Constrained Expressions: Toward Broad Applicability of Analysis Methods For Distributed Software Systems

L.K. Dillon
G.S. Avrunin
J.C. Wileden

COINS Technical Report 86–15
May 1986

Software Development Laboratory
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

G.S. Avrunin is with the Department of Mathematics and Statistics, University of Massachusetts, Amherst, MA 01003.

L.K. Dillon was with the Department of Computer and Information Science, University of Massachusetts, Amherst MA 01003. She is now with the Computer Science Department, University of California, Santa Barbara, CA 93106.

J.C. Wileden is with the Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

This work supported in part by the National Science Foundation under grant DCR-83-18776, by a Faculty Research Grant from the University of Massachusetts, and by the IBM Graduate Fellowship Program.

CONSTRAINED EXPRESSIONS: TOWARD BROAD APPLICABILITY OF ANALYSIS METHODS FOR DISTRIBUTED SOFTWARE SYSTEMS

Laura K. Dillon
Computer Science Department
University of California, Santa Barbara, CA 93106

George S. Avrunin
Department of Mathematics and Statistics
University of Massachusetts, Amherst, MA 01003

and

Jack C. Wileden
Department of Computer and Information Science
University of Massachusetts, Amherst, MA 01003

Categories and Subject Descriptors:

D.1.3 [Programming Techniques]: Concurrent Programming;
D.2.2 [Software Engineering]: Tools and Techniques;
D.2.4 [Software Engineering]: Program Verification;
D.3.2 [Programming Languages]: Design Languages;
F.3.1 [Logics and Meanings of Programs]: Specifying and Verifying and Reasoning About Programs

General Terms: Design, Languages, Theory, Verification

Additional Keywords and Phrases:

Constrained expressions, distributed software systems, software design tools, event-based, analysis of software designs, SDYMOL, CSP, Petri nets, Petri net languages.

This work supported in part by the National Science Foundation under grant DCR-83-18776, by a Faculty Research Grant from the University of Massachusetts and by the IBM Graduate Fellowship Program.

ABSTRACT

It is extremely difficult to characterize the possible behaviors of a distributed software system through informal reasoning. Developers of such systems would therefore benefit from tools that support formal reasoning about properties of a distributed system's behaviors. Ideally these tools should be applicable not only to completed programs but also to designs and other pre-implementation descriptions of a system as well. Furthermore, a desire for such tools should not limit a developer's choice of languages in which to describe the distributed software system during the various stages of its development.

Unfortunately, most previously existing approaches to formal analysis of distributed software system behavior are very limited in their applicability, each typically restricted to use with only a single language or in a single stage of the software development process. In this paper we present a basis for broadly applicable analysis methods for distributed software systems. The *constrained expression* formalism can be used with a wide variety of distributed system development notations to give a uniform, closed-form representation of a system's behavior. A collection of formal analysis techniques can then be applied to this uniform representation of a system's behavior to establish properties of the system. Examples of these formal analysis techniques and their use in conjunction with specific individual notations appear elsewhere. Here we illustrate the broad applicability of the constrained expression formalism by showing how constrained expression representations can be obtained from descriptions of systems in three quite different notations: SDYMOL, CSP, and Petri nets. Since features of these three notations span most of the significant alternatives for describing distributed systems, our examples offer persuasive evidence for the broad applicability of the constrained expression approach.

1. INTRODUCTION

The large number and complexity of the interactions that can take place among the asynchronous components of a distributed software system make it difficult to reason informally about the behavior of the system. Developers of distributed systems thus require powerful formal techniques for analyzing the systems they create. To be most effective, such techniques should be applicable throughout the software development process, but especially in the early, pre-implementation phases when errors are most easily corrected.

Developers of distributed systems also need congenial development notations, including design and implementation languages. It is obvious that different notations are suitable at different stages of software development, but features of the system under development may, for example, make one design language more appropriate than another. Software developers need to be able to choose the notation most appropriate for the immediate task.

Unfortunately, developers have generally been forced to make a choice between using powerful formal methods for analyzing their systems and making flexible use of appropriate development notations. Virtually all proposed analysis techniques have been based on a particular notation and depend in significant ways on special features of that notation. Indeed, many of these techniques force system developers to work in terms of sophisticated and abstract mathematical structures, rather than with standard design and implementation languages. Developers who wish to take advantage of the analysis techniques are thus required to work with a single development notation which may or may not be suitable for their immediate tasks.

One goal of our work on tools for distributed software system development has been *broad applicability*. Our aim is to produce tools that support formal analysis methods yet do not impose unnatural development notations on their users. Our approach to broad applicability has been to develop a formalism, called *constrained expressions*, that can be used with a wide variety of standard development notations and that is expressly designed to encode the information required for analysis of important properties of the behavior of distributed systems. With this formalism available, a developer is free to choose a notation appropriate for the type of system and its current state of development without sacrificing rigorous analysis. A description of the system in the chosen notation can then

be mechanically translated into a constrained expression representation of the behavior of the system, providing an appropriate formal structure on which to base analysis.

While the chosen development notation has presumably been designed to facilitate description and maximize expressive power, the constrained expression representation facilitates arguments concerning the order and number of occurrences of particular events. Arguments of just this type are required for analyzing a distributed system for such properties as mutual exclusion, deadlock, and starvation [3]. The constrained expression formalism thus offers distributed system developers a general intermediate form that both supports the formal analysis they require and allows them to work with whatever development notations they find appropriate. As a result, developers attain the benefits of formal rigor without the associated pain of unnatural concepts or notations.

In a previous paper [2], we showed how the constrained expression approach could be used to add analysis capabilities to a (single) particular distributed system design language. In the present paper, we illustrate the broad applicability of the approach by showing how constrained expression representations can be derived from descriptions of systems in three very different notations: SDYMOL, a distributed system design language based on buffered, asynchronous interprocess communication; CSP, a distributed system design and programming language based on synchronous, rendezvous-style interprocess communication; and Petri nets, a graphical formalism for describing concurrent systems based on a dataflow style of interprocess coordination. Since features of these three notations span most of the significant alternatives for describing distributed systems, our examples offer persuasive evidence for the broad applicability of the constrained expression approach. In section 2, we give a brief description of the constrained expression formalism. The use of the formalism with SDYMOL, CSP, and Petri nets is described in sections 3, 4, and 5, respectively, and illustrated by giving a constrained expression representation for a semaphore system described in each notation. In section 6, we discuss our experience in applying the constrained expression formalism with these and other notations and outline some of the directions of our current research in this area.

2. THE CONSTRAINED EXPRESSION FRAMEWORK

In this section we briefly describe the constrained expression representation of a

distributed system. Examples illustrating the constrained expression descriptive formalism and its use appear in the next several sections. A more formal presentation of the constrained expression framework is provided in an appendix.

A distributed system can, in general, produce a large number of widely different behaviors, although only one out of this large set of possible behaviors will actually be realized in any given execution, test run, or simulation of the system. The constrained expression framework is a general purpose formalism for describing the sets of possible behaviors of distributed systems. Given a description of a distributed system in some notation, such as a design or programming language, the corresponding constrained expression provides a concise, complete and well-defined representation of all the possible behaviors of that system. Although it might be the original or sole description of a system, a constrained expression representation is typically derived from a description of a distributed system in some other notation, as is illustrated in the examples in subsequent sections, and provides a closed-form representation of the potentially infinite set of possible behaviors of that system. In both of these respects, a constrained expression bears the same relationship to a distributed system that a regular expression bears to a finite state machine.

In the constrained expression framework, a behavior is viewed as a sequence of *event* occurrences. The set of things considered to be events varies depending upon the particular system and the level of detail at which it is being considered. For example, the events that are constituents of behaviors corresponding to a high level design would typically be much more abstract (e.g., 'respond to request for resource') than those that make up a behavior corresponding to a finished program (e.g., 'set *in_use* to TRUE'). In the constrained expression framework, however, events are always assumed to be indivisible and non-overlapping. This perspective does not limit the representational power of the formalism, since overlapping activities can be easily represented by treating their respective initiations and terminations as events. This view of events is essentially the same as that taken, for example, by Hoare [13].

In keeping with this event-based perspective, a constrained expression representation of a distributed system describes a set of event sequences, one corresponding to each possible behavior of the system. Associating a symbol with each type of event that can be a constituent of a system behavior (e.g., the symbol *resp_to_req* might be associated with

the 'respond to request for resource' event, while the symbol *set(in_use,TRUE)* might be associated with the 'set *in_use* to TRUE' event) generates an alphabet of event symbols. A constrained expression is then an expression over that alphabet representing a set of strings of event symbols. These strings are exactly the ones corresponding to those sequences of events that are possible behaviors of the system.

More precisely, a constrained expression is an expression over a set of symbols called the *augmented alphabet*, which contains two kinds of symbols. Most of the symbols in the augmented alphabet represent events that describe interesting aspects of some distributed system's possible behaviors. These typically include symbols such as the *resp_to_req* or *set(in_use,TRUE)* symbols mentioned in the examples above. The remaining symbols in the augmented alphabet are needed to capture important facets of the particular notation being used to describe the distributed system. These generally represent events such as the initiation or termination of some interaction among the system's components. Although these events may not be visible parts of a system's possible behaviors, they can be crucially important in determining which behaviors are actually possible. The use of such symbols is illustrated in the examples below.

