Toward Automating Analysis Support for Developers of Distributed Software*

Jack C. Wileden
Software Development Laboratory
Computer and Information Science Department
University of Massachusetts, Amherst

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

ABSTRACT

Developing large-scale, reliable software capable of exploiting the potential of distributed hardware systems will demand the support of powerful automated tools, especially analysis tools. Our *constrained expression* approach to analyzing such software systems seems to have several advantages, including broad applicability and reasonable efficiency, relative to other proposed approaches. Results of initial experiments with our approach, based on manual application and preliminary prototypes of some of the needed tools, have been encouraging. They have also demonstrated the need for us, and all researchers working on distributed software analysis, to undertake more extensive experimentation with larger, more realistic examples. In this paper, we report on our initial experimentation with constrained expression analysis and describe our plans for constructing the more robust, flexible and efficient prototype tool implementations needed to support more extensive experimentation.

Introduction

Hardware systems supporting distributed computing are constantly improving. Our ability to create software effectively exploiting the potential power of such systems has not kept pace. This is due in part to a lack of appropriate tools to aid developers of distributed software systems. Because of the inherent complexity of distributed software, constructing large-scale, reliable distributed software systems will be virtually impossible without powerful automated support tools.

We believe that tools for *analyzing* distributed software systems are especially important. In particular, there is a need for tools supporting analysis of the logical or behavioral aspects of distributed systems. These are tools that can help to uncover logical flaws or unintended properties in a system's behavior, such as deadlock, process starvation or synchronization anomalies. Ideally, such tools should be applicable not only to completed code but also to *pre-implementation descriptions* such as specifications and designs.

No existing tool effectively supports such analysis. Indeed, it seems unlikely that any single tool would ever fulfill all the analysis needs of distributed software developers. Rather, we expect that there will eventually be a collection of tools, each known to be of greatest value for a particular class of problems, or during a particular phase of software development. Together the tools in this collection will provide the necessary analysis capabilities. Assembling such a collection will require the development of a variety of analysis techniques, supported by theoretical studies of their capabilities and limitations. Also needed, however, will be experimental studies of the techniques, based on robust prototype implementations of the requisite tools, to establish the practical utility and shortcomings of the techniques.

In fact, several techniques have been proposed for analyzing distributed software systems at various stages in their development. All are known to have certain limitations. The practical significance of those limitations is not clear, however, since most of the proposed techniques have never been adequately tested through prototype implementation and experimentation. For similar reasons, the phases of development to which these techniques are best applied and the analysis questions that they are best suited for addressing are also not clearly established.

Over the last several years, we have been developing and experimenting with an approach for analyzing distributed software systems. While it too has some limitations, we believe that our *constrained expression* approach has several advantages relative to other approaches. In particular, the constrained expression approach appears to be:

- **Broadly Applicable:** Its use is not restricted to a limited set of design or programming languages nor to a limited range of analysis problems.
- **Relatively Efficient:** It offers a highly focused style of analysis that limits combinatorial explosion.
- **Straightforward to Implement:** Nothing more exotic than standard language processing and numerical analysis tools seems to be required.

To date we have demonstrated the approach's broad applicability by showing how it can be applied to systems described in a diverse set of design and programming notations, including CSP and Petri nets [14] and Ada [11]. We have also shown that it produces relatively efficient assessments of significant properties of distributed systems. We have, for example, used it in manually analyzing the dining philosophers problem, a distributed mutual exclusion mechanism and an automated gas station system. Finally, we have produced primitive prototype implementations of some of the central tools required to automate the constrained expression approach. These initial prototype implementations have demonstrated the feasibility of

*The work described here was supported in part by National Science Foundation grants DCR-84-08143 and DCR-85-00332 and by the Defense Advanced Research Projects Agency (ARPA Order No. 6104, ARPA Program Code No. 7E20) through National Science Foundation grant CCR-87-04478.

automating the approach using fairly straightforward technology.

Encouraged by these initial results, we now plan to continue development of the technique and to undertake a thorough assessment of its potential practical value. This will involve both continued theoretical work and serious experimentation with examples of realistic size and complexity. The experimentation with realistic examples will require tools that are more robust and extensible, are more efficient, and have better user interfaces than our current prototypes.

In the remainder of this paper, we report on some of our initial experimentation with the constrained expression approach and describe our plans for continued efforts toward automated analysis support for developers of distributed software. We begin with an overview of the current status of work on tools supporting analysis of distributed software systems. We then outline our constrained expression approach and describe how it can be used to analyze distributed software. We illustrate this by describing one of our recent experiments. This is followed by a description of the improved prototype toolset that we are currently building. We conclude with a summary of our plans for enhanced tools and further experimentation.

