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

LAURA K. DILLON
University of California, Santa Barbara
and
GEORGE S. AVRUNIN and JACK C. WILEDON
University of Massachusetts, Amherst

It is extremely difficult to characterize the possible behaviors of a distributed software system through informal reasoning. Developers of distributed systems require tools that support formal reasoning about properties of the behaviors of their systems. These tools should be applicable to designs and other preimplementation descriptions of a system, as well as to completed programs. Furthermore, they should not limit a developer's choice of development languages.

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 with this representation to establish properties of the system. Examples of these formal analysis techniques appear elsewhere. Here we illustrate the broad applicability of the constrained expression formalism by showing how constrained expression representations are obtained from descriptions of systems in three different notations: SDYMOL, CSP, and Petri nets. Features of these three notations span most of the significant alternatives for describing distributed software systems. Our examples thus offer persuasive evidence for the broad applicability of the constrained expression approach.

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**]: Language classifications—*design languages*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs

General Terms: Design, Languages, Theory, Verification

Additional Key Words and Phrases: Constrained expressions, CSP, distributed software systems, event-based analysis of software designs, Petri nets, Petri net languages, SDYMOL, software design tools

This work was 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.

Authors' addresses: L. K. Dillon, Computer Science Department, University of California, Santa Barbara, CA 93106; G. S. Avrunin, Department of Mathematics and Statistics, University of Massachusetts, Amherst, MA 01003; J. C. Wiledon, Department of Computer and Information Science, University of Massachusetts, Amherst, MA 01003.

1. INTRODUCTION

The large number and complexity of the possible interactions among the asynchronous components of a distributed software system make it difficult to reason informally about its behavior. Developers of distributed systems therefore require powerful formal techniques for analyzing their systems. Formal analysis is important throughout the software development process, especially in the early preimplementation phases when errors are most easily corrected.

Developers of distributed systems also need congenial development notations, including design and implementation languages. Different notations are suitable at different stages of software development, and features of the system under development may also make one language more appropriate than another. Software developers therefore may need to use a variety of development notations, even on a single project.

Unfortunately, developers are often forced to choose between using formal analysis methods and appropriate development notations. Virtually all proposed analysis techniques are based on some particular notation and rely on special features of that notation. Software developers who wish to use these techniques then must work with a single development notation, which may not be suitable for their immediate tasks. In addition, many of these techniques force developers to work with sophisticated and abstract mathematical structures, rather than standard design and implementation languages.

One goal of our work has been the *broad applicability* of analysis methods. We are producing tools that support formal analysis methods without imposing unnatural development notations on developers. Our approach to broad applicability is based on a formalism, called *constrained expressions*, that can be used with a wide variety of standard notations. This allows a developer to choose a notation appropriate for a system and its current state of development. A description of the system in the chosen notation is then mechanically translated into a constrained expression representation of the behavior of the system, which provides a basis for formal analysis.

Whereas a development notation is chosen to facilitate description and maximize expressive power, constrained expressions are designed to encode the information required for analysis of important system properties. The constrained expression representation of a distributed system provides a formal basis for arguments concerning the order and number of occurrences of particular events in behaviors of the system. Arguments of this nature have been widely used to analyze distributed systems for such properties as mutual exclusion, deadlock, and starvation (see, e.g., [3, 7, 15, 18, 20, 21]). The constrained expression formalism thus offers a general intermediate form that both supports formal analysis and allows development in the appropriate notations.

In a previous paper [5], we showed how constrained expressions are used to add analysis capabilities to a (single) particular distributed system design language. Further development of these techniques are described in a forthcoming report [2]. Algebraic manipulations or "simplifications" of constrained expressions are described in a companion paper [12]. Here we illustrate the broad applicability of the constrained expression approach by showing how constrained expression representations can be derived from descriptions of systems in three 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. These examples offer persuasive evidence for the broad applicability of the constrained expression approach.

In Section 2, we describe the constrained expression formalism. Constrained expression formulations of SDYMOL, CSP, and Petri nets are presented in Sections 3, 4, and 5, respectively. For purposes of illustration, we give a constrained expression representation for a semaphore system described in each notation. The accuracy and precision of constrained expression formulations of these notations are considered in Section 6. Finally, in Section 7, we discuss our experience with the constrained expression approach and the directions of our current research.

2. THE CONSTRAINED EXPRESSION FORMALISM

In this section we briefly describe the constrained expression formalism. A more formal description is given in the Appendix.

In the constrained expression framework, any particular behavior of a distributed system is viewed as defining a sequence of *event* occurrences. An *event symbol* is associated with each (potential) event occurrence. A single behavior thus determines a string over an event alphabet, and the set of all possible system behaviors determines a language over this alphabet.

Exactly what things are considered to be events depends upon the particular system and the level of detail at which it is considered. We assume only that events are indivisible and nonoverlapping. Overlapping activities are easily represented by treating their initiations and terminations as separate events.

A constrained expression consists of a collection of expressions over an alphabet of symbols called the *augmented alphabet.* The augmented alphabet contains symbols representing each potential event occurrence. It also contains symbols required to capture important facets of a particular notation, as will be seen in the examples in the next three sections. Such symbols do not necessarily correspond to visible events. The expressions are interpreted as representing a language. The constrained expression representation of a distributed system is defined so that this language is exactly the language determined by the set of possible system behaviors.

Essentially, a constrained expression consists of two parts, 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, alternation ($\vee$), concatenation (juxtaposition), and Kleene star ($*$), plus the *interleave* or *shuffle* operator ($\otimes$).[1] The interleave of two event strings represents their concurrent occurrence. The regular expression $ab \otimes cd$, for example, denotes

[1] The relative precedence of the regular expression operations from highest to lowest is $*$, juxtaposition, $\otimes$, and $\vee$.

the set {*abcd, acbd, acdb, cabd, cadb, cdab*}. It represents the concurrent occurrence of the event sequences *ab* and *cd*.

A system expression is viewed as a generator of "candidate" event sequences. That is, each prefix of a string in the language of the system expression is considered a candidate for a possible behavior. 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. The set of candidates contains event strings representing all the possible behaviors, but it may also include extraneous event strings. We consider the set of prefixes in conjunction with the constraint set to determine which candidates represent possible system behaviors.

The constraint set consists of a collection of *constraints*. Each constraint is an expression over the augmented alphabet and is formed by using the regular operators (including interleave) plus the *dagger* operator (†). This unary operator represents the interleave of zero or more copies of its argument.[2] The dagger describes some number of concurrent occurrences of an event sequence.

Each constraint represents a pattern of acceptable event sequences with respect to a subset of the augmented alphabet called a *constraint alphabet*. Every candidate event sequence is "filtered" through the constrained expression's constraint set to determine whether it represents a possible behavior. This filtering process involves the following steps. First, the candidate prefix is *projected* on the alphabet of some constraint. This involves "erasing" all symbols that are not in the constraint alphabet. If the resulting string belongs to the language of the constraint, the prefix is said to *satisfy* that constraint. Otherwise, the prefix contains an unacceptable pattern of event symbols, and it is eliminated from the set of candidates. Those prefixes that satisfy all the constraints in the constraint set are called *constrained prefixes*.

As a final step, the constrained prefixes are projected on a *terminal alphabet*. This alphabet is the subset of the augmented alphabet consisting of event symbols that correspond to significant system events. This yields the *interpreted language* of the constrained expression. The interpreted language represents exactly the set of possible behaviors of the distributed system. It is important to note that analysis techniques operate on the constrained expression, rather than on individual strings in the interpreted language. It is therefore never necessary to actually generate this language.

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 directly generating the set of possible behaviors. We have found it, however, to be both simpler and more natural. It is convenient to use the system expression, which describes the candidate event sequences, and the constraints, which are used to eliminate some of those sequences, for different purposes. We typically use the system expression to describe behavioral properties of a system. For example, a system expression might express the fact that some component of the distributed system first tries to receive three messages, then selects between two possible recipients and sends a message to one of them.

[2] The interleave operator preserves regularity [13], whereas the dagger operator does not.

Constraints are used primarily to express fixed semantic properties for a class of distributed systems. For instance, constraints enforce the synchronous nature of interprocess communication in a constrained expression for a CSP system and the Petri net firing rules in a Petri net constrained expression.[3]

Constrained expressions are usually generated from some other description of a system. As demonstrated below, it is possible to derive a constrained-expression representation of a system from descriptions in a wide variety of notations, including many design and programming languages. For each such notation, constrained expressions are derived using a set of *translation rules* and a set of *constraint templates*. The translation rules direct the transformation of a design or program into a system expression. The constraint templates provide generic versions of the constraints. They are instantiated for a given design or program to produce a particular constraint set.