A constrained expression consists of two parts, called a *system expression* and a *constraint set*. The system expression is a regular expression over the augmented alphabet. It is formed using the standard regular expression operators, plus the interleave or shuffle operator, denoted by $\triangle$. The interleave of two event strings represents their concurrent occurrence. Thus, the regular expression $ab \triangle cd$, which denotes the set of event strings $abcd, acbd, acdb, cabd, cadb, cdab$, represents the concurrent occurrence of the event sequences ab and cd.[1] In a constrained expression, the system expression may be viewed as the generator of "candidate" event sequences. That is, each prefix of a string in the set described by the system expression is considered a candidate to be a possible behavior of the distributed system. (We consider prefixes, rather than complete strings in the language of the system expression, in order to represent behaviors in which system components terminate prematurely.) This set of candidates contains all the possible behaviors, but in general it also includes event strings that do not in fact correspond to possible system behaviors. To determine which candidates actually represent possible system behaviors,

[1] The interleave operator preserves regularity [9].

we consider the set of prefixes in conjunction with the constraint set.

The constraint set, which is the second part of a constrained expression, consists of a collection of *constraints.* Each constraint is an expression over the augmented alphabet, formed by using the regular operators (including interleave) plus one additional operator, denoted by †. This unary operator represents the interleave of zero or more copies of its argument, and hence can be used to describe some number of concurrent occurrences of an event sequence. (Note that expressions containing this operator may not be regular.) Each constraint represents a pattern of acceptable event sequences with respect to some subset of the augmented alphabet, which is called that constraint's associated *constraint alphabet.* Every candidate event sequence in the set of prefixes arising from a constrained expression's system expression is "filtered" through the constrained expression's constraint set to determine whether it actually represents a possible behavior. This filtering process involves the following steps, which are repeated for each constraint in the constraint set. First, all event symbols in the prefix that are not in the constraint alphabet of the current constraint are erased from the prefix. Then, if the resulting string of event symbols conforms to the pattern represented by the constraint, the prefix is said to *satisfy* that constraint. Otherwise, the prefix contains an unacceptable pattern of event symbols from one of the constraint alphabets, and it is eliminated from the set of candidates. Exactly those prefixes satisfying all the constraints in the constraint set constitute the set of possible behaviors for the distributed system described by the constrained expression.

In principle, the set of all possible behaviors represented by a constrained expression is generated by exhaustively carrying out the process outlined above. That is, candidate event sequences are generated by taking prefixes from the set of strings described by the system expression, and then those candidates are filtered through the constraint set. The resulting set of prefixes, i.e., those that satisfy all the constraints in the constraint set, are called the *constrained prefixes.* As a final step, those symbols in the second part of the augmented alphabet, namely those used in capturing semantic details that may not be of interest as visible events, are erased from all the strings in the set of constrained prefixes. This yields a set called the *interpreted language* of the constrained expression, which is precisely the prefixes of the system expression that satisfy all the constraints and have been reduced to only the symbols representing significant, visible events in the distributed

system's behaviors. The interpreted language, then, represents exactly the set of possible behaviors of the distributed system. It is important to note that it is never necessary to actually generate this (possibly infinite) language. Instead, analysis techniques operate on the constrained expression itself, rather than on individual strings in the interpreted language.

Although the approach of describing a set of candidate event sequences and then eliminating those that fail to satisfy one or more constraints may seem more complicated than just directly generating the set of only the possible behaviors, we have found it to be both simpler and more natural. In particular, we have found it convenient to use the two parts of a constrained expression for two rather different purposes. We typically use the system expression part of a constrained expression primarily to describe behavioral properties of a particular system. For example, the event sequences described by a system expression might express the fact that some component of the distributed system first tries to receive three messages from certain places, then selects between two possible recipients and sends a message to one of them. We use constraints primarily to express fixed semantic properties for a class of distributed systems. For instance, in a constrained expression for a CSP system the constraints, among other things, enforce the synchronous nature of interprocess communication, while in a Petri net constrained expression the constraints primarily enforce the Petri net firing rules. Of course, there is nothing in the constrained expression framework that forces this separation of concerns, and we do not always adhere strictly to it ourselves. In fact, the two parts of the constrained expression framework provide a very flexible approach to representing the possible behaviors of distributed systems, and decisions regarding how to use the two parts in capturing the semantics of a particular class of distributed systems are left to the discretion of the framework's users. Alternative ways of dividing those semantics among the system expression and the constraint set are certainly possible and constitute an interesting area for further research.

Although occasionally created as the original or sole description of a distributed system, constrained expressions are most frequently generated by deriving them from a description in some other notation. As demonstrated in the remainder of this paper, it is possible to derive a constrained expression representation of the possible behaviors of a distributed system from descriptions in a wide variety of notations, including many

design and programming languages. For each such notation, constrained expressions can be derived by using a set of *translation rules*, which direct the transformation of a design or program into a system expression, along with a set of *constraint templates*, which provide generic versions of the constraints needed to capture the particular notation's semantic details. Developing the translation rules and constraint templates for a given notation, such as CSP or Petri nets, requires considerable effort and insight, but need be carried out only once for each notation. Once an appropriate set of translation rules and constraint templates has been created, carrying out the derivation process for a particular system's description in that notation is an entirely mechanical procedure of applying the translation rules and particularizing the constraint templates. One reason for adopting our approach to using the system expression and constraint set of the constrained expression framework is its effect on the constrained expression derivation process. We have found that our usage of system expressions and constraint sets simplifies the development of the translation rules and constraint templates, and also makes their correctness relatively apparent.

2.1 Related Work

Regular expression-like closed form descriptions of the sets of sequences of events representing system behaviors are also found in the work of Campbell and Habermann [5] (path expressions), Riddle [18] (message transfer expressions and event expressions), Welter [23] (counter expressions), and Shaw [19] (flow expressions). Message transfer expressions, event expressions, flow expressions, counter expressions and the COSY notation [14], which is based on path expressions, all rely on a filtering procedure, whereby sequences representing impossible behaviors are eliminated from a set containing all legal system behaviors and represented by a single, or possibly several, expressions.

The major difference between the constrained expression formalism and its ancestors (message transfer expressions, event expressions, and counter expressions) is the use of constraints to explicitly specify the conditions under which strings are to be eliminated from the set of possible behaviors described by an expression. The filtering procedures used with the earlier notations are equivalent, in the more general constrained expression formalism, to using a fixed, pre-defined set of constraints expressing synchronization requirements between communicating processes.

Like these earlier expression notations, flow expressions use pre-defined mechanisms for expressing constraints. Flow expressions, however, are also distinguished by the use of an additional operator, an infinite repetition operator, to permit the representation of infinite behaviors and by a different interpretation of events in the resulting expressions. These differences result in differences in descriptive power, ease of use for both description and analysis, and range of application [19,20].

The COSY notation closely resembles constrained expressions (although the dramatically different syntax of these notations obscures their underlying similarity). Differences in the two notations are primarily the result of a difference in their intended use. COSY was intended to be used directly by software developers for specifying concurrent software systems, not indirectly through a translation process like constrained expressions. The events in a COSY description, therefore, correspond to operations or procedures, rather than arbitrary system events. Path expressions, the COSY analogue of constraints, restrict the order in which operations can be invoked. A COSY description thus provides a non-algorithmic specification of the intended system behaviors, presumably for use in verification of a design or implementation. A constrained expression, on the other hand, is not taken as defining the intended observable behaviors of a system, but is used for exploring the properties of a particular design. Moreover, of course, the constrained expression formalism permits the use of a wide variety of notations, while COSY, itself intended for expressing specifications, admits no such generality.

A number of other event-based formalisms have been suggested for the description of distributed software systems. The trace models of Hoare [13] and Misra and Chandy [15] treat system behaviors as sets of sequences of communication events. Axiomatic proof techniques are provided for verifying properties of the systems. In contrast, the constrained expression approach uses a more general concept of event and provides algebraic analysis methods for establishing properties, such as absence of deadlock and limited use of shared resources, that can be interpreted as questions regarding the order and number of certain event occurrences in behaviors.

Greif [10] and Chen and Yeh [6] have developed event-based models that rely on explicit partial orderings of events. A system's behavior is represented by a set of events, along with certain partial order relations. A relation expresses a time order or enabling

relationship between events. Events which are not comparable in the partial orders are considered concurrent. This model is entirely compatible with the representation of system behaviors as sets of sequences of events used in the constrained expression formalism. If the relative order of two events is not determined by the nature of the system, then, for each of the possible orders, there will be system behaviors in which the events occur in that order. By considering the set of all possible system behaviors, the appropriate partial order can be recovered, even though events in a single behavior occur serially.

3. CONSTRAINED EXPRESSIONS FOR SDYMOL

To illustrate the generality of the constrained expression formalism, we describe the generation of constrained expression representations for systems expressed in three different distributed system development notations: SDYMOL, CSP and Petri nets. In both SDYMOL and CSP, a distributed system is modeled as a collection of sequential processes which communicate by passing messages. SDYMOL uses asynchronous message transmission, while CSP uses a synchronous interchange as its interprocess communication mechanism.