Tools for Analyzing Distributed Software

Tools for analyzing software may be broadly classified as *dynamic* or *static*. Tools for dynamic analysis are those that derive their information through some sort of animation of the system, such as an execution, simulation, or interpretation. Static analysis tools, on the other hand, derive their information entirely from inspection of the system's description, without requiring any kind of animation.

Dynamic analysis methods are basically variations on the traditional techniques of testing and debugging. The analysis tools associated with PAISLey [32,33] and DCDS [1], for example, are dynamic analysis tools. In each case, they support an interpretive animation, or simulation, of the behavior of a system described in the corresponding notation. The results of that animation can then be inspected in an attempt to determine properties of the system. Along with these dynamic analysis tools for use with pre-implementation descriptions, there has been some work on tools for testing and debugging of distributed programs. These include *post-mortem* debugging tools that work by generating trace information during the system's execution and then allowing the user to study the trace after execution has terminated (e.g., [24], [23], [22]) as well as more *interactive* debugging tools that permit the user to monitor, and possibly modify, the course of a system's execution (e.g., [8], [7], [16]).

Dynamic analysis methods, whether applied to pre-implementation descriptions or to completed programs, suffer from the fact that each distinct possible behavior of the system must be considered separately, by carrying out a separate testing or debugging "run." This aspect of dynamic analysis is problematic even for sequential software systems, since a realistically complex system has an extremely large, or possibly infinite, number of distinct possible behaviors. The problem is multiplied in distributed systems, since the nondeterministic interleaving of concurrent activities that characterizes such systems dramatically increases the number of possible behaviors. Moreover, nondeterminism and timing dependencies make it extremely difficult to identically reproduce a given test or debugging run, and hence to examine the effects of any modifications that might be made to the system. On the other hand, dynamic analysis is attractive because it is relatively straightforward to provide and produces concrete results directly from the description that the developer has created, whether that description is in a programming language or a pre-implementation notation. For these reasons, it may be of significant use in certain situations, such as during an initial exploration of a system's behavior or for detailed observation of some specific behavioral sequence.

Static analysis techniques can be further categorized as *state-based* or *event-based*. State-based analysis techniques proceed by considering the states or sequences of states that a system may attain, and attempting to determine properties of those states or state sequences. Such approaches are generally handicapped by the huge number of state variables that might be relevant to the state of a realistically complex system. This difficulty increases when state-based techniques are applied to distributed systems, since the number of state variables increases with the number of concurrently executing components. Moreover, the notion of overall system state is ill-defined for distributed systems, since there is no meaningful concept of global time in such systems [19].

One style of state-based analysis proceeds by generating a representation of the possible states of a system and determining whether certain states are reachable. This reachability analysis [18] is the fundamental approach to analyzing Petri nets. It has also been proposed by Taylor [29] and by Apt [2] as a general purpose approach to analyzing concurrent programs. The advantage of this style is that it is relatively straightforward to implement and that its exhaustive approach may lead to the discovery of behaviors that would otherwise be overlooked by the developer. It seems, however, to be prohibitively expensive to actually carry out for systems of realistic size, due to the necessity of generating and exploring the huge state spaces. Moreover, a large number of the states reported as reachable are typically uninteresting, and some may in fact be unreachable due to the highly simplified treatment of control flow dependencies used in this style of analysis. Thus the human analyst may be left with a major task in determining which of the reported states can actually be reached and which of those reachable states are of interest.

The other major style of state-based analysis is founded on theorem proving. Some logical system is associated with the state space and proofs regarding properties of that state space are attempted. Owicki and Gries were early advocates of this approach [26]. More recently, a number of researchers have investigated the application of temporal logic to this style of analysis (e.g., [27,28]). Reasoning about the state space instead of generating it is an attractive feature of this approach, but automating that reasoning has proven to be very difficult.

Event-based analysis is predicated on viewing a system's behavior as a sequence of event occurrences, where the granularity and complexity of the events considered varies according to the level at which the system is being described [31]. This approach seems particularly natural for analyzing distributed systems, in part because it avoids the ill-defined notion of the state of such systems. Hoare's trace model of CSP [17], Milner's CCS [25] and Lauer's COSY [21] are all examples of approaches that have

adopted event-based analysis. In each case, however, the analysis techniques are limited to use with the one specific notation with which they are associated, and their implementations, if any, have been preliminary at best.