Considerable effort and insight is required to develop the translation rules and constraint templates for a given notation. Once they have been developed, however, it is an entirely mechanical procedure to carry out the derivation process for a particular system's description.

2.1 Related Work

Regular expression-like descriptions of the sets of sequences of events representing system behaviors are also found in the work of Campbell and Habermann [7] (path expressions), Riddle [27] (message transfer expressions and event expressions), Welter [32], (counter expressions), and Shaw [28] (flow expressions). Message transfer expressions, event expressions, flow expressions, counter expressions, and the COSY notation [22], which is based on path expressions, all rely on a filtering procedure. A set containing all legal system behaviors is represented by one or more expressions. Sequences that do not represent behaviors are eliminated from this set to produce the set of possible system behaviors.

The major difference between the constrained expression formalism and most of its predecessors (message transfer expressions, event expressions, counter expressions, and flow expressions) is the use of constraints to specify the conditions under which strings are eliminated from the set of possible behaviors. The filtering procedures used with the earlier notations are equivalent to using a fixed, predefined set of constraints expressing synchronization requirements between communicating processes.

The COSY notation closely resembles constrained expressions but is intended for quite different purposes. COSY is intended for specifying concurrent software systems. Events in COSY correspond to operations or procedures, rather than arbitrary system events. Path expressions, the COSY analogue of constraints, restrict the order in which operations are invoked. A COSY description thus provides a declarative specification of system behaviors, presumably for use in verification of a design or implementation. A constrained expression, on the

[3] There is nothing in the constrained expression framework that forces this separation of concerns, and we do not always adhere strictly to it ourselves. The two parts of the constrained expression framework provide a flexible approach to representing the possible behaviors of distributed systems. Decisions on 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.

other hand, does not define the intended behaviors of a system. It is a representation of the system's possible behaviors that is derived from some other system description, and it is used for exploring properties of the system.

Trace models of distributed systems treat behaviors as sets of sequences of communication events [19, 23]. Properties of the systems are verified using axiomatic proof techniques. A generalization of this approach models behaviors as infinite sequences of *observations* [24] and uses temporal logic for proving properties of behaviors. This approach has the advantage that liveness properties are more easily specified. In contrast, the constrained expression approach uses a more general concept of event and provides algebraic methods for analyzing properties, such as absence of deadlock, limited use of shared resources, and most liveness properties, which can be interpreted as questions about the order and number of certain event occurrences.

In event-based models that rely on explicit partial orderings [8, 14], a system's behavior is represented by a set of events and partial order relations. The relations express time orderings or enabling relationships. Events that are not comparable are considered concurrent. The system expression and constraints in a constrained expression representation of a system can be viewed as imposing certain partial order relations on the set of events. This model is therefore compatible with the constrained expression formalism.

3. CONSTRAINED EXPRESSIONS FOR SDYMOL

3.1 An SDYMOL System

SDYMOL is a high-level design language focusing on interprocess communication and synchronization.[4] A concurrent system in SDYMOL is a collection of sequential processes executing concurrently and communicating by means of message transmission. The SDYMOL design for a process specifies how the process interacts with other processes through message transmission. It only abstractly describes the internal activities of the process itself.

Message transmission in SDYMOL is both a communication and a synchronization mechanism. Each process contains a memory location called its *buffer* and a number of named *ports.* The buffer holds a message that is sent or received through a port. To send or receive messages through a port, the port must be connected to a *link* by a *channel.* An *inbound* port can be connected to several links, whereas an *outbound* port can only be connected to a single link. A link is an unbounded, unordered repository for messages that have been sent through the outbound ports connected to the link and not yet received through any inbound ports connected to it. Messages are represented by a finite number of *message types.*

A process sends a message through an outbound port p by executing a **send** p statement. This causes the current contents of the buffer of the process to be copied into the link l connected to p. The contents of the buffer are not modified. If p is not connected to a link, **send** p is equivalent to a null statement.

[4] SDYMOL is a simplified version of the Dynamic Modeling Language (DYMOL), which was designed to be used with the Dynamic Process Modeling Scheme (DPMS) [33].

A process requests receipt of a message through an inbound port p by executing a **receive** p statement. The request is fulfilled by nondeterministically selecting a link l that contains one or more messages and is connected to p, nondeterministically selecting a message m from the messages in l, removing m from l, and placing m into the process'es buffer. If there are no messages in any of the links connected to p, the requesting process waits. The wait continues at least until a message becomes available. The appearance of a message in a link connected to p does not necessarily end a wait. Competing requests might be lodged in the interim, and requests are not serviced in any particular order.

The syntax of SDYMOL is based on that of Algol 60. SDYMOL provides 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 represented by primitive statements consisting of user-defined identifiers. Such statements are semantically null. They serve as placeholders for activities that are to be elaborated in subsequent system descriptions.

A SDYMOL design for Dijkstra's solution to the mutual exclusion problem is shown in Figure 1. The system consists of three processes represented by the circles labeled s, u_1, and u_2. Boxes represent links. Arrows connecting links and ports represent communication channels.

The *user processes*, u_1 and u_2, periodically require access to some shared resource. The *semaphore process* s acts as a binary semaphore. The semaphore assures that the resource is used in a mutually exclusive fashion.

The availability of the resource is represented by an *ok* message residing in the $s.p$ link of the semaphore process.[5] The **send** $s.p$ statement in the semaphore process enables Dijkstra's P operation.

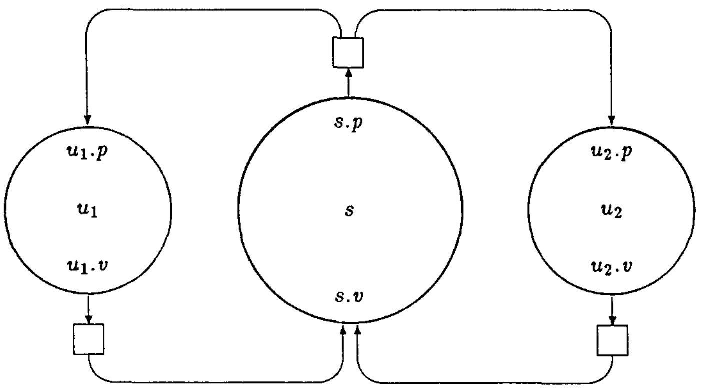
The **receive** $u_i.p$ statements in the user processes represent the P operations. If both u_1 and u_2 try to execute this statement simultaneously (they simultaneously attempt a P operation), one of the processes receives the single *ok* message residing in the link. The other process waits until a message becomes available. That is, one process is granted access to the resource and the other one waits. When a user process is finished with the resource, it deposits an *ok* message in its $u_i.v$ link. The **receive** $s.v$ statement in the semaphore process models Dijkstra's V operation.

The **internal test** predicate in the body of a user process represents some internal process computation. The results of this computation determine if the cycle requiring access to the resource is repeated. This cycle is represented by the three statements in the body of the **while** statement. The *use-resource*$_i$ statement is an example of an identifier statement. It models an internal activity (use of the resource) performed by u_i.

3.2 The SDYMOL Constrained Expression

We show how to derive a constrained expression representation for the SDYMOL system shown in Figure 1, after first describing the augmented and terminal alphabets.

[5] We identify an outbound port with the associated link here and in the rest of the paper. This is possible because the structure of an SDYMOL system is static.



```
u_i:
    while internal test do
        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.
```

Fig. 1. SDYMOL solution to the mutual exclusion problem.

The event symbols used in the constrained expression representation for the system are shown in Figure 2. Conceptually, we associate these symbols with certain system events, as indicated in this figure. Some of these symbols, however, are needed in constraints for describing permissible event sequences and do not correspond to actual system events.

Event symbols represent 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, while a *starve* symbol represents an attempt to receive that results in starvation (the process waiting forever). The *ne*, or *nonevent*, symbol does not represent an actual system event, since it does not occur in any string representing a legitimate system behavior. It is used in constraints assuring that certain patterns of event symbols do not occur in constrained prefixes. We discuss the role of this symbol in more detail below.

The terminal alphabet is determined by the questions that analysis is intended to address. Suppose, for example, we want to determine if the user processes in Figure 1 can starve and if the resource use is mutually exclusive. The former question can be determined by asking if there are any behavior sequences containing a $starve(u_1.p)$ or $starve(u_2.p)$ symbol, and the latter by asking if there are any legal behavior sequences containing 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, the terminal alphabet might be defined to be $\{starve(u_i.p), ur_i, rec(u_i.v, s.v, ok)\}_{i=1,2}$. Of course, the terminal alphabet could be any other subset of the augmented alphabet containing these six symbols.