We illustrate each of the above notations by using them to describe Dijkstra's solution to the mutual exclusion problem. We then show how constrained expression representations for the systems are generated in each case. The generation of a constrained expression representing the behavior of the SDYMOL system is described in full detail in this section. Descriptions of the procedures used with CSP and Petri nets are then outlined in sections 4 and 5.

3.1. An SDYMOL system

SDYMOL is a simplified version of the Dynamic Modelling Language (DYMOL), which was designed to be used with the Dynamic Process Modelling Scheme (DPMS) [24]. 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. At the same time, it facilitates the representation of the potential interactions between the processes in a concurrent system.

A concurrent system in SDYMOL is composed of individual sequential processes,

communicating with each other by means of message transmission. 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.

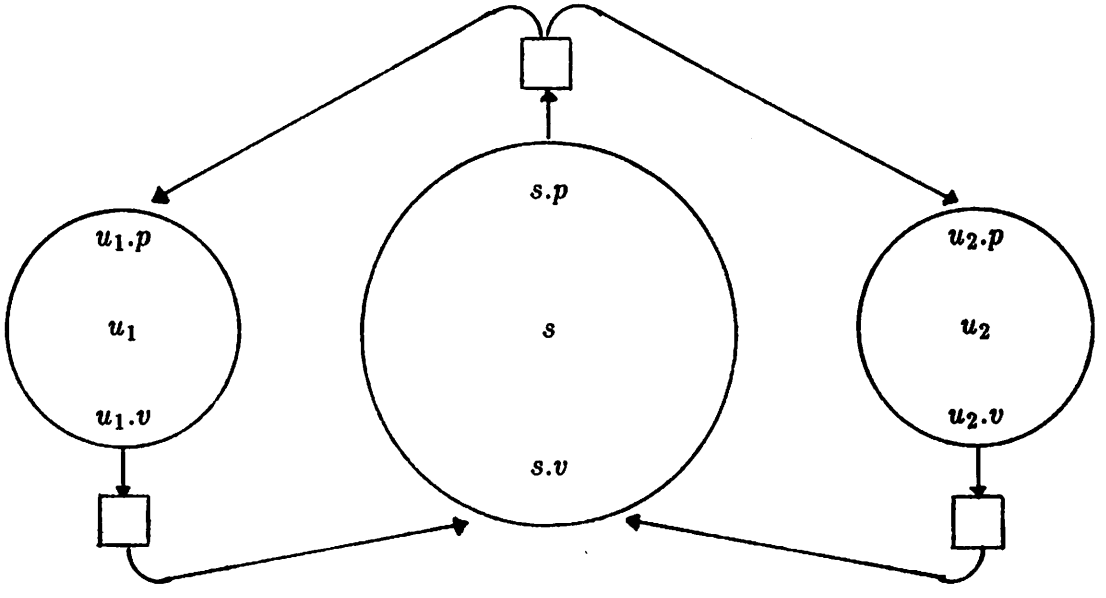
Message transmission as modeled in SDYMOL is both a communication and a synchronization mechanism. A process may send a message through an outbound *port* into a *link* associated with that port. The link is essentially an unbounded, unordered repository that is used to mediate the asynchronous message 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 that 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** ...). Internal process computations are abstractly represented in SDYMOL by primitive statements that do not contain any SDYMOL keywords, but consist

of single identifiers. Such statements are semantically null, serving simply as placeholders for activities that may be elaborated in subsequent system descriptions.

An SDYMOL solution to the mutual exclusion problem is shown in figure 1. This figure defines a system consisting of three processes, represented by the circles labeled s, u_1 and u_2. Boxes represent links, each of which is associated with an outbound port. Arrows connecting links and inbound ports represent interprocess communication channels. SDYMOL programs for the processes in the system are given in the figure.



u_i:

```
while internal test do
    begin
        receive u_i.p;
        use-resource_i;
        send u_i.v
    end.
```

s:

```
set buffer := ok;
do forever
    begin
        send s.p;
        receive s.v
    end.
```

Figure 1

SDYMOL solution to the mutual exclusion problem

The "user processes", u_1 and u_2, model asynchronous processes that periodically

require access to some shared resource. The "semaphore process", s, models a binary semaphore, which coordinates the activities of the user processes to assure that the resource is used in a mutually exclusive fashion.

The availability of the resource is represented by an *ok* message residing in the semaphore process' $s.p$ link[1]. This link is connected to the $u_i.p$ ports of the user processes. By attempting to receive a message through these ports, the user processes can determine if the resource is available. Thus, the **receive** $u_i.p$ statement in the body of a user process models Dijkstra's P operation. Note that if both user processes check for the resource simultaneously (modeling the situation where both processes attempt a P operation 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. (Hence one process is actually granted access to the resource, while the other one waits.) The $u_i.v$ links of the user processes are connected to the semaphore process' $s.v$ port. When a user process is finished with the resource, therefore, it signals this fact by depositing an *ok* message in the appropriate link. The **receive** $s.v$ statement in the body of the semaphore process thus models Dijkstra's V operation.

The **internal test** predicate in the body of a user process models some internal computation performed by the process to determine whether to repeat another cycle of computation requiring access to the resource. This cycle is represented by the three statements within the scope of the **while** statement. The second of these statements, the *use-resource*$_i$ statement, is an example of an identifier statement. It models an internal activity (the use of the resource) performed by the user process. Each user process is thus modeled as repeatedly executing a P operation, using the resource, and then signaling its availability with a V operation.

The semaphore process begins by initializing its buffer. It then cycles forever, depositing an *ok* message in the link connected to its $s.p$ port, which enables a P operation, and then waiting to receive a message through its $s.v$ port, which indicates that the re-

[1] Because the structure of the system in this example is static (i.e., processes are neither created nor destroyed during execution and the interprocess communication channels are all fixed), there is no need to distinguish between an outbound port and its link. Here and in the remainder of this paper, therefore, we identify outbound ports with their associated links.

source is once again available for use.

3.2. The SDYMOL constrained expression

In this section we show how to derive a constrained expression representation for the SDYMOL system shown in figure 1. We first describe the derivation of the augmented and terminal alphabets, then the system expression, and finally the constraints and constraint alphabets.

Recall that the augmented alphabet contains symbols representing events in the system as well as symbols needed to express certain aspects of the semantics of a distributed system development notation. The symbols required for this SDYMOL system are shown in figure 2. Conceptually, it is helpful to associate these symbols with events that can occur in the behaviors of the system, as indicated in this figure. Strictly speaking, however, some of 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), the modification and use of buffer contents, the transmission and receipt of messages, and the use of the resource. 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 *starve* 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 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.

The decision as to what symbols represent events of interest and should be retained in the interpreted language of a constrained expression representation for an SDYMOL system depends on the particular question that analysis of the representation is intended to address. Ideally, the terminal alphabet contains the symbols required to analyze some aspect of the modeled system's behavior, and as few other symbols as possible. Suppose, for example, we wish to determine if either of the user processes in the SDYMOL design

Symbol	*Associated Event*
$ch(l,p)$	Link l and port p are connected by a channel.
$def(q,m)$	The buffer of process q is modified so that it contains a message of type m ($m = @$ represents an undefined buffer).
$use(q,m)$	The buffer of process q is presumed to contain a message of type m ($m = @$ represents an undefined buffer).
$send(l,m)$	A message of type m is sent to link l ($m \neq @$).
$rec(l,p,m)$	A message of type m is transmitted from link l through port p ($m \neq @$).
ur_i	The process u_i uses the resource ($i = 1,2$).
$stop(q)$	Process q terminates normally.
$starve(p)$	The process containing inbound port p starves waiting for a message on a channel connected to p.
$ne(q)$	A non-event involving process q (marks positions in the process expression for q that cannot be reached in legitimate system behaviors).

q ranges over all processes, l ranges over all links, p ranges over all inbound ports, and m ranges over all message types and over the reserved symbol @, which represents an undefined message type.

Figure 2

SDYMOL event symbols

in figure 1 can starve, and if the desired mutually exclusive utilization of the resource 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 the system that contain a $starve(u_1.p)$ or $starve(u_2.p)$ symbol, and the latter as asking if there are any legal behavioral traces of the system that contain a ur_i symbol followed by a ur_j symbol, where $1 \leq i,j \leq 2$, with no intervening $rec(u_i.v, s.v, ok)$ symbol. In this case, therefore, we might choose the set

$$\{\, starve(u_i.p),\ ur_i,\ rec(u_i.v, s.v, ok) \,\}_{i=1,2}$$

for the terminal alphabet.

3.3. The SDYMOL system expression

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

$$\epsilon = \iota\left(\underset{q}{\triangle}\, \pi_q\right).$$

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

The initial expression is simply the concatenation of certain symbols from the augmented alphabet. It begins with a sequence of $ch(l,p)$ symbols representing the interprocess communication pathways in the system. 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 $def(q,@)$ symbols, one for each process, indicating that the contents of the processes' buffers are initially undefined. Finally, a $send(l,m)$ symbol is included for each message of type m that initially resides in a link l. The system expression derived from an SDYMOL system thus begins with a series of constraint alphabet symbols which encode the initial status of links, buffers and interprocess communication pathways, as will be illustrated in figure 4.