As we explain in the next section, constrained expression analysis is a broadly applicable event-based approach that appears susceptible to significant levels of automation. In essence it is a rigorous formulation of a method of analysis based on arguments about number and order of event occurrences that has been widely, if less rigorously, used in conjunction with concurrent and distributed software systems (e.g., [15]). The approach seems to complement the others we have surveyed due to its highly focused style of analysis, which limits the amount of uninteresting information that it produces, and due to its generality, which makes it amenable to use with a wide variety of programming languages and pre-implementation notations. Furthermore, while some other analysis methods depend upon highly sophisticated tools such as automated theorem provers, it appears that relatively straightforward, well understood technology such as standard compiler construction tools and an integer programming package will be sufficient for automating a considerable portion of our approach.

The Constrained Expression Formalism

Constrained expression analysis of distributed systems is an event-based approach. In this approach, system descriptions given in a wide variety of design notations and programming languages are translated into formal representations, called *constrained expression representations*, to which a variety of analysis methods can then be applied. This approach to analysis allows developers of distributed systems to work in the notations and languages most appropriate to their tasks, while rigorous analysis of the systems they develop is based on the constrained expression representation.

We give here a brief overview of the constrained expression formalism. A detailed and rigorous presentation is given in [10], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [5]. The use of constrained expressions with a variety of design notations is illustrated in [5] and [14].

The constrained expression formalism treats behaviors of a distributed system as sequences of events. These events can be of arbitrary complexity, depending on the system characteristics of interest and the level of system description under consideration. By associating an *event symbol* to each event, we can regard each possible behavior of the system as a string over the alphabet of event symbols. Sets of such strings, and properties of those sets, then become the primary objects of interest in assessing the possible behaviors of a distributed system.

We use interleaving to represent concurrency. Thus, a string representing a possible behavior of a system that consists of a number of concurrently executing components is obtained by interleaving, or shuffling, strings representing the behaviors of the components. The events themselves are assumed to be atomic and indivisible, with only one event taking place at any time. Events that are to be explicitly regarded as overlapping in time can be represented by treating their initiation and termination as distinct atomic events. This view of events is essentially the same as that taken, for example, by Hoare in [17].

In the constrained expression formalism, the set of strings of event symbols representing behaviors of a particular distributed system is obtained by a two-step process. First a regular expression, called the *system expression*, is derived from a description of the system in some notation such as a design or programming language. The set of prefixes of the language of this system expression includes strings representing all the possible behaviors of the system. (We consider prefixes, rather than complete strings in the language of the system expression, in order to represent behaviors in which parts of the system terminate abnormally.)

Then this set is "filtered" to remove prefixes that do not represent possible behaviors of the system. A string survives this filtering process if its projections on certain subalphabets of the alphabet of event symbols lie in the languages of other expressions, called *constraints*. These constraints, which are not necessarily regular, are used to enforce various aspects of the semantics of the design or programming language, such as the appropriate synchronization of rendezvous between different tasks or consistent use of data. The process of deriving the appropriate regular expressions and constraints from a system description can be regarded as a standard compilation, and can be fully automated.

As a final step, symbols representing events that are not of interest when describing the final system behaviors are erased. The resulting set of strings is the *interpreted language* of the constrained expression. The interpreted language thus represents exactly the possible behaviors of the system. It is important to note that it is never necessary to actually generate this (possibly infinite) language. The analysis techniques operate on the constrained expression itself, rather than on individual strings in the interpreted language.

Analysis Based on the Constrained Expression Formalism

We have developed a number of powerful analysis techniques based on the constrained expression formalism. We give here a brief description of the most important of these techniques and show how it would be applied to a design for a distributed software system.

The fundamental approach is to phrase a question about the behavior of a system represented by a constrained expression in terms of whether a particular event symbol, or pattern of event symbols, occurs in a string representing a behavior of the system being analyzed. For example, questions about whether the system deadlocks might be phrased in terms of the occurrence of symbols representing the starvation of the component processes of the system.

We then assume that the specified symbol, or pattern of symbols, does indeed occur in a such a string. Starting from this assumption, we generate inequalities involving the number of occurrences of event symbols in segments of the string. (Inequalities holding in segments of a string can reflect properties involving the order of occurrence of events, as well as simply the number of their occurrences.) If the system of inequalities thus generated is inconsistent, we may conclude that the original assumption was incorrect, and the specified symbol or pattern of symbols does not occur in a string corresponding to a behavior of the system. If the system is consistent, we use the inequalities in attempting to construct a string containing the specified pattern.