Symbol	Associated event
$def(q, m)$	A message of type m is deposited in the buffer of process q.
$use(q, m)$	The buffer of process q is used when its value is of type m.
$send(l, m)$	A message of type m is sent to link l.
$rec(l, p, m)$	A message of type m is transmitted from link l through port p.
ur_i	The resource is used by process u_i for $i = 1, 2$.
$stop(q)$	Process q terminates normally.
$starve(p)$	The process starves at a **receive** p statement.
$ne(q)$	This nonevent symbol is used for process q.

Fig. 2. SDYMOL event symbols. For a given SDYMOL system, q ranges over all processes, l ranges over all links (outbound ports), p ranges over all inbound ports, and m ranges over all message types and over the reserved symbol $\perp$, which represents an undefined message.

3.3 The SDYMOL System Expression

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

$$\epsilon = \iota\left(\bigotimes_q \pi_q\right).$$

The initial expression describes the initial status of all links and buffers. Each process expression represents the sequential activity of a process.

The initial expression is a concatenation of symbols. It contains a $def(q, \perp)$ symbol for each process q indicating that the contents of all buffers are initially undefined and a $send(l, m)$ symbol for each message of type m that initially resides in a link l.

Each process expression is obtained from the SDYMOL code for a process through the statement-by-statement application of the translation rules shown in Figure 3. The application of the rules T_1–T_4 involves replacing the statement on the left by the sequence of symbols on the right.[6] The remaining rules are applied recursively. For instance, if the statement

while internal test do set buffer := m

appears in a process q, it is transformed into $def(q, m)^*$ by the application of T_3 and T_6.

The sequence of events that occurs when a process q executes a **send** l statement depends on the contents of its buffer. If the buffer contains a message, that message is placed in l. Hence, we have the disjunction over all defined message types on the right in rule T_1. Otherwise, the buffer is undefined and the **send** is a null operation. This possibility is represented by the final disjunct.

With the exception of T_2 and T_5, the rest of the translation rules can be similarly interpreted. The $starve(p)ne(q)$ alternative produced by a **receive** p

[6] In T_4 *identifier symbol* denotes the event symbol associated with the activity modeled by the statement consisting of the single identifier *identifier*. Thus, the symbol ur_i is associated with the statement *use resource$_i$* in the example.

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

Fig. 3. SDYMOL translation rules. Here q denotes the process defined by the SDYMOL program, curly brackets ($\{\}$) signify the recursive application of these rules, l ranges over all links connected to p, and m and n range over all defined message types and over $\perp$.

statement within a process q represents a request for a message through port p that is never fulfilled. In this case, q does not participate in future system events. Hence, if a $starve(p)$ symbol appears in a constrained prefix, then no symbols associated with q may appear in the prefix anywhere to the right of the $starve(p)$ symbol. We guarantee this by placing the nonevent symbol $ne(q)$ immediately after the $starve(p)$ symbol in T_2 and generating a constraint (described below) to filter out prefixes containing any $ne(q)$ symbols.

The $ne(q)$ symbol is used in a similar fashion in T_5. The Kleene star on the right in T_5 represents the repeated execution of the embedded statement $\langle s\rangle$. By itself, however, the Kleene star only specifies that the statement is executed some finite number of times. The $ne(q)$ symbol is used to guarantee that no subsequent statements in the code of the process are ever reached.

The system expression derived from the SDYMOL design in Figure 1 is shown in Figure 4. Many prefixes of this system expression do not represent legal behavior sequences. Consider, for example, the prefix

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

This string represents an event sequence in which an *ok* message is received from the link *s.p* before any messages have been deposited in the link. Several constraints in the SDYMOL constrained expression filter out this prefix.

System expression:

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

Initial expression:

$$\iota = def(s, \bot)def(u_1, \bot)def(u_2, \bot)$$

Process expressions:

$$\begin{aligned}\pi_s = {} & def(s, ok)((use(s, ok)send(s.p, ok) \vee use(s, \bot)) \\ & (rec(u_1.v, s.v, ok)def(s, ok) \vee rec(u_2.v, s.v, ok)def(s, ok) \\ & \vee starve(s.v)ne(s)))^* ne(s)stop(s) \\ \pi_{u_i} = {} & ((rec(s.p, u_i.p, ok)def(u_i, ok) \vee starve(u_i.p)ne(u_i)) \\ & ur_i(use(u_i, ok)send(u_i.v, ok) \vee use(u_i, \bot)))^* stop(u_i)\end{aligned}$$

Fig. 4. System expression for the SDYMOL design in Figure 1.

3.4 The SDYMOL Constraints

Constraints are required in an SDYMOL constrained expression because the translation rules of Figure 3 do not completely capture the semantics of SDYMOL. Each constraint describes the patterns of event symbols from the associated constraint alphabet that can appear in strings representing legal system behaviors. We use five different types of constraints for this purpose. The constraint templates that produce the constraints are shown in Figure 5. With the exception of the constraint $\kappa_3(q)$ for a process q, the constraint alphabet for a constraint consists of the symbols appearing in the constraint. The constraint alphabet for $\kappa_3(q)$ is the set $\{ne(q)\}$.

The first type of constraint, κ_1, pertains to the use and modification of buffer contents. See Figure 2, in which the symbol $def(q, m)$ is identified with placing a message of type m into the buffer of process q and the symbol $use(q, m)$ is identified with a use of the buffer when it contains a value of message type m. The undefined message, $\bot$, represents a buffer whose contents are currently undefined. The constraint $\kappa_1(q)$ requires that a use of the buffer of q when it contains a message of type m be preceded by placement of a message of type m into the buffer and that there be no intervening modifications to the buffer. The constraint does not require that any use of the buffer actually occur between successive modifications. Thus $\kappa_1(q)$ assures that the buffer of q is used and modified in a consistent fashion.

The second type of SDYMOL constraint, κ_2, relates to system termination. All behaviors described by the constrained expression must be complete behaviors. Each process must either terminate or become permanently blocked waiting to receive a message through some port.[7] This is assured for each process q by the constraint $\kappa_2(q)$.

The third type of constraint, κ_3, guarantees that nonevent symbols do not appear in strings representing behaviors. As explained above, this eliminates certain illegal patterns of event occurrences.

The fourth type of SDYMOL constraint, κ_4, relates to the transmission of messages. To understand this constraint, note that $(ab)\dagger$ represents the set of all strings containing equal numbers of a's and b's such that any prefix contains at

[7] A process that executes a **do forever** statement must eventually starve.

$$\kappa_1(q) = \left(\bigvee_{m'} def(q, m')use(q, m')^*\right)^*$$

$$\kappa_2(q) = \left(\bigvee_{p'} starve(p')\right) \vee stop(q)$$

$$\kappa_3(q) = \lambda$$

$$\kappa_4(l, m) = send(l, m)^* \otimes \left(send(l, m)\left(\bigvee_{p''} rec(l, p'', m)\right)\right)^\dagger$$

$$\kappa_5(l, p, m) = \left(send(l, m)\left(\bigvee_{p''} rec(l, p'', m)\right)\right)^\dagger starve(p)\left(send(l, m)\left(\bigvee_{p''\neq p} rec(l, p'', m)\right)\right)^\dagger$$
$$\vee \left(send(l, m) \vee \left(\bigvee_{p''} rec(l, p'', m)\right)\right)^*$$

Fig. 5. SDYMOL constraint templates. These templates are instantiated for each process q, link l, (inbound) port p, and message type $m \neq \perp$ of the given system. In the disjunctions, m' ranges over all message types (including $\perp$), p' ranges over all (inbound) ports of q, and p'' ranges over all (inbound) ports connected to l.

least as many a's as b's. The "dagger subexpression" in $\kappa_4(l, m)$ thus guarantees that each reception of a message of type m from a link l is preceded by a corresponding placement of a message of type m into l. The interleaved $send(l, m)^*$ allows more messages of type m to be placed in l than are ever received.

The final type of constraint, κ_5, pertains to process starvation. A process cannot starve waiting for a communication through a port p that is connected to a link l if there are messages residing in l. For a link l, an (inbound) port p connected to l, and a message type $m \neq \perp$, the constraint $\kappa_5(l, p, m)$ is used in stating this requirement, as follows.

The first disjunct of $\kappa_5(l, p, m)$ guarantees that if a process starves at a **receive** p statement, then there are no messages of type m that it can receive from l. The first "dagger subexpression" of this disjunct assures that there are no messages of type m in l when an attempt to fulfill the request is made. The second "dagger subexpression" assures that all messages of type m that are subsequently placed in l are used to service other receive requests.

The second disjunct of $\kappa_5(l, p, m)$ is required in case the process does not starve at a **receive** p statement. In this case $\kappa_5(l, p, m)$ does not impose any restrictions on the order or number of symbols representing events in which messages of type m are placed in l or received from l.