The process expressions, whose interleave constitutes the remainder of the SDYMOL system expression, are obtained from the SDYMOL code for the processes in the system through the statement-by-statement application of the set of translation rules given in

figure 3. Each rule indicates that an SDYMOL statement of the given form is to be transformed into the symbol sequence that appears on the right.

The application of the rules pertaining to the SDYMOL basic statements, the rules $T_1 - T_4$ of figure 3, 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*. Thus, the identifier ur_i is associated with the statement *use-resource*$_i$ in the example.) 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 $def(q,m)^*$ by the application of T_3 and T_6.

When interpreted using the associations established in figure 2, most of these rules are easily understood. The sequence of events produced by a process 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 on the right in rule T_1. Otherwise the buffer is undefined, as indicated by the final alternative in the translation column of this rule. As we show below, constraints are generated 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. The $starve(p)\, ne(q)$ 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 $starve(p)$ symbol appears in a constrained prefix of

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

q denotes the process defined by the SDYMOL program, curly brackets ({ }) signify the recursive application of these rules, l ranges over all links, and m and n range over all defined message types and over the undefined message type @.

Figure 3

SDYMOL translation rules

the constrained expression derived from an SDYMOL design of a system, where p is an inbound port of a process q, no symbols associated with q appear in the prefix anywhere to the right of this $starve(p)$ symbol. We do this by placing the non-event symbol, $ne(q)$, immediately after the $starve(p)$ symbol in T_2 and then generating a constraint (described below) to filter out prefixes containing the non-event symbol for q.

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 on the right 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 (described below) that eliminates prefixes containing $ne(q)$ symbols filters out these prefixes. The $ne(q)$ symbol is thus required in T_5 to accurately express the semantics of the SDYMOL **do forever** statement.

The system expression derived from the SDYMOL solution to the mutual exclusion problem presented in figure 1 is shown in figure 4. Clearly, many prefixes of this system expression do not represent possible behaviors of the system. Consider, for example, the prefix

$$\iota\ rec(u_1.v, u_1.p, ok)\ def(u_1, ok)\ starve(u_2.p)\ ne(u_2)\ ur_2,$$

where ι is the initial expression of figure 4. As there is no $send(u_1.v, ok)$ symbol preceding the $rec(u_1.v, u_2.p, ok)$ symbol, this string represents an event sequence in which an ok message is received from the link $u_1.v$ before any messages of that type have been deposited in the link. Even if an ok message were available in the link $u_1.v$, however, an event sequence representing a possible behavior of this system could not contain a

Initial expression:

$$\iota = ch(s.p, u_1.p)\ ch(s.p, u_2.p)\ ch(u_1.v, s.v)\ ch(u_2.v, s.v)$$
$$def(s, @)\ def(u_1, @)\ def(u_2, @)$$

Process expressions:

$$\pi_s =$$
$$def(s, ok)\Big[\ \Big(use(s, ok)\ send(s.p, ok)\ \vee\ use(s, @)\Big)$$
$$\Big(rec(s.p, s.v, ok)\ def(s, ok)\ \vee\ rec(u_1.v, s.v, ok)\ def(s, ok)$$
$$\vee\ rec(u_2.v, s.v, ok)\ def(s, ok)\ \vee\ starve(s.v)\ ne(s)\Big)\Big]^*$$
$$ne(s)\ stop(s)$$

$$\pi_{u_i} =$$
$$\Big[\Big(rec(s.p, u_i.p, ok)\ def(u_i, ok)\ \vee\ rec(u_1.v, u_i.p, ok)\ def(u_i, ok)$$
$$\vee\ rec(u_2.v, u_i.p, ok)\ def(u_i, ok)\ \vee\ starve(u_i.p)\ ne(u_i)\Big)$$
$$ur_i$$
$$\Big(use(u_i, ok)\ send(u_i.v, ok)\ \vee\ use(u_i, @)\Big)\Big]^*\ stop(u_i)$$

System expression:

$$\epsilon = \iota(\pi_s\ \triangle\ \pi_{u_1}\ \triangle\ \pi_{u_2})$$

Figure 4

System expression derived from the SDYMOL solution to the mutual exclusion problem shown in figure 1

$rec(u_1.v, u_1.p, ok)$ symbol, as the link $u_1.v$ and port $u_1.p$ are not connected by a channel, and so, according to the SDYMOL design notation semantics, messages cannot be transmitted from $u_1.v$ through $u_1.p$. As we show below, any of a number of constraints in the derived constrained expression representation of the system would filter out this prefix.

3.4. The SDYMOL constraints

Constraints are required in the constrained expression derived from an SDYMOL design because some of the SDYMOL language semantics are not expressed by the translation rules of figure 3. Each constraint describes the patterns of event symbols from the associated constraint alphabet that can appear in strings representing legal SDYMOL system behaviors. The constraints are used to restrict the order and number of certain symbols in prefixes of the system expression that are retained as constrained prefixes, so that 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 the system. In any legal SDYMOL system behavior, messages can only be received through (inbound) port p from link l if there is a communication channel connecting l and p. Hence, for each link l and port p a constraint is needed to assure that symbols representing the transmission of messages from l through 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.[1] The constraint alphabet $\{ ch(l,p),\ rec(l,p,m) \}_m$, where m ranges over all defined message types, is used in stating this behavioral requirement. The corresponding constraint is

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

Using the associations of figure 2, this constraint is easily understood. If there is a channel

[1] Constraints to assure this, of course, would be unnecessary if the translation rule T_2 were modified so that the alternation was performed over all ports connected to a link, rather than all ports in the system. The translation rule is more general, however, in the form we have given it. This form for T_2 is required when interprocess communication channels can be changed dynamically during execution, as in a general DYMOL system. For such systems a constraint is required to assure that communication takes place while appropriate communication channels are in existence. This, in fact, is the approach taken by Wileden [25] in his original constrained expression formulation of DYMOL.

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 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 $rec(l,p,m)$ 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 an SDYMOL system. As the initial expression of figure 4 does not contain a $ch(u_1.v,u_1.p)$ symbol, for example, the constraint $\kappa_1(u_1.v,u_1.p)$ obtained from the SDYMOL design in figure 1 filters out the prefix shown above, which represents an event sequence in which a message is transmitted from $u_1.v$ through $u_1.p$. The constrained expression derived from the SDYMOL design of a system contains a constraint $\kappa_1(l,p)$ for each link l and port p in the system.

A second type of constraint on SDYMOL system behavior pertains to the use and modification of buffer contents during the operation of a given process. In order to accurately represent the dependency of message transmission and certain iterative and conditional statement executions on the contents of the buffer of a process q, we use the constraint alphabet $\{\, def(q,m),\ use(q,m) \,\}_m$, where m ranges over all defined message types and over the undefined message @. The undefined message allows for representing an empty buffer. The constraint corresponding to this alphabet is

$$\kappa_2(q) = \left(\bigvee_m def(q,m)use(q,m)^* \right)^*.$$

As above, interpretation of this constraint is facilitated by the associations of figure 2, in which the symbol $def(q,m)$ is identified with the placing of a message of type m into the buffer of process q and the symbol $use(q,m)$ 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 presuming 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 q in an SDYMOL system.

The third type of constraint relates to system termination. We would like to assure that all behaviors described by the constrained expression representation of an SDYMOL system 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. We use the constraint alphabet $\{ stop(q), starve(p) \}_p$, where p ranges over the set of inbound ports of the process q, for formulating this restriction on the behavior of a process q. The corresponding constraint,

$$\kappa_3(q) = \left(\bigvee_p starve(p) \right) \vee stop(q),$$

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 constrained expression therefore contains a constraint $\kappa_3(q)$ for every process q in the system.

The fourth type of constraint 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 process q in an SDYMOL system we use the constraint alphabet, $\{ ne(q) \}$, and the constraint,

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

to filter out strings containing the non-event symbol associated with q. The constraint $\kappa_4(u_2)$ obtained from the SDYMOL design of figure 1, for example, assures that the prefix presented above is not retained in the set of constrained prefixes.

The fifth type of constraint relates to the transmission of messages in the system. 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 $\{ send(l,m),\ rec(l,p,m) \}_p$, where p ranges over all ports in the system, in stating this requirement. The corresponding constraint is

$$\kappa_5(l,m) = send(l,m)^* \triangle \left(send(l,m)\left(\bigvee_p rec(l,p,m)\right)\right)^\dagger .$$

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 and b's such that any prefix contains at least as many a's as b's." Thus, as each $send(l,m)$ symbol is associated with the placing of a message of type m into link l and each $rec(l,p,m)$ 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 $send(l,m)^*$). 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 $m \neq @$ in the system.