An example of the use of this method of analysis is given in [6], where an informal version is used to detect an error in a solution to the distributed mutual exclusion problem and to verify that a modification removes the defect. Other examples appear in [5] and [10], where the method is applied to the dining philosophers problem, and in [4], where it is applied to a version of the automated gas station example used by Helmbold and Luckham [16] to illustrate their run-time monitoring approach to debugging Ada tasking programs. We now illustrate the use of this method by presenting a portion of the analysis of [4].

We have shown that constrained expression representations can be mechanically produced from system descriptions in a wide variety of design notations [14,11], but we have concentrated our recent efforts on the analysis of systems described in an Ada-based design language we have developed with Laura Dillon and our student Usha Sundaram. This language, called CEDL [12], focuses on the expression of communication and synchronization among the tasks in a distributed system, and language features not related to concurrency are kept to a minimum. Thus, for example, data types are limited, but almost all of the Ada control-flow constructs and the various forms of the Ada **select** statement have correspondents in CEDL. We have chosen to work with a design notation based on Ada because Ada is one of the few programming languages in relatively widespread use that explicitly provides for concurrency, and because we expect our work on analysis of designs to contribute to and benefit from the Arcadia Consortium's work on Ada software development environments [30].

Figures 1 and 2 illustrate the use of the CEDL notation. They show the declarations and one task body of a CEDL version of the automated gas station.

In this system, customers repeatedly arrive at the gas station and prepay for gas. This is represented by the rendezvous between a `CUSTOMER` task and the `OPERATOR` task at the `PREPAY` entry of the `OPERATOR`. If no customer is waiting, the operator activates the pump. Otherwise, the operator enters the customer's request in a queue.

After prepaying, a customer goes to the pump and starts it, pumps gas, and then stops the pump. These activities are represented by two calls to entries of the `PUMP` in the body of the `CUSTOMER` task. The customer then collects change from the operator, as modelled by an accept statement in the body of the `CUSTOMER` task.

After a customer has shut it off, the pump reports to the operator. This is modelled by the call to `OPERATOR.CHARGE` in the `accept FINISH_PUMPING` statement in the body of the `PUMP` task. The operator, who waits for a customer to prepay or for a report from the pump, gives change to the customer after this report. If another customer is waiting, the operator then reactivates the pump.

A set of translation rules for producing constrained expression representations from CEDL designs is given in [11]. These translation rules produce a *task expression* for each task in the system and a collection of constraints; the system expression of the constrained expression representation is the interleave of these task expressions. In Figure 4, we show the task expression corresponding to the pump task in our system. This task expression was derived using the translation rules of [11] and then simplified and reduced [13]. The line numbers in this figure are included for reference. The event symbols used in the task expression are essentially those of [11], with some simplification and abbreviation. A table showing the symbols and the associated events is given in Figure 3.

```
package COMMON is
  type C_NAME is (c1,c2);  -- names for two
                           -- customers
  type COUNTER is (zero,one,two,three);
                           -- enough to handle
                           -- 3 customers
end COMMON;

use COMMON;
task OPERATOR is
  entry PREPAY(CUSTOMER_ID : in C_NAME);
  entry CHARGE;
end OPERATOR;

task PUMP is
  entry ACTIVATE;
  entry START_PUMPING;
  entry FINISH_PUMPING;
end PUMP;

use COMMON;
task CUSTOMER_1 is
  entry CHANGE;
end CUSTOMER_1;

use COMMON;
task CUSTOMER_2 is
  entry CHANGE;
end CUSTOMER_2;
```

Figure 1: Task declarations for the two-customer gas station system

```
task body PUMP is
  begin
    loop
      accept ACTIVATE;
      accept START_PUMPING;
      accept FINISH_PUMPING do
        ...      -- compute charge for
                 -- this transaction
        OPERATOR.CHARGE;    -- report charge
                            -- to operator
      end FINISH_PUMPING;
    end loop;
  end PUMP;
```

Figure 2: Body of the `PUMP` task

Our analysis then proceeds by generating a system of inequalities relating the numbers of occurrences of certain events in a behavior of the distributed system under analysis. This process of generating inequalities begins with the assumption that a particular pattern of event symbols, reflecting a particular property of the system under analysis, occurs in a string in the interpreted language of the constrained expression. It is, of course, a primary task of the analyst to choose appropriate properties of the system for investigation.

Consider the question of whether a customer who prepays always gets to pump gas. Prepaying is modelled by a rendezvous between the CUSTOMER and OPERATOR tasks at the entry OPERATOR.PREPAY and pumping is modelled by a rendezvous between the CUSTOMER and PUMP tasks at the entry PUMP.START_PUMPING. In the bodies of the CUSTOMER tasks, the call to OPERATOR.PREPAY is followed immediately by the call to PUMP.START_PUMPING. Therefore, the only way that a customer can prepay but fail to pump is for the CUSTOMER task to starve calling the entry PUMP.START_PUMPING.