Taken together, the constraints $\kappa_5(l, p, m)$, for all defined message types m, assure that if a process starves at a **receive** p statement, then there are no messages of any type that it can receive from l. If the process does not starve at a **receive** p statement, then none of these constraints restricts the order or number of messages that are sent to or received from l. Of course, the constraints $\kappa_4(l, m)$ still restrict the order and number of messages sent to and received from l in this case.

These five 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. Application of the translation rules and instantiation of these constraint templates completes the derivation of an SDYMOL constrained expression.

4. CONSTRAINED EXPRESSIONS FOR CSP

4.1 A CSP System

We illustrate the derivation of CSP constrained expressions using the CSP solution to the mutual exclusion problem shown in Figure 6. It is based on the integer semaphore example presented by Hoare [18]. The line numbers in this figure are used for reference below. We discuss only aspects of CSP relevant to this example. A more complete description of CSP can be found in Hoare's original paper [18].

The system $\mathscr{BS}$, contains a *semaphore process*, S, and two *user processes*, U_1 and U_2. The *terminator processes* T_1 and T_2 cause the user processes to (eventually) terminate.

S begins by assigning a 1 to its local variable, VAL. It then executes the repetitive command in lines S2–5. On each cycle of this command, S waits until one of the user processes sends it a v-signal (S2–3) or until VAL is greater than 0 and one of the user processes sends it a p-signal (S4–5). Depending on which of the *successful* guards is arbitrarily selected for execution, it then either increments VAL or decrements VAL. A repetitive command terminates when all of its guards *fail* and all processes named in input commands of *open* guards have terminated. (A guard is open if the boolean expressions in the guard list are satisfied.) Hence, the repetitive command in S terminates when both U_1 and U_2 have terminated. S starves if the guards in lines (S2–5) fail and one of U_1 or U_2 never terminates. If S starves and VAL is not positive, then S starves waiting for a v-signal. If S starves and VAL is positive, then S starves waiting for either a v-signal or a p-signal.

The user process U_i, for $i = 1, 2$, initializes $CONT_i$ and then cycles, sending a p-signal, using the resource, and sending a v-signal, until it receives an e-signal from T_i (in U2); or it starves at the $S!p(\,)$ command or the $S!v(\,)$ command (in U3). If U_i receives an e-signal, the second guard never again succeeds. Hence, U_i waits for T_i to terminate, at which point it also terminates. If T_i never terminates, U_i starves waiting for (another) e-signal.

The process T_i, for $i = 1, 2$, waits for U_i to request its e-signal and then terminates. If U_i never requests an e-signal from T_i, then T_i starves.

4.2 The CSP Constrained Expression

In this section we show how to derive a constrained expression representation for $\mathscr{BS}$ from the CSP program in Figure 6. This example illustrates two important aspects of CSP constrained expressions. It shows how constrained expressions can be used to represent the semantics of CSP (synchronized) communication primitives. It also shows how the semantics of CSP-guarded commands and repetitive commands can be expressed. It does not, however, show how to express the full semantics of expression evaluation and of the CSP assignment command. We indicate how we handle these after discussing the example.

```
𝒮𝒮 = [S :: SEM || U₁ :: USER₁ || U₂ :: USER₂ || T₁ :: TERMINATOR₁ || T₂ :: TERMINATOR₂]

SEM ≡ (S1)VAL integer; VAL := 1;
        (S2)*[U₁?v() → VAL := VAL + 1           —U₁ releases resource
        (S3)▯U₂?v() → VAL := VAL + 1            —U₂ releases resource
        (S4)▯VAL > 0; U₁?p() → VAL := VAL − 1 —grant resource to U₁
        (S5)▯VAL > 0; U₂?p() → VAL := VAL − 1]—grant resource to U₂

USERᵢ ≡(U1)CONTᵢ boolean; CONTᵢ := t;—initialize CONTᵢ
        (U2)*[Tᵢ?e() → CONTᵢ := f        —get termination signal
        (U3)▯CONTᵢ → S!p(); urᵢ; S!v()] —else request, use
                                          and release resource

TERMINATORᵢ ≡ Uᵢ!e()—send termination signal
```

Fig. 6. CSP solution to the mutual exclusion problem.

Symbol	Associated event
def(X, a)	Value a is assigned to variable X.
use(X, a)	Variable X is used when it has the value a.
rec(P, Q, c)	c-signal is sent from process P and received by process Q.
ur_i	The resource is used by process U_i ($i = 1, 2$).
starve(P)	Process P starves.
stop(P)	Process P terminates.
send(P, Q, c)	Process P is ready to send a c-signal to process Q.
send'(P, Q, c)	Process P is 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 request a c-signal from P.
wait(Q, P, Q, c)	Process Q starves waiting for process P 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)	This is the nonevent symbol for process P.

Fig. 7. CSP event symbols.

The event symbols for this example are shown in Figure 7. Event symbols represent the use and modification of variables, interprocess communications (i.e., signals between processes), the use of the resource, and the starvation and termination of processes.

The other event symbols are used to model aspects of the CSP semantics. Let $\alpha = (P, Q, c)$ represent a c-signal from a *source process* P to a *target process* Q. The *rec*(α), *send*(α), and *send'*(α) symbols are used in constraints that assure communicating processes are properly synchronized. The *rec*(α) symbol represents the communication event. A *send*(α) symbol signifies that the source process is ready to communicate, while a *send'*(α) symbol signifies that the communication has taken place and the source is ready to continue with its processing. A *wait*(P', α) symbol encodes information about the starvation event *starve*(P'), where P' is either the source or target process of α. It signifies that P' starves waiting for the communication α. These symbols are used to guarantee that a process is not represented as starving on a communication that can take place. The *hang*(P) symbol indicates that the process P is assumed to starve (e.g., an iterative command in some other process should not terminate if P is named by the input command associated with an open guard and P does not terminate), and the *term*(P) symbol indicates that P is assumed to terminate. As in SDYMOL, the nonevent symbol *ne*(P) is used to eliminate certain patterns of impossible event occurrences.

4.3 The CSP System Expression

The system expression derived from Figure 6 is the interleave of five process expressions:

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

It is shown in Figure 8. We give line numbers in this figure for reference in the discussion below.

The initial *def*$(VAL, 1)$ symbol in line 1 of π_S is produced by the assignment command in S. The "star subexpression" and the *term*(U_1) and *term*(U_2) symbols (in line 10) are produced by the repetitive command. Each guarded command generates a disjunct (lines 2–5) representing its execution. The disjuncts in lines 6–9 represent the starvation of S within the repetitive command. Two disjuncts are required because there are two possible truth values for the guard lists preceding the input commands. If VAL is not positive when S starves, then S starves waiting for v-signals from the user processes. This explains the *wait*(S, U_i, S, v) symbols for $i = 1, 2$ in line 6 of π_S. The disjunction of the *hang*(U_1) and *hang*(U_2) symbols in line 7 indicates that either U_1 or U_2 must also starve in this case. The *starve*(S) symbol signifies the starvation of S, while the *ne*(S) symbol signifies that no further symbols from π_S can appear. The order of the *wait* and *hang* symbols is, of course, arbitrary. The interpretation and generation of the disjunct in lines 8 and 9 is similar.

Both U_1 and U_2 must terminate for the repetitive command to terminate. The "star subexpression" is thus followed by *term*(U_i) symbols for $i = 1, 2$. The *stop*(S) symbol represents the termination of S.

The translation of the user process U_i for $i = 1, 2$ requires the translation of two output commands. Each output command produces two disjuncts. One disjunct represents the communication taking place. It consists of the appropriate *send* and *send'* symbols. The second disjunct represents the starvation of U_i at the output command. Since the output command is not part of a guard, starvation at the command does not imply starvation of any other process. Thus, this disjunct does not require a *hang*(S) symbol. Similarly, no *hang* symbols are generated by an input command that is not part of a guard. The interpretation and generation of the process expression π_{T_i} for $i = 1, 2$ is obvious.