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 $m \neq @$, we define the constraint alphabet $\{ ch(l,p),\ send(l,m),\ rec(l,p',m),\ starve(p) \}_{p'}$, where p' ranges

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

$$\kappa_2(q) = \Big(\bigvee_{m'} def(q,m')use(q,m')^+\Big)^*$$

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

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

$$\kappa_5(l,m) = send(l,m)^* \triangle \left(send(l,m)\Big(\bigvee_{p} rec(l,p,m)\Big)\right)^\dagger$$

$$\kappa_6(l,p,m) = ch(l,p)\left(send(l,m)\Big(\bigvee_{p'} rec(l,p',m)\Big)\right)^\dagger$$

$$starve(p)\left(send(l,m)\Big(\bigvee_{p'\neq p} rec(l,p',m)\Big)\right)^\dagger$$

$$\vee \left(send(l,m) \vee \Big(\bigvee_{p'} rec(l,p',m)\Big)\right)^* \triangle (ch(l,p) \vee starve(p) \vee \lambda)$$

For a given SDYMOL system, q ranges over the set of processes, l ranges over the set of links, p and p' range over the set of (inbound) ports, $m \neq @$ and m' range over the set of message types, and, for a process q in the system, $P(q)$ denotes the set of (inbound) ports of q.

Figure 5

SDYMOL constraints

over all ports in the system, and the corresponding constraint,

$$\kappa_6(l,p,m) = ch(l,p)\left(send(l,m)\Big(\bigvee_{p'} rec(l,p',m)\Big)\right)^{\dagger}$$
$$starve(p)\left(send(l,m)\Big(\bigvee_{p'\neq p} rec(l,p',m)\Big)\right)^{\dagger}$$
$$\vee\left(send(l,m)\vee\Big(\bigvee_{p'} rec(l,p',m)\Big)\right)^{*}\triangle\left(ch(l,p)\vee starve(p)\vee\lambda\right).$$

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 $starve(p)$ 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 $starve(p)$ 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 defined message types m, 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. (Of course the constraints $\kappa_1(l,p)$ and $\kappa_5(l,p,m)$, for all defined message types m, still

restrict the order and number of messages sent to and received from l in these cases.) The collection of constraints $\kappa_6(l,p,m)$, for the links l, the ports p, and the defined message types m in the system, 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 an SDYMOL design. They are summarized in figure 5.

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

4. CSP

4.1. A CSP system

To illustrate the derivation of a constrained expression representing the possible behaviors of a CSP system, we use the CSP solution to the mutual exclusion problem shown in figure 6. It is based on the integer semaphore example presented by Hoare [12]. We briefly discuss only the aspects of the CSP semantics that pertain to this example; the reader is referred to Hoare's original paper [12] for a more complete description of CSP.

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.[1]

The semaphore process begins by assigning the value 1 to the variable VAL. It then executes as many iterations 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 (S4–5), at which point the corresponding guard(s) *succeed.* It then increments or decrements the value of VAL, depending on which of the successful guards

[1] We use "terminate" here, and in the remainder of this section, as in [12]. Termination of a CSP process corresponds to the "running to completion" of an SDYMOL process, or what is sometimes termed "successful termination".

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

$SEM \equiv$

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

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

(S3) $\quad \Box U_2?v() \rightarrow VAL := VAL + 1$ -- U_2 releases resource

(S4) $\quad \Box VAL > 0; U_1?p() \rightarrow VAL := VAL - 1$ -- grant resource to U_1

(S5) $\quad \Box VAL > 0; U_2?p() \rightarrow VAL := VAL - 1]$ -- grant resource to U_2

$USER_i \equiv$

(U1) $CONT_i$ **boolean;** $CONT_i := \mathbf{t};$ -- initialize continuation variable

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

(U3) $\quad \Box CONT_i \rightarrow S!p(); ur_i; S!v()]$ -- if not terminated, request, use

---- and release resource

$TERMINATOR_i \equiv U_i!e()$ -- send termination signal

Figure 6

CSP solution to the mutual exclusion problem

is arbitrarily selected for execution. A repetitive command in CSP terminates 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 a v-signal. On the other hand, if S starves and the value of VAL is positive, then S starves waiting for either a v-signal or a p-signal.

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 modified, the second guard in the alternative command succeeds. If this 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, then uses the resource, an activity that is abstractly represented by the ur_i command, and then 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, as appropriate) 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.

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.

4.2. The CSP constrained expression

Because the semantics of CSP are considerably more complex than the semantics of SDYMOL, the procedure for deriving CSP constrained expressions is more complex

than the procedure for deriving SDYMOL constrained expressions. Rather than give a full description of such a procedure here, we limit ourselves to describing the derivation of a constrained expression representation for the CSP system of figure 6. This description 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. It also shows how to express the semantics of 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 event symbols used in the constrained expression representation of this system are shown in figure 7. As indicated in the first six lines of this figure, event symbols 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 used in the representation of 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, a $send(\alpha)$ symbol indicates that the source process is ready to communicate, while a $send'(\alpha)$ symbol indicates that the communication has been realized and the source process is ready to continue with its sequential activity. These, along with the $rec(\alpha)$ 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 $wait(P', \alpha)$ symbol encodes information about the starvation event $starve(P')$. 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 for communications that can be realized. Similarly, the $hang(P)$ 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

Symbol	*Associated Event*
$use(X,a)$	Use of variable X that presumes it to have the value a
$def(X,a)$	Assignment of value a to variable X
$rec(P,Q,c)$	c-signal from process P to process Q
ur_i	Use of the resource by process U_i $(i = 1,2)$
$starve(P)$	Starvation of process P
$stop(P)$	Termination of process P
$send(P,Q,c)$	Process P ready to send a c-signal to process Q
$send'(P,Q,c)$	Process P ready to resume execution after sending a c-signal to process Q
$wait(P,P,Q,c)$	Process P starves waiting for process Q to become ready to receive a c-signal from P
$wait(Q,P,Q,c)$	Process Q starves waiting for process P to become ready to send a c-signal to Q
$hang(P)$	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 7

CSP event symbols

starvation of the process has already appeared. When instantiated for the CSP system of figure 6, the symbols shown in figure 7 constitute the augmented alphabet of the constrained expression representation for this system.

The system expression derived from figure 6 consists of the interleave of five process expressions:

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

The process expressions, which are shown in figure 8, represent the sequential activity of the five processes in the system. (The line numbers in this figure serve only for reference in the discussion below.)

The initial $def(VAL, 1)$ 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 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 $wait(S, U_i, S, v)$ symbols, for $i = 1, 2$. According to the CSP semantics, however, if both U_1 and U_2 terminate, the repetitive command terminates and thus S cannot starve. If S starves, one of U_1 or U_2 must also starve, as represented by the disjunction of the $hang(U_i)$ symbols, for $i = 1, 2$. Finally, the $starve(S)$ symbol signifies the starvation of S and an $ne(S)$ symbol immediately follows the $starve(S)$ symbol, since no further symbols from the process expression can appear after a $starve(S)$ symbol in a behavioral trace 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

$\pi_S =$

(1) $def(VAL, 1)$

(2) $\Big[rec(U_1, S, v) \Big(\bigvee_{j \in Z} use(VAL, j)\ def(VAL, j+1) \Big)$

(3) $\quad \vee\ rec(U_2, S, v) \Big(\bigvee_{j \in Z} use(VAL, j)\ def(VAL, j+1) \Big)$

(4) $\quad \vee \Big(\bigvee_{j>0} use(VAL, j) \Big)\ rec(U_1, S, p) \Big(\bigvee_{j \in Z} use(VAL, j)\ def(VAL, j-1) \Big)$

(5) $\quad \vee \Big(\bigvee_{j>0} use(VAL, j) \Big)\ rec(U_2, S, p) \Big(\bigvee_{j \in Z} use(VAL, j)\ def(VAL, j-1) \Big)$

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

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

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

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

$\pi_{U_i} =$

(1) $def(CONT_i, \mathrm{t})$

(2) $\Big[rec(T_i, U_i, e)\ def(CONT_i, \mathrm{f})$

(3) $\quad \vee\ use(CONT_i, \mathrm{t}) \Big(send(U_i, S, p)\ send'(U_i, S, p) \vee wait(U_i, U_i, S, p)\ starve(U_i)\ ne(U_i) \Big)$

(4) $\qquad\qquad ur_i \Big(send(U_i, S, v)\ send'(U_i, S, v) \vee wait(U_i, U_i, S, v)\ starve(U_i)\ ne(U_i) \Big)$

(5) $\quad \vee\ use(CONT_i, \mathrm{f})\ wait(U_i, T_i, U_i, e)\ hang(T_i)\ starve(U_i)\ ne(U_i) \Big]^*$

(6) $\quad use(CONT_i, \mathrm{f})\ term(T_i)\ stop(U_i)$

$\pi_{T_i} = \Big(send(T_i, U_i, e)\ send'(T_i, U_i, e) \vee wait(T_i, T_i, U_i, e)\ starve(T_i)\ ne(T_i) \Big)\ stop(T_i)$

Figure 8

Process expression derived from BS

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 6 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 an output command can only appear in a command list (i.e., not in a guard), starvation at an output command does not necessarily imply the starvation of another process. Thus, no $hang(S)$ symbol is required in this alternative. (Similarly, the translation of an input command appearing in a command list, rather than a guard, does not require any *hang* symbols.) The interpretation and generation of the process expression π_{T_i}, for $i = 1,2$, is obvious.