Since the two customer tasks in the system are treated symmetrically, we may thus begin our analysis by assuming that a $starve_c$(C1,P.start) symbol occurs in a constrained prefix and generating a system of inequalities starting from that assumption. We will show here how some of these inequalities are produced.

Let s be a constrained prefix containing the event symbol $starve_c$(C1,P.start), let $|event\ symbol|$ denote the number of occurrences of *event_symbol* in s, and let $|Pi|$ be 1 or 0 according as the projection of s on the alphabet of the task expression τ(P) lies in the language of the expression (P1)*Pi. We thus have

$$|starve_c(\mathrm{C1}, \mathrm{P.start})| = 1.$$

Working backward through the task expression τ(P) using the semantics of the regular expression operators and ignoring the *stop*(P) symbols for the moment, we have

$$\begin{aligned}
|call(\mathrm{P}, \mathrm{O.charge})| &= \sum_i |beg_rend(\mathrm{Ci}, \mathrm{P.finish})| \\
&\quad -|\mathrm{P5}| \\
\sum_i |beg\ rend(\mathrm{Ci}, \mathrm{P.finish})| &= \sum_i |end\ rend(\mathrm{Ci}, \mathrm{P.start})| \\
&\quad -|\mathrm{P4}| \\
|end_rend(\mathrm{C1}, \mathrm{P.start})| &= |beg_rend(\mathrm{C1}, \mathrm{P.start})| \\
|end_rend(\mathrm{C2}, \mathrm{P.start})| &= |beg_rend(\mathrm{C2}, \mathrm{P.start})| \\
\sum_i |beg_rend(\mathrm{Ci}, \mathrm{P.start})| &= |end_rend(\mathrm{O}, \mathrm{P.act})| - |\mathrm{P3}| \\
|end\ rend(\mathrm{O}, \mathrm{P.act})| &= |beg\ rend(\mathrm{O}, \mathrm{P.act})| \\
|beg\ rend(\mathrm{O}, \mathrm{P.act})| &= \sum_i |end\ rend(\mathrm{Ci}, \mathrm{P.finish})| \\
&\quad +1 - |\mathrm{P2}| \\
\sum_i |end_rend(\mathrm{Ci}, \mathrm{P.finish})| &= |resume(\mathrm{P}, \mathrm{O.charge})| \\
|resume(\mathrm{P}, \mathrm{O.charge})| &= |call(\mathrm{P}, \mathrm{O.charge})| \\
|beg\ rend(\mathrm{C1}, \mathrm{P.finish})| &= |end\ rend(\mathrm{C1}, \mathrm{P.finish})| \\
|beg\ rend(\mathrm{C2}, \mathrm{P.finish})| &= |end\ rend(\mathrm{C2}, \mathrm{P.finish})| \\
|kill\ rend(\mathrm{P.finish})| &= |\mathrm{P5}| \\
|starve_c(\mathrm{P}, \mathrm{O.charge})| &= |kill_rend(\mathrm{P.finish})| \\
\sum_i |beg_rend(\mathrm{Ci}, \mathrm{P.finish})| &\geq |starve_c(\mathrm{P}, \mathrm{O.charge}) \\
|starve_a(\mathrm{P.finish})| &= |\mathrm{P4}| \\
\sum_i |end_rend(\mathrm{Ci}, \mathrm{P.start})| &\geq |starve_a(\mathrm{P.finish})| \\
|starve_a(\mathrm{P.start})| &= |\mathrm{P3}| \\
|end\ rend(\mathrm{O}, \mathrm{P.act})| &\geq |starve_a(\mathrm{P.start})| \\
|starve_a(\mathrm{P.act})| &= |\mathrm{P2}|.
\end{aligned}$$

Since exactly one of the alternatives P2 through P5 occurs, we must also have

$$|\mathrm{P2}| + |\mathrm{P3}| + |\mathrm{P4}| + |\mathrm{P5}| = 1.$$

Symbol	*Associated event*
call(T,E)	Task T calls entry E
beg_rend(T,E)	Begin rendezvous with task T at entry E
end_rend(T,E)	End rendezvous with task T at entry E
resume(T,E)	Resume task T after rendezvous at entry E
$starve_c$(T,E)	Task T starves on call to entry E
$starve_a$(E)	Task starves waiting to accept a call at entry E
kill rend(E)	Rendezvous at entry E is aborted
stop(T)	Execution of task T stops

In the symbols used in the task expression in Figure 4, the task name CUSTOMER.1 is abbreviated to Ci, PUMP is abbreviated to P, and OPERATOR is abbreviated to O. Entry names are also abbreviated.