4.4 The CSP Constraints

Seven types of constraints are required in the CSP constrained expression. The templates for generating the constraints are shown in Figure 9. For this figure and the discussion below, we take V to be the set of variables in the CSP system and $t(X)$ to be the type of $X \in V$. We let C denote the set of communications

System Expression:

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

Process Expressions:

$\pi_S =$

(1) $\mathit{def}(VAL, 1)\Bigg($

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

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

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

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

(6) $\quad \vee \Big(\bigvee_{j \le 0} \mathit{use}(VAL, j)\Big)\mathit{wait}(S, U_1, S, v)\mathit{wait}(S, U_2, S, v)$

(7) $\quad\quad (\mathit{hang}(U_1) \vee \mathit{hang}(U_2))\mathit{starve}(S)\mathit{ne}(S)$

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

(9) $\quad\quad \mathit{wait}(S, U_2, S, p)(\mathit{hang}(U_1) \vee \mathit{hang}(U_2))\mathit{starve}(S)\mathit{ne}(S)$

(10) $\quad \Bigg)^{*} \mathit{term}(U_1)\mathit{term}(U_2)\mathit{stop}(S)$

$\pi_{U_i} =$

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

(2) $\quad \mathit{rec}(T_i, U_i, e)\mathit{def}(CONT_i, f)$

(3) $\quad \vee\ \mathit{use}(CONT_i, t)$

(4) $\quad\quad (\mathit{send}(U_i, S, p)\mathit{send}'(U_i, S, p) \vee \mathit{wait}(U_i, U_i, S, p)\mathit{starve}(U_i)\mathit{ne}(U_i))\mathit{ur}_i$

(5) $\quad\quad (\mathit{send}(U_i, S, v)\mathit{send}'(U_i, S, v) \vee \mathit{wait}(U_i, U_i, S, v)\mathit{starve}(U_i)\mathit{ne}(U_i))$

(6) $\quad \vee\ \mathit{use}(CONT_i, f)\mathit{wait}(U_i, T_i, U_i, e)\mathit{hang}(T_i)\mathit{starve}(U_i)\mathit{ne}(U_i)$

(7) $\quad)^{*}\mathit{use}(CONT_i, f)\mathit{term}(T_i)\mathit{stop}(U_i)$

$\pi_{T_i} =$

(1) $(\mathit{send}(T_i, U_i, e)\mathit{send}'(T_i, U_i, e) \vee \mathit{wait}(T_i, T_i, U_i, e)\mathit{starve}(T_i)\mathit{ne}(T_i))$

(2) $\quad \mathit{stop}(T_i)$

Fig. 8. System expression for the CSP program in Figure 6.

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

$$\kappa_2(P) = stop(P) \vee starve(P)$$

$$\kappa_3(P) = \lambda$$

$$\kappa_4(\alpha) = (send(\alpha)rec(\alpha)send'(\alpha))^*$$

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

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

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

Fig. 9. The CSP constraint templates.

that can take place. For a communication $\alpha = (P, Q, c) \in C$, $source(\alpha) = P$ is the source process, $target(\alpha) = Q$ is the target process, and c is the *constructor*. With the exception of the constraint $\kappa_3(P)$ for a process P, the constraint alphabet for a constraint is the set of symbols that appear in the constraint. The constraint alphabet for $\kappa_3(P)$ is the set $\{ne(P)\}$.

The first three types of CSP constraints, κ_1, κ_2, and κ_3, are analogous to the corresponding SDYMOL constraints.

The fourth type of CSP constraint, κ_4, relates to interprocess communication. An input command in a process P *corresponds* to an output command in a process Q if the input command names Q as the source, the output command names P as the destination, and both commands name the same constructor. In CSP, corresponding input and output commands are executed simultaneously. For a communication $\alpha \in C$, the constraint $\kappa_4(\alpha)$ assures that the source process and target process are at corresponding commands when the communication takes place. Let $\alpha = (P, Q, c)$. The $send(\alpha)$ and $send'(\alpha)$ symbols produced by a $Q!c(\,)$ command are used as markers in the process expression π_P. The $send(\alpha)$ symbol follows all symbols representing events that precede execution of the output command, while the $send'(\alpha)$ symbol precedes all symbols representing events that follow execution of the output command. The $rec(\alpha)$ symbol produced by a $P?c(\,)$ command is used in a similar fashion in the process expression π_Q. It follows all symbols representing events that precede execution of the input command and precedes all symbols representing events that follow execution of the input command. The $rec(\alpha)$ symbol also represents the simultaneous execution of the corresponding commands. The constraint $\kappa_4(\alpha)$ requires that each $rec(\alpha)$ symbol lie between corresponding $send(\alpha)$ and $send'(\alpha)$ symbols.[8]

The fifth type of CSP constraint, κ_5, is analogous to the fifth type of SDYMOL constraint. The constraint $\kappa_5(\alpha)$, for $\alpha \in C$, filters out strings representing event sequences in which both $target(\alpha)$ and $source(\alpha)$ starve at corresponding commands. Hence, these constraints assure that a process does not starve if it can communicate.

[8] We could equally well have reversed the role of the *send* and *rec* symbols, employing $send(\alpha)$, $rec(\alpha)$, and $rec'(\alpha)$ in the translation rules for $Q!c(\,)$ and $P?c(\,)$ and modifying κ_4 to be $(rec(\alpha)send(\alpha)rec'(\alpha))^*$.

The sixth type of CSP constraint, κ_6, also relates to the starvation of processes. If a string from the language of a process expression π_Q contains a *starve*(Q) symbol that is produced by the translation of a guard in a repetitive command, then it also contains a *hang*(P) symbol, where P is one of the processes from which Q is waiting for a communication. The *hang* symbols are generated because one of these processes must starve in order for Q to starve at the repetitive command. (Otherwise, the repetitive command terminates.) In this case, the constraint $\kappa_6(P)$ assures that the string also contains a *starve*(P) symbol.

The final type of CSP constraint, κ_7, assures that a repetitive command is executed until all processes that can make a guard succeed have terminated. The representation of the execution of a repetitive command is followed by a sequence of *term*(P) symbols. One *term*(P) symbol is generated for each process P that must terminate for the repetitive command to terminate. The constraint $\kappa_7(P)$ assures that a *term*(P) symbol does not appear in a constrained prefix unless it is preceded by a *stop*(P) symbol.

This completes our treatment of the constrained expression representation of the system of Figure 6. We now discuss some aspects of the derivation of constrained expression representations of CSP systems that are not illustrated in that example.

In the CSP system of Figure 6, interprocess communication is limited to the exchange of timing signals. More generally, a CSP output command may specify values that are assigned to target variables in the target process. We represent the transfer of information between CSP processes in the same way that we represent the transfer of information between SDYMOL processes. (See Section 3.)

The evaluation of an expression and the execution of an assignment command can *fail* in CSP. The evaluation of an expression fails if any of the operations it requires are undefined. An assignment command fails if the structures of the target variables and assignment values do not *match*, as in [17] in which the notion of matching structures is made precise. Dillon's dissertation [10] shows how to represent the failure of SDYMOL expression evaluation. Essentially the same approach can be used to represent the failure of expression evaluation and of assignment commands in a CSP system, provided that certain restrictions are placed on the structure of its variables.

The derivation of a CSP system expression is clearly more complex than the derivation of an SDYMOL system expression. The additional complexity stems from the complex semantics of the CSP repetitive command. In SDYMOL, the condition for termination of a repetitive command is determined by local data (e.g., the contents of the process'es buffer). In CSP one must consider the status of all processes that could affect a guard in a repetitive command to determine if the command terminates or results in starvation. Moreover, the precise semantics of an input command in CSP depends on whether the command appears in a guard or a command list.

To represent the failure of a CSP repetitive command, we partition the domains of variables in the guard lists so that the truth values of the guard lists are constant on each element of the partition. A disjunct is then produced for each element to specify the processes that must starve if the repetitive command fails. The termination of a repetitive command is represented in a similar fashion.

Despite this additional complexity, the derivation of a CSP system expression is a straightforward compilation task. The generation of the CSP constraints is no more complex than the generation of the SDYMOL constraints. Additionally, the CSP constraints are all regular since they do not involve the dagger operator.

5. CONSTRAINED EXPRESSIONS FOR PETRI NETS

5.1 A Petri Net System

The Petri net formalism is a classical model of concurrent systems. A Petri net is represented graphically as a collection of nodes, called *places* and *transitions*, that are connected by directed arcs [26]. A *marking* specifies the *tokens* that reside in the places of the net. Tokens enable the *firing* of transitions. When a transition fires, it produces certain token movements. Transitions are labeled with symbols from an event alphabet, so that each firing sequence determines a string of event symbols. The Petri net language is the set of strings determined by all firing sequences [16]. Alternatively, the Petri net language may be defined as the set of strings determined by those firing sequences that take the net from its *initial* marking to a specified *final* marking [16, 25].

Figure 10 depicts a labeled Petri net that is based on the solution to the mutual exclusion problem presented in [26]. The net has seven places, P_1 through P_7, and six transitions, t_1 through t_6. We use circles to represent places and bars to represent transitions in this figure. Each transition is labeled with a symbol from the alphabet $\{p_1, p_2, v_1, v_2, ur_1, ur_2\}$. A dot in a place represents a token.