Seven types of constraints are required in the constrained expression representation of a CSP system to appropriately restrict the number and order of symbols in strings representing legal behaviors. 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,

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

over the constraint alphabet $\{ def(X,v), use(S,v) \}_{v \in t(X)}$. Clearly, these constraints assure that the dependency of guarded command execution on the value of a process' vari-

ables is accurately modeled 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 modeled in an SDYMOL constrained expression representation of an SDYMOL system.

The second type of constraint relates to 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,

$$\kappa_2(\alpha) = \Big(send(\alpha)\, rec(\alpha)\, send'(\alpha)\Big)^*,$$

defined over the constraint alphabet $\{\, send(\alpha), send'(\alpha), rec(\alpha) \,\}$, for each possible communication $\alpha \in C$. Given the interpretations of the event symbols shown in figure 7, this constraint is easily understood. For a communication $\alpha = (P, Q, c)$, a $rec(\alpha)$ 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. A $send(\alpha)$ symbol is associated with the readiness of P to send a c-signal to Q and a $send'(\alpha)$ 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 latter 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, a $send(\alpha)$ symbol follows all symbols representing events that precede the output event, while a $send'(\alpha)$ symbol precedes all symbols representing events that follow the output event. Besides representing the execution of corresponding input and output commands, a $rec(\alpha)$ symbol is used in a similar fashion in the process expression π_Q for the process Q. A $rec(\alpha)$ symbol follows symbols representing events that precede the input event and precedes symbols

representing events that follow the input event. By requiring that each $rec(\alpha)$ symbol lie between corresponding $send(\alpha)$ and $send'(\alpha)$ symbols, the constraint $\kappa_2(\alpha)$ assures that the processes P and Q are properly synchronized when 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 a constraint

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

and corresponding constraint alphabet $\{ stop(P), starve(P) \}$, 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 a constraint,

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

defined over the constraint alphabet $\{ ne(P) \}$, for each process 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 assures that processes do not starve waiting on communications that can be realized. For each communication $\alpha \in C$, we define the constraint,

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

over the constraint alphabet $\{ wait(source(\alpha), \alpha), wait(target(\alpha), \alpha) \}$, 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 is used to assure 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,

$$\kappa_6(P) = \Big(\mathit{hang}(P)^* \triangle \mathit{starve}(P)\Big) \vee \lambda,$$

which is defined over the alphabet $\{\mathit{hang}(P), \mathit{starve}(P)\}$. 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 $\mathit{starve}(Q)$ symbol that is produced by the translation of a guard in an alternative command, then it also contains a $\mathit{hang}(P)$ 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 $\mathit{hang}(P)$ symbol appears in a constrained prefix, the corresponding $\mathit{starve}(P)$ symbol must also appear. Therefore, 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 starves (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 required by the CSP semantics. 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 $\mathit{term}(P)$ symbols, for those processes P that must terminate before the repetitive command containing the alternative command terminates. The constraints,

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

and associated constraint alphabets $\{\mathit{stop}(P), \mathit{term}(P)\}$, for the processes P in a CSP system, thus assure that a $\mathit{term}(P)$ symbol cannot appear in a constrained prefix without the associated $\mathit{stop}(P)$. These constraints thus eliminate prefixes in which repetitive commands 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 9.

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

$$\kappa_2(\alpha) = \Big(send(\alpha)\, rec(\alpha)\, send'(\alpha) \Big)^*$$

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

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

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

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

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

Figure 9

CSP constraints

In the CSP system of figure 6, interprocess communication is limited to the exchange of simple timing signals. In a more general CSP system, however, a source process can send values which are assigned to target variables in 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 transfer of information between SDYMOL processes is represented in the constrained expression 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 [12]. 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 Dillon's dissertation [8].

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 domains 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.

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.

5. PETRI NETS

5.1. A Petri net system

The Petri net formalism is a classical model of concurrent systems. A Petri net is generally represented graphically as a collection of nodes, called *places* and *transitions*, connected by directed arcs [17]. A behavior of the concurrent system represented by the Petri net then corresponds to a sequence of transition *firings*, which move tokens between places in the net. 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 [11,16].

A labeled Petri net based on the solution to the mutual exclusion problem presented in [17] is shown in figure 10. The Petri net has seven *places*, P_1 through P_7, represented by circles, and six *transitions*, t_1 through t_6, represented by bars and labeled with symbols from the alphabet $\{p_1, p_2, v_1, v_2, ur_1, ur_2\}$. A *marking* of a Petri net is signified by putting dots, representing tokens, in places. There is thus a single token in each of the places P_1, P_4, and P_5, and no tokens in any of the places P_2, P_3, P_6, or P_7, in the initial marking of the Petri net shown in figure 10.

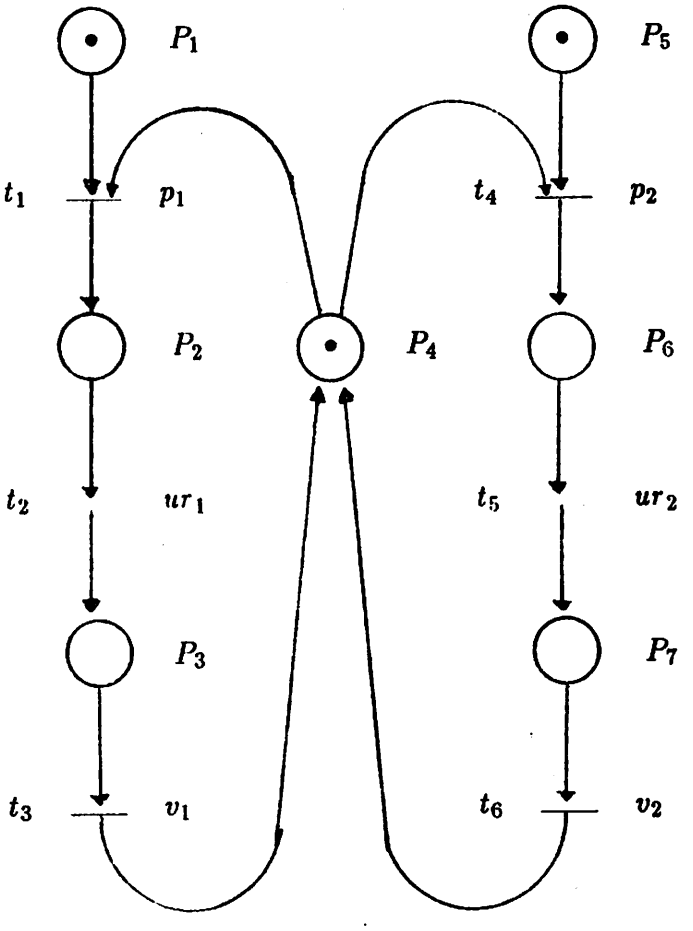


Figure 10

A Petri net solution to the mutual exclusion problem

The arcs connecting transitions and places in a Petri net determine the rules govern-

ing the firing of transitions. A transition t_m is *fireable* if there is a token in each place P_i such that there is an arc from P_i to t_m. When t_m *fires*, a token is removed from each such P_i, and a token is added to each of the places P_j such that there is an arc from t_m to P_j. In the Petri net of figure 10, therefore, the transition t_1 is fireable, and if fired, results in a token being removed from the place P_1, a token being removed from the place P_4, and a token being added to the place P_2. The transition t_4 is also fireable. The firing of t_4 results in a token being removed from P_4, a token being removed from P_5, and a token being added to P_6. 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 $t_1 t_2 t_3$ in the Petri net of figure 10 thus determines the string $p_1\, ur_1\, v_1$.

The place P_4 models a semaphore process in the Petri net of figure 10. The number of tokens in P_4 represents the value of the semaphore. Dijkstra's P and V operations are modeled by the transitions labeled p_i and v_i, while the use of the resource is modeled by the transitions labeled ur_i, for $i = 1, 2$. A P operation (a firing of t_1 or t_4) decrements the value of the semaphore (removes a token from P_4) and enables a transition representing a use of the resource (t_2 or t_5). A V operation (a firing of t_3 or t_6) increments the value of the semaphore (adds a token to P_4). Clearly the transitions representing the use of the resource are mutually exclusive.

The Petri net constrained expression

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 *end*. The augmented alphabet for the constrained expression derived from the Petri net of figure 10 is thus given by,

$$\{ p_i, v_i, ur_i \}_{i=1,2} \cup \{ put_i, take_i \}_{i=1,\ldots,7} \cup \{ end \}.$$

Naturally, the terminal alphabet is just the set of transition labels. In this example, therefore, we have the terminal alphabet $\{p_1, p_2, v_1, v_2, ur_1, ur_2\}$.

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

$$\epsilon = \iota\Big(\bigvee_m \pi_m\Big)^* end,$$

where m ranges over the indices of the transitions in the Petri net and *end* 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 and transition expressions derived from the labeled Petri net of figure 10 are shown in figure 11.