Figure 3: Event Symbols Used in the PUMP Task Expression and Associated Events

P1 $\Big($ *beg rend*(O,P.act)*end rend*(O,P.act)

$\big(\bigvee_i$ *beg_rend*(Ci,P.start)*end_rend*(Ci,P.start)$\big)$

$\big(\bigvee_i$ *beg_rend*(Ci,P.finish)*call*(P,O.charge)*resume*(P,O.charge)

end_rend(Ci,P.finish)$\big)\Big)^*$

P2 $\Big($ $starve_a$(P.act)*stop*(P)

P3 $\vee$ *beg_rend*(O,P.act)*end_rend*(O,P.act)$starve_a$(P.start)*stop*(P)

P4 $\vee$ *beg_rend*(O,P.act)*end_rend*(O,P.act)$\big(\bigvee_i$ *beg_rend*(Ci,P.start)

end_rend(Ci,P.start)$\big)$ $starve_a$(P.finish)*stop*(P)

P5 $\vee$ *beg rend*(O,P.act)*end rend*(O,P.act)$\big(\bigvee_i$ *beg rend*(Ci,P.start)

end rend(Ci,P.start)$\big)\big(\bigvee_i$ *beg rend*(Ci,P.finish)$\big)$

$starve_c$(P,O.charge)*kill_rend*(P.finish)*stop*(P)$\Big)$

Figure 4: Task Expression τ(P) Associated with the Task PUMP

These inequalities represent the unconstrained behavior of the PUMP task. In a similar fashion, we can generate inequalities from the other task expressions.

The constraints in a CEDL constrained expression representation enforce the appropriate synchronization between tasks, as well as other aspects of the semantics of CEDL that are not conveniently represented in the task expressions. These constraints are used to generate additional inequalities. For example, a constraint that represents the semantics of the rendezvous between the PUMP task and the OPERATOR task at the entry PUMP.ACTIVATE implies that

$$|call(\mathrm{O}, \mathrm{P.act})| = |beg\_rend(\mathrm{O}, \mathrm{P.act})| \quad \text{and}$$
$$|end\_rend(\mathrm{O}, \mathrm{P.act})| = |resume(\mathrm{O}, \mathrm{P.act})|.$$

Other constraints lead to similar inequalities.

Having generated the full system of equations and inequalities, we use a standard branch-and-bound integer linear programming package [20] to determine whether it is consistent. (For convenience, we usually choose the objective function to minimize the sum of the variables.) For the gas station example described here, the system consists of roughly 100 equations and inequalities and the linear programming package finds a solution to the system of inequalities corresponding to a behavior in which the task CUSTOMER_1 starves because of a deadlock. This requires approximately one minute of computer time on a Celerity C1260. A similar analysis shows that, when the call to the operator is moved out of the accept FINISH PUMPING statement in the body of the PUMP task and the order of two calls in the body of the OPERATOR task is reversed, a customer who prepays always gets to pump gas. The constrained expression analysis thus detects an error in the design and establishes that a modification to the design eliminates the problem.

Tools Supporting Constrained Expression Analysis

The constrained expression approach to analysis offers several potentially significant advantages. It can be applied at a number of stages of the software development process, including especially the pre-implementation stages of specification and design, and it can be used with a wide variety of design notations and programming languages [14]. This allows system designers to work with the languages and notations they find most appropriate without sacrificing the ability to do rigorous analysis. Because analysis based on the constrained expression formalism works with whole classes of system behaviors and can be conducted in a highly directed fashion, the problem of combinatorial explosion is ameliorated. Moreover, reporting of spurious errors is reduced compared to most state-based analysis methods, since constrained expression analysis methods can take data dependency into account.

Preliminary experiments, like the one outlined above, have been extremely promising. Applied to a variety of small concurrent systems, these methods have been able to detect subtle errors, and to prove rigorously that modifications to the systems eliminate those errors [4,5,6,10] However, considerable experience with distributed systems of realistic size and complexity will be necessary before the constrained expression analysis techniques can become practical tools for software developers. Only through experimentation with such realistic examples can the strengths and weaknesses of the techniques, the classes of problems for which they are best suited, and their most appropriate role in the software development process be accurately determined.