The arcs in a Petri net determine the rules for firing transitions. A transition t_m is *fireable* if there are at least as many tokens in each place P_i as there are arcs from P_i to t_m. The firing of t_m removes a token from each P_i for every such arc and adds a token to each of the places P_j for every arc from t_m to P_j. For example, the transition t_1 in Figure 10 is fireable. The firing of t_1 removes a token from P_1 and from P_4 and adds a token to P_2. A firing sequence determines the string obtained by replacing transitions with their transition labels. The firing sequence $t_1 t_2 t_3$ of the net in Figure 10, for instance, determines the string $p_1 ur_1 v_1$.

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

5.2 The Petri Net Constrained Expression

To express the rules governing the firing of transitions, we use symbols representing the addition of tokens to places and the removal of tokens from places. In general, the augmented alphabet consists of the transition labels, the symbols put_i and $take_i$ for each place P_i in a Petri net, and the special symbol *end*. A put_i symbol signifies the addition of a token to P_i, and a $take_i$ symbol signifies the removal of a token from P_i. The symbol *end* is explained below. Naturally, the terminal alphabet is just the set of transition labels.

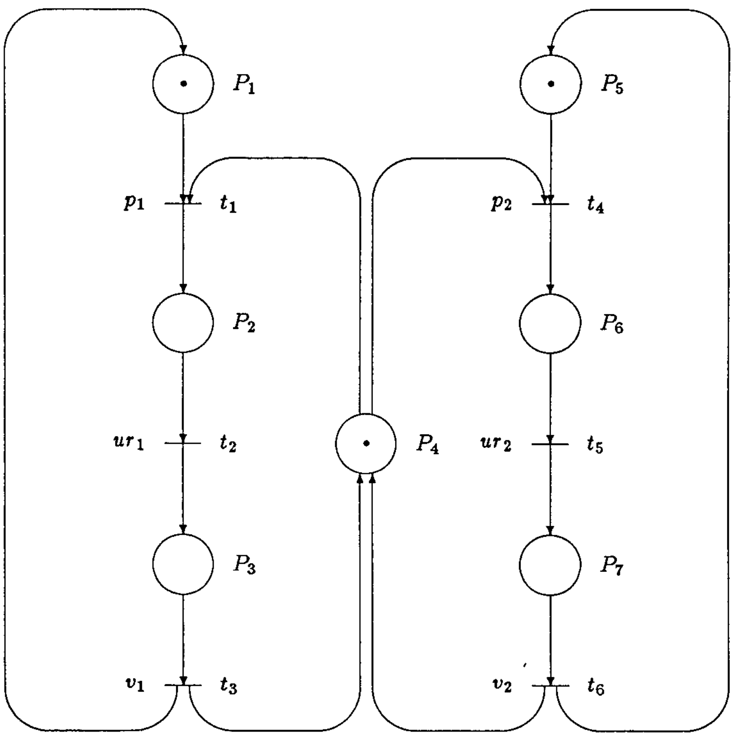


Fig. 10. Petri net solution to the mutual exclusion problem.

5.3 The Petri Net System Expression

The Petri net system expression ϵ has the form,

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

where m ranges over the indices of the transitions and *end* is a special symbol in the augmented alphabet. The *initial expression* ι indicates the number of tokens and their locations in the initial marking of the net. It is a concatenation of put_i symbols, one for each token residing in a place P_i in the initial marking.

The *transition expression* π_m associated with a transition t_m is the concatenation of (1) a sequence of $take_i$ symbols, one for each arc from a place P_i to t_m; (2) the label associated with the transition; and (3) 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. The system expression derived from the Petri net in Figure 10 is shown in Figure 11.

Many prefixes of the Petri net system expression represent firing sequences that can never occur because transitions are shown as firing when they are not fireable. Moreover, a proper prefix of the Petri net system expression can end in the "middle" of a transition. That is, it can end with a proper prefix of a string from the language of a transition expression. Such a prefix does not encode the

System expression:

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

Initial expression:

$$\iota = put_1 put_4 put_5$$

Transition expressions:

$$\pi_1 = take_1 take_4 p_1 put_2$$

$$\pi_2 = take_2 ur_1 put_3$$

$$\pi_3 = take_3 v_1 put_1 put_4$$

$$\pi_4 = take_4 take_5 p_2 put_6$$

$$\pi_5 = take_6 ur_2 put_7$$

$$\pi_6 = take_7 v_2 put_4 put_5$$

Fig. 11. System expression derived from the Petri net in Figure 10.

$$\kappa_1 = end$$

$$\kappa_2(P_i) = \begin{cases} (put_i take_i)\dagger \otimes (put_i)^* & \text{if no final marking is specified} \\ (put_i take_i)\dagger \otimes \underbrace{(put_i \cdots put_i)}_{n_i \text{ times}} & \text{if the final marking has } n_i \text{ tokens in } P_i \end{cases}$$

Fig. 12. The Petri net constraint templates.

full semantics of the firing of the transition, since all the necessary token movements have not taken place. The Petri net constraints filter out proper prefixes of the system expression, along with strings that do not represent legitimate firing sequences.

5.4 The Petri Net Constraints

We use two types of Petri net constraints. They are shown in Figure 12. For each place P_i, the integer $n_i \geq 0$ denotes the number of tokens residing in P_i in the final marking for the net. The alphabet for a constraint is the set of symbols appearing in the constraint.

The first type of Petri net constraint consists of a single constraint, κ_1. This constraint filters out proper prefixes of the system expression. For example, the string $put_1 put_4 put_5 take_4 take_5 p_2$ is filtered out by this constraint. It represents an event sequence in which the transition labeled p_2 has fired, but a token has not yet been deposited in P_6.

The second type of Petri net constraint, κ_2, assures that the rules governing the firing of transitions are followed and that the required number of tokens reside in each place at the end of a firing sequence. We use the first template if no final marking is specified. Otherwise, we use the second. The "dagger subexpression" in the constraint $\kappa_2(P_i)$, for a place P_i, guarantees that a token is available in P_i whenever one is removed. Hence, constrained prefixes correspond to legal firing sequences. The n_i interleaved put_i symbols in the second template for κ_2 correspond to the tokens left in P_i at the end of the firing sequence.

6. ISSUES OF CORRECTNESS

Naturally, the "correctness" of the derivation procedure associated with a development notation must be considered. Using the terminology of [31], this involves showing that the constrained expression representations produced by the derivation procedure are both *accurate* and *precise*.[9] A constrained expression representation D of a system S is *accurate* if every event string representing a behavior of S is contained in the interpreted language of D. It is *precise* if every string in the interpreted language represents a legal behavior.

Accuracy of a constrained expression representation is essential. Analysis based on a constrained expression that is not an accurate representation of a system could miss potential errors. Precision in a constrained expression representation is equally important. Analysis based on a constrained expression that is not a precise representation of a system could report "errors" that cannot occur. It could also certify that certain (desirable) events occur when they do not. Analysis of a representation that is both accurate and precise will not report any spurious errors, nor inspire false confidence.

The verification of the accuracy and precision of a derivation procedure requires an appropriate formal definition of the associated development notation. This verification is facilitated if the formal description includes a simple criterion for determining whether a particular sequence of events can occur in a behavior of a system described in the development notation. Such a criterion exists for the Petri net formalism. We demonstrate how this criterion can be used to verify the derivation procedure described in Section 5 above.

Consider the constrained expression representation derived for a Petri net according to the translation rules and constraint templates of the previous section. To simplify the discussion, we assume a Petri net in which each transition t_i is associated with a distinct label s_i, and we let the terminal alphabet of the constrained expression consist of the transition symbols s_i. We now show that the constrained expression representation is both accurate and precise.

Assume first that no final marking is specified for the Petri net. Then a sequence $t_{i_1} \cdots t_{i_n}$ of transitions is a possible firing sequence if and only if $t_{i_1} \cdots t_{i_{n-1}}$ is a possible firing sequence and, after it has fired, the number of tokens in P_j, for each place P_j, is greater than or equal to the number of arcs from P_j to t_{i_n}. The empty firing sequence is, of course, always possible.

PROPOSITION 1. *Suppose that $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence. Let $|symb|_\sigma$ denote the number of occurrences of the symbol symb in the string σ. Then we have*

(i) The string $s_{i_1} \cdots s_{i_n}$ is in the interpreted language of the constrained expression, and

(ii) For each place P_j, the number of tokens in P_j after the firing sequence

[9] This terminology is not standard. We use the terms "accurate" and "precise" instead of "consistent" and "complete" because these latter terms are overloaded in the literature.