The initial expression is simply a concatenation of put_i symbols, one for each token residing in a place P_i in the initial marking. The initial expression thus indicates the initial number of tokens and their locations in the Petri net.

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 of figure 11, for example, it is evident that the firing of t_1, associated with the symbol p_1, removes a token from the place P_1, removes a token from the place P_4, and adds a token to the place P_2. 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

$$\iota = put_1\ put_4\ put_5$$

$$\pi_1 = take_1\ take_4\ p_1\ put_2$$

$$\pi_2 = take_2\ ur_1\ put_3$$

$$\pi_3 = take_3\ v_1\ put_1\ put_4$$

$$\pi_4 = take_4\ take_5\ p_2\ put_6$$

$$\pi_5 = take_6\ ur_2\ put_7$$

$$\pi_6 = take_7\ v_2\ put_4\ put_5$$

$$\epsilon = \iota\,(\pi_1 \vee \pi_2 \vee \pi_3 \vee \pi_4 \vee \pi_5 \vee \pi_6)^*\,end$$

Figure 11

System expression derived from the Petri net presented in figure 10

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,

$$\kappa_1 = end,$$

defined over the constraint alphabet $\{ end \}$. 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,

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

over the constraint alphabet $\{ put_i, take_i \}$, 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,

$$\kappa_2(P_i) = (put_i\, take_i)^\dagger \triangle \left(\underbrace{put_i \cdots put_i}_{n_i \text{ times}} \right),$$

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

Assume, for example, that the initial marking of the Petri net shown in figure 10

$$\kappa_1 = end$$

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

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

$$\kappa_2(P_3) = (put_3\ take_3)^\dagger$$

$$\kappa_2(P_4) = (put_4\ take_4)^\dagger \triangle put_4$$

$$\kappa_2(P_5) = (put_5\ take_5)^\dagger \triangle put_5$$

$$\kappa_2(P_6) = (put_6\ take_6)^\dagger$$

$$\kappa_2(P_7) = (put_7\ take_7)^\dagger$$

Figure 12

Constraints derived from the Petri net of figure 10

also defines the desired final marking of the Petri net. The constraints derived from the Petri net in this case are shown in figure 12.

6. CONCLUSION

In this paper we have demonstrated the broad applicability of the constrained expression formalism. In particular, we have shown how it can be used with development notations providing different communication primitives (synchronous and asynchronous), based on different underlying models of computation (state machines and Petri nets), and appropriate to different stages of software development. In addition to the three notations discussed here, the constrained expression formalism has also been used with an Ada-based design language [2] and with a notation (another subset of DYMOL) that provides primitives for dynamically altering the communication pathways in a distributed system [25]. This range of applications demonstrates the broad applicability of the constrained expression formalism as a representation for distributed system behaviors.

Further evidence for its broad applicability comes from two other projects currently underway at the University of Massachusetts. In one of these, an event-based language closely related to the constrained expression formalism is being used as a basis for high-level debugging of distributed systems [4]. A prototype toolset supporting this debugging

method has been implemented and is currently being used in conjunction with a distributed problem-solving testbed system by a research group at the University. Essentially the same event-based language is being used by another research group in an intelligent user interface system that is part of an office automation project [7]. Here the language is used to describe various office procedures to the interface system, which then uses those descriptions to guide or assist users in carrying out the office procedures.

Our primary concern, however, is with using the constrained expression formalism as a basis for building tools for distributed software system development. It is in this context that the broad applicability of the constrained expression approach is particularly significant. The constrained expression representation of a distributed system provides a framework for formally reasoning about properties, such as mutually exclusive use of shared resources and absence of deadlock, that can be characterized by the patterns of certain event symbols appearing in strings representing behaviors of a system. This analysis is facilitated by the use of finite representations (i.e., constrained expressions) for potentially infinite sets of behaviors. Using the constrained expression representation of a system, one reasons about the sequences of events in all the behaviors of a set collectively, rather than one at a time. This reduces the problems of combinatorial explosion inherent in most distributed system analysis techniques. The constrained expression representation of a distributed system also facilitates analysis by directly encoding relationships between the order and number of occurrences of particular events in behaviors of a system. It thus highlights the information required for reasoning about behavioral properties that involve interactions among the parts of a distributed system.

We have developed a number of techniques for analyzing constrained expressions. Detailed presentations of these techniques appear in [3] and [8], while their application in a realistic distributed software development setting is illustrated in [2]. In the remainder of this paper we describe our ongoing efforts toward automating these analysis techniques as part of a prototype toolset supporting the constrained expression approach.

Although significant as an illustration of the broad applicability of the constrained expression approach, the use of constrained expressions and their associated analysis techniques in conjunction with SDYMOL, CSP or Petri nets seems unlikely to be of much near-term practical value to developers of distributed software. Our work on a prototype

toolset has therefore been directed toward tools specifically applicable to an Ada-like design language that we have developed. This language focuses on the description of synchronization and communication in multi-task systems [2]. To achieve this focus, it provides a full range of Ada-style control constructs and task interaction (Ada rendezvous) features, but only a limited set of data definition and manipulation constructs. In particular, the language has a very limited set of primitive data types, consisting primarily of enumeration types, which are easily represented and are sufficient for high-level expression of control flow dependencies. The prototype toolset, with its corresponding focus on analysis of synchronization and communication properties of distributed systems, will comprise a constrained expression *deriver*, a *simplifier*, a *behavior generator*, and a collection of *analysis tools*.

The deriver is a tool to automate the process of producing a constrained expression representation from a software description in some other notation. A first version of a deriver has recently been implemented at the University of Massachusetts, Amherst [21] and is now being tested both there and at the University of California, Santa Barbara. This deriver is implemented in Ada and produces the constrained expression representations corresponding to distributed software system designs expressed in our Ada-like design language. Its implementation is carefully organized to support a table-driven approach to translation, so that derivers for later versions of that language or for other software development notations can be easily created by modifying the current version.

The derivation procedure associated with a particular distributed system development notation is completely general, so that it can be applied to any syntactically correct description. As a result, however, the constrained expression derived from a given system description is often unnecessarily complex. As a first step toward analysis, therefore, it is useful to simplify the constrained expression representation of a system using, for example, the reduction procedure and message flow analysis algorithms described in [8]. A tool to automate this process, called the simplifier, is currently under construction at the University of California in Santa Barbara. This tool will accept the constrained expression representation produced as output by the deriver and will produce a simplified version of that constrained expression representation. This simplification step often leads to dramatic reduction in the effort required for later analyses. In constructing the simplifier, we

are also exploring some intriguing relationships between constrained expression simplification and some static analysis techniques proposed for application to distributed software systems [22].

The behavior generator is a tool for producing example behaviors from a constrained expression representation. We have experimented with a prototype version of this tool, implemented in PROLOG at the University of Massachusetts [1]. This version is capable of producing an example behavior from the set described by a given constrained expression, and can also be interactively guided in a search for a behavior possessing specific properties. Such capabilities are potentially of great value to users of the constrained expression toolset. Using the behavior generator, a developer of distributed software can explore properties of the system, detect certain classes of errors and recognize possible improvements. Moreover, in the course of carrying out the algebraic analysis, the analysis tools will produce vast amounts of information characterizing classes of behaviors that have some particular property. The behavior generator tool will help the developer to transform this information into concrete examples of behaviors having the given property. Without such help, a developer would find the output of the analysis tools much more difficult to understand and use.

The analysis tools will automate the analysis techniques that have been developed for constrained expressions. In particular, the algebraic analysis techniques described in [3] are naturally applied to the constrained expression representation of a distributed system. These techniques determine whether a particular event, or pattern of events, appears in any possible behavior described by a given constrained expression. This determination is made through a process of iteratively generating inequalities involving numbers of occurrences of events in different subsequences of such a behavior. The generation process is driven by the form of the constrained expression. If the set of inequalities should become inconsistent at any point during this process, then the event or pattern of events in question cannot occur in any possible behavior described by the constrained expression. Otherwise, the resulting set of inequalities characterizes the set of behaviors containing the event or event pattern. A first implementation of analysis tools supporting the algebraic techniques is presently under development at the Unversity of Massachusetts. Designing appropriate methods and heuristics for guiding the inequality generation process is a major focus of

our current work on these tools.

The broad applicability of the constrained expression framework dramatically increases the value of our work on the prototype toolset. Because most of the tools are designed to work not on some specific distributed software development notation, but rather on a constrained expression representation of possible distributed system behavior, they can be readily used in conjunction with any notation that a developer might wish to adopt. Naturally, an appropriate deriver would be required for producing constrained expression representations from descriptions in the chosen notation. Given a suitable set of translation rules, however, this is primarily a matter of some rather straightforward modifications to the tables in our table-driven deriver. Certain aspects of the operation of the other tools in the toolset may of course rely upon specific details of the particular form of constrained expressions that correspond to designs in the the Ada-like notation. For the most part, however, the valuable capabilities of the simplifier, behavior generator and analysis toolset should be readily adapted to support analysis of descriptions in other notations as well. This will permit developers of distributed software to select a notation, or several notations, based on the naturalness and expressive power of the notation's constructs rather than on the availability of suitable analysis tools.

