''_j j}$ is in disjunctive form

(2) define the index set $I = \left\{ j \;\middle|\; \text{ISIN}(\pi''_{n''_j j}, S_3(q)) \right\}$

(3) return $\bigvee_{j \in I} \pi''_{1j} \ldots \pi''_{n''_j j}$

Figure 4

Algorithm for the third stage of the
process expression reduction procedure

The algorithm for reducing a process expression π_q, satisfying (i)–(iii) of (1.7), therefore, consists of applying the algorithm STAGE1 to π_q, to produce a process expression π'_q, satisfying (i)–(iv) of (2.1), then applying the algorithm STAGE2 to the process expression π'_q, to produce a process expression π''_q, satisfying (i)–(iv) of (2.6), and finally, applying the algorithm STAGE3 to the process expression π''_q, to produce the reduced process expression π'''_q, satisfying (i)–(iii) of (2.11). This algorithm, which we call REDUCE, is summarized in figure 5. The next proposition shows that this algorithm satisfies the criteria of (VI.3.5)).

Algorithm REDUCE:

Input:

π_q, where π_q satisfies (i)–(iii) of (1.7)

Algorithm:

(1) define $\pi'_q =_s$ STAGE1(π_q)

(2) define $\pi''_q =_s$ STAGE2(π'_q),

(3) return $\pi'''_q =_s$ STAGE3(π''_q)

Figure 5

Algorithm for reducing an SDYMOL process expression

(2.12) **Proposition.** *If* $\pi'''_q =_s$ REDUCE(π_q), *where* π_q *satisfies (i)–(iii) of (1.7), then*

(i) $\pi'''_q \in \mathcal{DE}(A_q)$

(ii) $P(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = P(\pi'''_q)|_{\{\kappa_3(q),\kappa_4(q)\}}$, *and*

(iii) $P(\pi'''_q)|_{\kappa_3(q)} = \mathcal{L}(\pi'''_q)$.

Proof. Define π'_q and π''_q as in steps (1) and (2) of the algorithm REDUCE. Write

$\pi''_q =_e \bigvee_{1\le j\le m''} \pi''_{1j}\ldots\pi''_{n''_j j}$, where $\bigvee_{1\le j\le m''} \pi''_{1j}\ldots\pi''_{n''_j j}$ is in disjunctive form and the terms in this expression for π''_q have been arranged so that, for some $1 \le k'' \le m''$, $\text{ISIN}(\pi''_{n''_j j}, S_3(q))$ is true, for $1 \le j \le k''$, and false, for $k'' < j \le m''$. According to the definition of the algorithm STAGE3, we then have $\pi'''_q =_e \bigvee_{1\le j\le k''} \pi''_{1j}\ldots\pi''_{n''_j j}$. The desired property (i), therefore, is immediate.

Proof of (ii): Proposition (2.9) implies that $P(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = P(\pi''_q)|_{\kappa_3(q)}$. Now, since $\text{ISIN}(\pi''_{n''_j j}, S_3(q))$ is false, for $j > k''$, and $\text{LASTCOMP}(\pi''_q, S_3(q))$, we reason that $\text{ISIN}(\pi''_{1j}\cdots\pi''_{n''_j j}, S_3(q))$ is false, for $j > k''$. This implies that the prefixes of the jth term of π''_q, for $j > k''$ do not contain any symbols from the alphabet $S_3(q)$, and so do not satisfy the constraint $\kappa_3(q)$. Reasoning then that

$$\begin{aligned} P(\pi''_q)|_{\kappa_3(q)} &= \bigcup_{1\le j\le m''} P\left(\pi''_{1j}\cdots\pi''_{n''_j j}\right)\Big|_{\kappa_3(q)} \\ &= \bigcup_{1\le j\le k''} P\left(\pi''_{1j}\cdots\pi''_{n''_j j}\right)\Big|_{\kappa_3(q)} \\ &= P(\pi'''_q)|_{\kappa_3(q)}, \end{aligned}$$

we conclude that $P(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = P(\pi'''_q)|_{\kappa_3(q)}$. But every prefix of π''_q, and hence of π'''_q, satisfies the constraint $\kappa_4(q)$ (implied by proposition (2.9)), and so we have $P(\pi_q)|_{\{\kappa_3(q),\kappa_4(q)\}} = P(\pi'''_q)|_{\kappa_3(q)} = P(\pi'''_q)|_{\{\kappa_3(q),\kappa_4(q)\}}$, as desired.

Proof of (iii): Observe that $\text{LASTCOMP}(\pi''_q, S_3(q))$ and $\text{ISIN}(\pi''_{n''_j j}, S_3(q))$ imply that $P\left(\pi''_{1j}\cdots\pi''_{n''_j j}\right)\Big|_{\kappa_3(q)} = \mathcal{L}\left(\pi''_{1j}\cdots\pi''_{n''_j j}\right)$, for $j \le k''$. The desired result (iii) follows easily from this observation and the definition of π'''_q. ∎