$t_{i_1} \cdots t_{i_n}$ is equal to

$$|put_j|_\iota - \sum_{k=1}^{n} |take_j|_{\pi_{i_k}} + \sum_{k=1}^{n} |put_j|_{\pi_{i_k}}.$$

PROOF. We proceed by induction on n. The result obviously holds when $n = 0$.

Suppose $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence with $n > 0$. The induction hypothesis implies that $s_{i_1} \cdots s_{i_{n-1}}$ occurs in the interpreted language, so $\iota\pi_{i_1} \cdots \pi_{i_{n-1}}$ *end* is a constrained prefix, and that the number of tokens in P_j after $t_{i_1} \cdots t_{i_{n-1}}$ is

$$|put_j|_\iota - \sum_{k=1}^{n-1} |take_j|_{\pi_{i_k}} + \sum_{k=1}^{n-1} |put_j|_{\pi_{i_k}}.$$

Since $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence, the number of tokens in a place P_j after $t_{i_1} \cdots t_{i_{n-1}}$ is greater than or equal to the number of arcs from P_j to t_{i_n}. We see from the translation rules that $|take_j|_{\pi_{i_n}}$ is equal to the number of these arcs. Together with the fact that $\iota\pi_{i_1} \cdots \pi_{i_{n-1}}$ *end* is a constrained prefix, this implies that $\iota\pi_{i_1} \cdots \pi_{i_n}$ *end* is a constrained prefix and $s_{i_1} \cdots s_{i_n}$ is in the interpreted language. So (*i*) is proved.

To prove (*ii*), we note that the number of tokens in a place P_j after $t_{i_1} \cdots t_{i_n}$ is equal to the number of tokens in P_j after $t_{i_1} \cdots t_{i_{n-1}}$ plus the number of arcs from t_{i_n} to P_j minus the number of arcs from P_j to t_{i_n}. The translation rules imply that the number of arcs from t_{i_n} to P_j is $|put_j|_{\pi_{i_n}}$ and the number of arcs from P_j to t_{i_n} is $|take_j|_{\pi_{i_n}}$. The desired result then follows from the induction hypothesis applied to the sequence $t_{i_1} \cdots t_{i_{n-1}}$. □

PROPOSITION 2. *Suppose that $s_{i_1} \cdots s_{i_n}$ is a string in the interpreted language. Then $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence.*

PROOF. Again we argue by induction on n, with the case $n = 0$ clear.

Suppose $s_{i_1} \cdots s_{i_n}$ in a string in the interpreted language for some $n > 0$. Then $\iota\pi_{i_1} \cdots \pi_{i_n}$ *end* is a constrained prefix. Inspection of the constraints shows that $\iota\pi_{i_1} \cdots \pi_{i_{n-1}}$ *end* must also be a constrained prefix, so $s_{i_1} \cdots s_{i_{n-1}}$ is in the interpreted language. We conclude by induction that $t_{i_1} \cdots t_{i_{n-1}}$ is a possible firing sequence.

Since the string $\iota\pi_{i_1} \cdots \pi_{i_n}$ *end* satisfies all the constraints $\kappa_2(P_j)$, we must have for each j,

$$|take_j|_{\pi_{i_n}} \le |put_j|_\iota - \sum_{k=1}^{n-1} |take_j|_{\pi_{i_k}} + \sum_{k=1}^{n-1} |put_j|_{\pi_{i_k}}.$$

The left side of this inequality is the number of arcs from the place P_j to the transition t_{i_n}. By the previous proposition applied to $t_{i_1} \cdots t_{i_{n-1}}$, the right side is equal to the number of tokens in P_j after $t_{i_1} \cdots t_{i_{n-1}}$. Thus, $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence. □

These two propositions imply the following theorem.

THEOREM 1. *Let S be a Petri net with no final marking specified, and let D be the constrained expression representation of S obtained using the translation rules* *of Section 5. Assume that the alphabet of the interpreted language of D consists of the labels s_i corresponding to the transitions t_i of S, and that these labels are all distinct. Then $t_{i_1} \cdots t_{i_n}$ is a possible firing sequence of S if and only if $s_{i_1} \cdots s_{i_n}$ is in the interpreted language of D.*

The theorem says that D gives a precise and accurate representation of the finite behaviors of the Petri net S. The case in which a final marking for S is specified follows easily, since a firing sequence is possible for a net with final marking if and only if it is possible for the same net without specifying the final marking and it results in the specified final marking. The theorem above characterizes the possible firing sequences of the net without final marking, and an examination of the constraints $\kappa_2(P_j)$ shows that they enforce the requirement of the specified final marking. Clearly, the case in which not all transitions are labeled and/or the transition labels are not all distinct also follows easily from the theorem.

In principle, a similar demonstration could be given for any development language with a suitably formal definition. At the current stage in our work, however, we are content with less formal determinations of accuracy and precision. We have informally checked the accuracy and precision of our constrained expression derivation procedures, including those for SDYMOL and CSP, by justifying each translation rule and constraint template on the basis of interpretation of the event symbols. Typically, a first version of a derivation procedure is devised through a combination of informal reasoning and experimentation. The derivation procedures associated with other development notations can serve as a model for this first version. The proposed derivation procedure is then "tested" and revised until a satisfactory procedure is obtained. Of course, given a formal description of the development language, a more formal approach to defining derivation procedures would be conceivable.

We note that the problems of precision and accuracy are not restricted to showing that constrained expression representations properly reflect the formal semantics of a development notation. Proving that a derivation procedure accurately and precisely reflects a formal description of a development notation is of little value if the formal description does not accurately and precisely correspond to the practical implementations of that notation. The task of verifying this correspondence between the formal description and the implementation, whether it be an actual compiler or simply a software developer's understanding of a design language, is at least as difficult and important.

7. CONCLUSION

We have shown how constrained expressions are 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 for different stages of software development. In addition to the three notations discussed here, constrained expressions have been used with an Ada-based design language [5] and with a notation that provides primitives for dynamically altering the communication pathways in a distributed system [33]. This range of applications demonstrates the broad applicability of the constrained expression approach.

Further evidence comes from two other projects that are in progress at the University of Massachusetts. In one of these, an event-based language resembling constrained expressions is being used for high-level debugging of distributed systems [6]. A prototype toolset supporting this debugging method has been implemented and is being used in a distributed problem-solving testbed system. Another research group is using a similar event-based language in an intelligent user interface system that is part of an office automation project [9]. The language is used to describe office procedures. These descriptions are interpreted by the interface system, which assists users in carrying out the procedures.

The primary concern of our research is with building tools to help in the development of distributed software systems. It is in this context that the broad applicability of the constrained expression approach is significant. Constrained expressions provide a framework for formal analysis of properties, such as mutually exclusive use of shared resources and absence of deadlock, that can be characterized by the patterns of event symbols appearing in strings representing behaviors of a system. A constrained expression is a finite representation for a potentially infinite set of behaviors. It can therefore be used to reason about the sequences of events in all the behaviors of the set collectively, rather than one at a time. Since the constrained expression explicitly encodes relationships between the order and number of event occurrences in behaviors of a system, it directly supports reasoning about behavioral properties that involve interactions among the parts of a distributed system.

We have developed several techniques for analyzing constrained expressions. Detailed presentations of these techniques appear elsewhere [2, 4, 10, 12]. Their application in a realistic distributed software development setting is illustrated in [3].

We have begun development of a prototype toolset incorporating the constrained expression approach. The toolset is targeted for use with an Ada-like design language, called CEDL, that we have developed [11, 29]. The toolset contains a constrained expression *deriver*, a *simplifier*, a *behavior generator*, and a collection of *analysis tools*.

The deriver produces a constrained expression representation for a system from a CEDL design [30]. A first version of a deriver has been implemented in Ada [29]. The current deriver is tabledriven. Derivers for later versions of CEDL and for other software development notations can, therefore, be easily created by modifying the existing tool.

The derivation procedure associated with a distributed system development notation is general, so that is can be applied to any syntactically correct description. Specific features of a particular distributed system, however, often make it possible to produce a simpler constrained expression representation of the system than the one produced by the general derivation procedure. We have developed techniques for "simplifying" constrained expressions [10, 12]. The simplifier automates this process by accepting a constrained expression and producing a simplified version of that expression. Use of a constrained expression simplifier will reduce the effort required for later analyses. In constructing the simplifier, we are exploring relationships between constrained expression simplification and certain static analysis techniques [31].

The behavior generator is a tool for producing example behaviors from a constrained expression representation of a system. We have implemented a preliminary version of this tool in Prolog [1]. It generates a string from the interpreted language of a constrained expression. The existing behavior generator is interactive so that it can be guided in a search for a behavior possessing specific properties. This capability will be of great value to users of the constrained expression toolset. A developer of distributed software can explore properties of a system, detect certain classes of errors, and recognize possible improvements using the behavior generator. Moreover, the analysis tools produce vast amounts of information about behaviors that have a particular property. The behavior generator will help the developer transform this information into concrete examples of behaviors having the given property.