For two important reasons, however, even experimentation with such examples requires robust implementations of constrained expression analysis tools. First, automated support is necessary for the application of the analysis techniques to systems much larger than the examples we have already studied. It is not possible to cope with systems of several hundred inequalities with only paper and pencil. Second, the utility of constrained expression analysis tools in practice will be affected by such factors as the efficiency of their implementations and the kind and quality of their interfaces with other tools and with human analysts. These issues can only be explored through a serious software engineering effort, involving the design, construction, and evaluation of prototype tools. For these reasons, we regard the construction of such tools, and their application to realistic examples of concurrent systems, as an integral part of our research on analysis techniques.

We have begun to construct a constrained expression toolset to support such experimentation. This toolset consists of three main tools. The first of these is a *deriver*, which is essentially a compiler used to produce constrained expression representations from designs given in some programming or design notation. We have nearly completed work on a prototype of this tool for use with the CEDL design language. The prototype is being written in Ada, using standard compiler construction tools and the Graph Definition Language and Graphite processor that we have developed as part of the Arcadia project [9,30]. It was designed to make modification as straightforward as possible, so that enhancements to CEDL and the translation rules used to produce constrained expression representations would be easy to implement. Further development of this tool will be the result of improvements in our translation rules for CEDL or modifications to the internal representation used for constrained expressions. The deriver will be supplemented by constrained expression simplification tools now under construction at the University of California, Santa Barbara.

The second tool is a *behavior generator*, which is used to produce strings in the interpreted language of a constrained expression. This tool is used in the initial exploration of a concurrent system, when the analyst "walks" through the system to get an idea of its functioning. It is also used when the inequalities produced in other stages of analysis are consistent and the analyst tries to produce an actual system behavior satisfying the inequalities. With a large system, this involves a substantial amount of bookkeeping, and often a great deal of backtracking as one tries to satisfy a large number of inequalities and constraints simultaneously. We have a complete specification [3] for the behavior generator, and a partial implementation in LISP. Although we expect to continue to use LISP for rapid prototyping, later versions of this tool will migrate to Ada for compatibility with the Arcadia project and other tools.

Finally, we will need an *inequality generator* to provide automated support for the generation of the inequalities in analysis and their conversion to a form suitable for input to the integer programming package. We have built a prototype inequality generator using LISP that has been useful for some work with small systems. This prototype, however, does not make use of some important types of constraints and generates inequalities

in a relatively undirected fashion compared to the heuristics we have developed for use in hand analyses. Further development of the inequality generator may involve attempts to integrate the heuristics into our current prototype or the construction of a separate tool based on those heuristics, and will certainly depend on the results of ongoing theoretical work on modularizing analysis and incremental generation of inequalities. As with the behavior generator, we expect to continue to use LISP in early versions of the inequality generator, with eventual migration to Ada.

It is apparent that interfaces between these tools, and between some of the tools and the analyst, are extremely important, and we are concerned with both kinds of interfaces. Intertool interfaces, in the context of our project, will primarily consist of internal representations of constrained expressions and representations of inequalities suitable for input to the integer programming package. The latter will be dictated by the particular integer programming package that we are using at any given time. We plan to implement the internal representation of constrained expressions using the Graph Definition Language and Graphite processor [9]. This will allow us to explore alternatives or even adopt a completely new internal representation with only minimal impact on the implementation of other aspects of our tools.

The user interfaces to the existing, exploratory versions of our tools are far from friendly. In the long term, we envision a sophisticated user interface employing multiple windows and a pointing device. Initially, however, we need to address more basic concerns such as appropriate forms in which to report the activity of the deriver, the inequality generator, the integer programming package and the behavior generator. Eventually, we will want to hide the constrained expressions from the tool user as much as possible, so that developers of concurrent software will be able to reap the benefits of using the tools without having to understand the constrained expression formalism. We have tentative ideas in this direction, and will be exploring them further as we build the prototype toolset.

Conclusion

Developing large-scale, reliable software capable of exploiting the potential of distributed hardware systems will demand the support of powerful automated tools, especially analysis tools. Our *constrained expression* approach to analyzing such software systems seems to have several advantages, including broad applicability and reasonable efficiency, relative to other proposed approaches. In this paper we have outlined the constrained expression approach, indicated its relationship to some alternative approaches, and described our experimental use of the approach in analyzing a simple but nontrivial example of a distributed software system design.

Results of initial experiments with our approach, based on manual application and preliminary prototypes of some of the needed tools, have been encouraging. They have also demonstrated the need for us, and all researchers working on distributed software analysis, to undertake more extensive experimentation with larger, more realistic examples.