We anticipate that our ongoing research on this project will lead to enhancements of several kinds. For instance, developing the necessary translation rules for producing constrained expression representations from our Ada-like design language led to a natural extension of the constrained expression formalism that permits a more efficient representation of certain constraints on a system's behaviors. This extended formalism facilitates, for example, the description of the FIFO ordering of tasks on the queues associated with entries. While our toolset design is based on existing analysis techniques, we are simultaneously investigating methods for strengthening and extending these techniques. Initial experimentation with the constrained expression-based analysis tools should provide valuable insight for their further development and refinement and will permit some preliminary evaluation of their practicality when applied to descriptions of realistic size and complexity. It may also suggest additional improvements in the constrained expression formalism. Any and all of these enhancements will further increase the value of the broad applicability of the constrained expression approach.

APPENDIX

Formal definition of constrained expressions

We let $\mathcal{RE}(A)$ denote the set of regular expressions over the alphabet A, where here 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 $\emptyset$) by finite applications of the usual regular expression operators, alternation (represented by $\vee$), concatenation (represented by juxtaposition), and transitive closure (represented by *), and an additional operator, the shuffle or interleave operator (represented by $\triangle$). (The relative precedence of the regular expression operators, from highest to lowest, is *, juxtaposition, $\triangle$, and $\vee$.) The operator $\triangle$ has been shown to preserve regularity [9] and is useful for representing concurrent activity. The regular expression $ab \triangle cd$, 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, then $ab \triangle cd$ represents the possible behaviors of the two processes executing concurrently.

In order to express certain constraints on the order and number of symbols that appear in legal behavioral traces of a system, we need to introduce an additional operator, the unary concurrent closure, or dagger, operator (represented by †). The † represents the shuffle of zero or more copies of its argument, and is at the same level of precedence as *. Thus, for instance, $(ab)^\dagger$ represents the set of strings consisting of equal numbers of the symbols a and b, with the additional property that, in any prefix, the symbol a occurs at least as many times as the symbol b. The † operator is used in constraints to ensure, for example, that at least as many messages have been sent as have been received.

The *event expressions* over an alphabet A are formed from the symbols of A, the null string, and the empty set by finite applications of the regular expression operators (including $\triangle$) and the † operator. We write $\mathcal{EE}(A)$ for the set of event expressions over A.

Let $S \subseteq A$. We define a homomorphism $\rho_S : A^* \rightarrow S^*$, called *projection on S*, by extending the map $A \rightarrow S^*$ given by

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

We think of ρ_S as "erasing" the symbols not belonging to S. Let u be a string of symbols from A. For $\kappa \in \mathcal{EE}(S)$, we say that u *satisfies* κ if $\rho_S(u)$ belongs to the language of κ. We can regard κ as imposing a constraint on the way in which the symbols belonging to S can occur in strings; u satisfies κ if this constraint is satisfied.

Essentially, a constrained expression consists of a regular expression ϵ over an alphabet A of event symbols, together with a collection of subsets of A and event expressions over those subsets. These event expressions are regarded as constraining the patterns of event symbols, and we are interested only in those prefixes of the language of ϵ which satisfy all of the event expressions. More formally, we have

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

(i) $\mathbf{F} = (A, E, \hat{S}, \hat{C})$, where

  (a) A is an alphabet of event symbols,

  (b) E is a subset of A,

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

  (d) $\hat{C} = \{\kappa_j\}_{j \in J}$, where $\kappa_j \in \mathcal{EE}(S_j)$, for each $j \in J$, and

(ii) $\epsilon \in \mathcal{RE}(A)$.

We say that A is the *augmented alphabet* of the constrained expression and E is the *terminal alphabet.* The regular expression ϵ is the *system expression*, the κ_j are called *constraints*, and the S_j are the *constraint alphabets.* The symbols in the terminal alphabet represent system events of interest to the designer. The augmented alphabet consists of these symbols and others required for technical reasons, such as the *ne* symbol used in sections 3 and 4.

Let $\mathbf{D} = (\mathbf{F}, \epsilon)$ be a constrained expression, and let $P(\epsilon)$ be the collection of prefixes of the language of the regular expression ϵ. A prefix $u \in P(\epsilon)$ is called a *constrained prefix* if it satisfies all the constraints κ_j, i.e., if, for each $j \in J$, $\rho_{S_j}(u)$ belongs to the language of κ_j. We write $P(\epsilon)|_{\hat{C}}$ for the set of constrained prefixes of $\mathbf{D}$. Finally, the *interpreted language* of $\mathbf{D}$, $I\mathcal{L}(\mathbf{D})$, is obtained by projecting the constrained prefixes onto the terminal alphabet. Thus, $I\mathcal{L}(\mathbf{D}) = \rho_E(P(\epsilon)|_{\hat{C}})$. It is this interpreted language that corresponds to the behaviors of the system represented by the constrained expression $\mathbf{D}$.

References

[1] S. Avery, "Development of a Behavior Generator for Constrained Expressions," Dept. of Comp. and Info. Science, Univ. of Massachusetts, Amherst, SDLM/84-2, June 1984.

[2] G. Avrunin, L. Dillon, J. Wileden, and W. Riddle, "Constrained Expressions: Adding Analysis Capabilities to Design Methods for Concurrent Software Systems," *IEEE Trans. on Software Engineering*, vol. SE-12, no. 2, 278–292, February 1986.

[3] G. Avrunin and J. Wileden, "Describing and Analyzing Distributed System Designs," *ACM Trans. on Programming Languages and Systems*, vol. 7, no. 3, 380–403, July 1985.

[4] P. Bates and J. Wileden, "High Level Debugging of Distributed Systems," *Journal of Systems and Software*, vol. 3, no. 4, 255–264, December 1983.

[5] R. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expressions," *Lecture Notes in Computer Science*, vol. 16, Springer-Verlag, Heidelberg, 1974, 89–102.

[6] B. Chen and R. T. Yeh, "Formal Specification and Verification of Distributed Systems," *IEEE Trans. on Software Engineering*, vol. SE-9, no. 6, 710–722, November 1983.

[7] W. Croft and L. Lefkowitz, "Task Support in an Office System," *ACM Trans. on Office Info. Systems*, vol. 2, no. 3, 197–212, July 1984.

[8] L. Dillon, "Analysis of Distributed Systems Using Constrained Expressions," Ph.D. Dissertation, Dept. of Comp. and Info. Science, University of Massachusetts, Amherst. Available as Dept. of Computer and Info. Science Technical Report TR 84-18, September 1984.

[9] S. Ginsburg, *The Mathematical Theory of Context-Free Languages*. McGraw Hill, New York, 1966.

[10] I. Greif, "A Language for Formal Problem Specification," *Comm. ACM*, vol. 20, no. 12, 931–935, December 1977.

[11] M. Hack, "Petri Net Languages," Computation Structures Group, Massachusetts Institute of Technology, Cambridge, Memo 124, June 1975.

[12] C. Hoare, "Communicating Sequential Processes," *Comm. ACM*, vol. 21, no. 8, 666–677, August 1978.

[13] C. Hoare, "Communicating Sequential Processes," *Prentice-Hall*, Englewood Cliffs, New Jersey, 1985

[14] P. Lauer, P. Torrigiani and M. Shields, " COSY: A System Specification Language

Based on Paths and Processes," *Acta Informatica*, 451–503, 1979.

[15] J. Misra and K. Chandy, "Proofs of Networks of Processes," *IEEE Trans. on Software Engineering*, vol. SE-7, no. 4, 417–426, July 1981.

[16] J. Peterson, "Computation Sequence Sets," *Journal of Comp. and System Sciences*, vol. 13, no. 1, 1–24, August 1976.

[17] J. Peterson, "Petri Nets," *ACM Computing Surveys*, vol. 9, no. 3, 223–252, September 1977.

[18] W. Riddle, "An Approach to Software System Behavior Modeling," *Computer Languages*, Vol. 4, 29–47, 1979.

[19] A. Shaw, "Software Descriptions with Flow Expressions," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 3, 242-254, May 1978.

[20] A. Shaw, "Software Specification Languages based on Regular Expressions," Institut für Informatik, Eidgenössische Technische Hochschule, Zürich, June 1979.

[21] U. Sundaram, "A Constrained Expression Deriver for an Ada-Like Design Language," Dept. of Comp. and Info. Science, Univ. of Mass., Amherst, Tech. Rpt. TR 86-19, April 1986.

[22] R. Taylor, "A General Purpose Algorithm for Analyzing Concurrent Programs," *Comm. ACM*, vol. 26, no. 5, 362-376, May 1983.

[23] M. Welter, "Counter Expressions," Dept. of Comp. Science, Univ. of Michigan, Ann Arbor, RSSM/24, October 1976.

[24] J. Wileden, "Techniques for Modelling Parallel Systems with Dynamic Structure," *Journal of Digital Systems*, 177–197, Summer 1980.

[25] J. Wileden, "Constrained Expressions and the Analysis of Designs for Dynamically-Structured Distributed Systems," *Proc. 1982 Int. Conf. on Parallel Processing*, 340–344, August 1982.