The analysis tools automate the algebraic analysis techniques of [2], [4], and [5]. These techniques can be used to determine whether a particular pattern of events appears in any behavior described by a given constrained expression. Their application involves the generation of a system of inequalities involving the number of occurrences of events in subsequences of the behavior. If this system of inequalities is inconsistent, the hypothesized pattern of events cannot occur. If it is consistent, the inequalities provide information about the behaviors in which the pattern might occur. This information can then be used to guide the production of such behaviors.

The process of generating inequalities is driven by the form of the constrained expression. Preliminary work on automating this process is very promising [2], and a prototype inequality generator based on these methods is currently under development. We are using a standard integer linear programming package to solve the systems of inequalities produced by the inequality generator. When combined with this package, the inequality generator will provide powerful support for analysis of distributed software systems.

Most of the above tools could be used with any notation a developer might adopt. An appropriate deriver would be required for producing constrained expression representations from descriptions in the chosen notation. This requirement is primarily a matter of modifying the tables in our current deriver to reflect the appropriate translation rules and constraint templates. Certain aspects of other tools will rely upon specific details of the form of CEDL constrained expressions. For the most part, however, the simplifier, behavior generator, and analysis tools could be readily adapted to support analysis of descriptions in other notations. This would permit developers to select notations based on their naturalness and expressive power, rather than on the availability of analysis tools.

APPENDIX. FORMAL DEFINITION OF CONSTRAINED EXPRESSIONS

We let $\mathscr{RE}(A)$ denote the set of regular expressions over the alphabet A, where regular expressions over an alphabet are formed from the symbols in the alphabet, the null string (represented by λ), and the empty set (represented by $\emptyset$) by finite applications of the usual regular expression operators, alternation (represented by $\vee$), concatenation (represented by juxtaposition), transitive closure (represented by $*$), and an additional operator, the shuffle or interleave operator (represented by $\otimes$). The operator $\otimes$ has been shown to preserve regularity [13] and is useful for representing concurrent activity.

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 $\dagger$). The $\dagger$ 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 $\dagger$ 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 through finite applications of the regular expression operators (including $\otimes$) and the $\dagger$ operator. We write $\mathscr{E}\mathscr{E}(A)$ 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

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

We think of ρ_S as "erasing" the symbols not belonging to S. Let u be a string of symbols from A. For $\kappa \in \mathscr{E}\mathscr{E}(S)$, 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 that satisfy all of the event expressions. More formally, we have

Definition 1. *A* constrained expression *is an ordered pair* $D = (F, \epsilon)$, *where*

(i) $F = (A, E, \hat{S}, \hat{C})$, *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 $\kappa_j \in \mathscr{E}\mathscr{E}(S_j)$, for each $j \in J$, and

(ii) $\epsilon \in \mathscr{R}\mathscr{E}(A)$.

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

REFERENCES

1. AVERY, S. M. Development of a behavior generator for constrained expressions. Tech. Rep. SDLM/84-2, Software Development Laboratory, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, June 1984.
2. AVRUNIN, G. S. Experiments in constrained expression analysis. Tech. Rep. 87-125, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, 1987.
3. AVRUNIN, G. S., AND WILEDEN, J. C. Algebraic techniques for the analysis of concurrent systems. In *Proceedings of the Sixteenth Annual Hawaii International Conference on System Science* (Jan. 1983). Western Periodicals, 1983, pp. 51–57.
4. AVRUNIN, G. S., AND WILEDEN, J. C. Describing and analyzing distributed system designs. *ACM Trans. Program. Lang. Syst. 7*, 3 (July 1985), 380–403.
5. AVRUNIN, G. S., DILLON, L. K., WILEDEN, J. C., AND RIDDLE, W. E. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng. SE-12*, 12 (Feb. 1986), 278–292.
6. BATES, P., AND WILEDEN, J. C. High level debugging of distributed systems. *J. Syst. Softw. 3*, 4 (Dec. 1983), 255–264.
7. CAMPBELL, R. H., AND HABERMANN, A. N. The specification of process synchronization by path expressions. In *Lecture Notes in Computer Science*, Springer-Verlag, Heidelberg, 1974, pp. 89–102.
8. CHEN, B. S., AND YEH, R. T. Formal specification and verification of distributed systems. *IEEE Trans. Softw. Eng. 6* (Nov. 1983), 710–722.
9. CROFT, W., AND LEFKOWITZ, L. Task support in an office system. *ACM Trans. Office Inf. Syst. 2*, 3 (July 1984), 197–212.
10. DILLON, L. K. Analysis of distributed systems using constrained expressions. Ph.D. dissertation, Univ. of Massachusetts, Amherst, Sept. 1984. Also available as TR 84-18.
11. DILLON, L. K. Overview of the constrained expression design language. Tech. Rep. TRCS86-21, Computer Science Dept., Univ. of California, Santa Barbara, Oct. 1986.
12. DILLON, L. K. Simplification and reduction of CEDL constrained expressions. Tech. Rep. TRCS86-29, Computer Science Dept., Univ. of California, Santa Barbara, Nov. 1986.
13. GINSBURG, S. *The Mathematical Theory of Context-Free Languages*. McGraw Hill, New York, 1966.
14. GREIF, I. A language for formal problem specification. *Commun. ACM 20*, 12 (Dec. 1977), 931–935.
15. HABERMANN, A. N. Synchronization of communicating processes. *Commun. ACM 25*, 3 (Mar. 1972), 171–176.
16. HACK, M. Petri net languages. Memo 124, Computation Structures Group, MIT, Cambridge, Mass., June 1975.
17. HOARE, C. A. R. Communicating sequential processes. *Commun. ACM 21*, 8 (Aug. 1978), 666–677.
18. HOARE, C. A. R. Specifications, programs and implementations. Tech. Mono. PRG-29, Oxford Univ. Computing Laboratory, Oxford, England, June 1982.
19. HOARE, C. A. R. *Communicating Sequential Processes*. Prentice-Hall International, Englewood Cliffs, N.J. 1985.
20. HOLZMANN, G. J. A theory of protocol validation. *IEEE Trans. Comput.* (Aug. 1982), 730–738.
21. LAMPORT, L. A new approach to proving the correctness of multiprocess programs. *ACM Trans. Program. Lang. Syst.* (July 1979), 84–97.
22. LAUER, P. E., TORRIGIANI, P. R., AND SHIELDS, M. W. Cosy: A system specification language based on paths and processes. *Acta Inf.* (1979), 451–503.

23. MISRA, J., AND CHANDY, K. M. Proofs of networks of processes. *IEEE Trans. Softw. Eng. SE-7*, 4 (July 1981), 417-426.
24. NGUYEN, V., DEMERS, A., GRIES, D., AND OWICKI, S. A model and temporal proof system for networks of processes. *Distributed Comput.* (Jan. 1986), 7-25.
25. PETERSON, J. Computation sequence sets. *J. Comput. Syst. Sci. 13*, 1 (Aug. 1976), 1-24.
26. PETERSON, J. Petri nets. *ACM Comput. Surv. 9*, 3 (Sept. 1977), 223-252.
27. RIDDLE, W. E. An approach to software system behavior modeling. *Comput. Lang.* (Elmsford, N.Y.) *4* (1979), 29-47.
28. SHAW, A. C. Software descriptions with flow expressions. *IEEE Trans. Softw. Eng. SE-4*, 3 (May 1978), 242-254.
29. SUNDARAM, U. A constrained expression deriver for CEDL: An overview. Tech. Rep. SDLM 86-1, Software Development Laboratory, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, Aug. 1986.
30. SUNDARAM, U., AVRUNIN, G. S., AND WILEDEN, J. C. Design of the deriver for CEDL. To be published.
31. TAYLOR, R. N. A general-purpose algorithm for analyzing concurrent programs. *Commun. ACM 6*, 5 (May 1983), 362-376.
32. WELTER, M. Counter expressions. Tech. Rep. RSSM/24, Dept. of Computer and Communication Science, Univ. of Michigan, Ann Arbor, Oct. 1976.
33. WILEDEN, J. C. Techniques for modelling parallel systems with dynamic structure. *J. Digital Syst.* (Summner 1980), 177-197.
34. WILEDEN, J. C. Constrained expressions and the analysis of designs for dynamically-structured distributed systems. In *Proceedings of the 1982 International Conference on Parallel Processing* (Bellaire, Mich., Aug. 1982). IEEE Computer Society Press, New York, pp. 340-344.

Received June 1986; revised April and December 1987; accepted December 1987