In this paper, we have outlined our plans for constructing the more robust, flexible and efficient prototype tool implementations needed to support more extensive experimentation. We look forward to the completion of this implementation activity, and to the availability of similarly robust implementations of tools supporting alternative approaches to analysis of distributed software. Serious experimentation, leading to meaningful assessments and comparisions of the various approaches, their practical utility and particular shortcomings, will then be possible. This process should eventually result in a collection of analysis tools with complementary capabilities that together will support analysis across a wide range of problem classes and phases of the software development process. Such a collection of tools would be extremely valubable to developers of distributed software.

REFERENCES

[1] M. W. Alford. SREM at the age of eight; the distributed computing design system. *Computer*, 18:36–46, April 1985.

[2] K. R. Apt. A static analysis of CSP programs. In *Proceedings of the Workshop on Program Logic*, Pittsburgh, June 1983.

[3] S. Avery. *Development of a Behavior Generator for Constrained Expressions.* Technical Report SDLM84-2, Software Development Laboratory, Department of Computer and Information Science, University of Massachusetts, Amherst, June 1984.

[4] G. S. Avrunin. *Experiments in Constrained Expression Analysis.* Technical Report 87-125, Department of Computer and Information Science, University of Massachusetts, 1987.

[5] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: adding analysis capabilities to design methods for concurrent software systems. *IEEE Transactions on Software Engineering*, SE-12(2):278–292, 1986.

[6] G. S. Avrunin and J. C. Wileden. Describing and analyzing distributed software system designs. *ACM Transactions on Programming Languages and Systems*, 7(3):380–403, July 1985.

[7] P. C. Bates and J. C. Wileden. High-level debugging of distributed systems: the behavioral abstraction approach. *Journal of Systems and Software*, 3:255–264, 1983.

[8] A. F. Brindle, R. N. Taylor, and D. F. Martin. *A Debugger for Ada Tasking.* Technical Report ATR-85(8033)-1, Aerospace Corporation, 1985.

[9] L. A. Clarke, J. C. Wileden, and A. L. Wolf. GRAPHITE: a meta-tool for Ada environment development. In *Proceedings of 2nd International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

[10] L. K. Dillon. *Analysis of Distributed Systems Using Constrained Expressions.* PhD thesis, University of Massachusetts, Amherst, 1984.

[11] L. K. Dillon. *A Constrained Expression Formulation of CEDL.* Technical Report TRCS86-22, Department of Computer Science, University of California, Santa Barbara, November 1986. Revised July 1987.

[12] L. K. Dillon. *Overview of the Constrained Expression Design Language.* Technical Report TRCS86-21, Department of Computer Science, University of California, Santa Barbara, October 1986.

[13] L. K. Dillon. *Simplification and Reduction of CEDL Constrained Expressions.* Technical Report, Department of Computer Science, University of California, Santa Barbara, October 1986.

[14] L. K. Dillon, G. S. Avrunin, and J. C. Wileden. *Constrained Expressions: Toward Broad Applicability of Analysis Methods for Distributed Software Systems.* To appear in *ACM Transactions on Programming Languages and Systems.*

[15] A. N. Habermann. Synchronization of communicating processes. *Communications of the ACM*, 15(3):171–176, 1972.

[16] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

[17] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[18] R. M. Karp and R. E. Miller. Parallel program schemata. *Journal of Computer and Systems Science*, 3(4):167–195, May 1969.

[19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978.

[20] A. H. Land and S. Powell. *Fortran Codes for Mathematical Programming: Linear, Quadratic and Discrete.* John Wiley & Sons, Ltd., London, 1973.

[21] P. Lauer, P. Torrigiani, and M. Shields. COSY: a system specification language based on paths and processes. *Acta Informatica*, 12(2):451–503, 1979.

[22] R. J. Leblanc and A. D. Robbins. Event-driven monitoring of distributed programs. In *Proceedings of the 5th International Conference on Distributedomputing Systems*, pages 515–522, May 1985.

[23] C. H. LeDoux and D. S. Parker. Saving traces for Ada debugging. In *Proceedings of the Ada International Conference*, pages 97–108, May 1985.

[24] B. P. Miller, C. Macrander, and S. Sechrest. A distributed program monitor for Berkeley Unix. In *Proceedings of the 5th International Conference on Distributed Computing Systems*, pages 43–54, May 1985.

[25] R. Milner. *A Calculus of Communicating Systems.* Volume 92 of *Lecture Notes in Computer Science*, Springer-Verlag, Berlin, 1980.

[26] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[27] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Transactions on Programming Languages and Systems*, 4:455–495, July 1982.

[28] K. Ramamritham and R. M. Keller. Specification and synthesis of synchronizers. In *Proceedings of the 1980 International Conference on Parallel Processing*, pages 311–321, August 1980.

[29] R. N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[30] R. N. Taylor, L. A. Clarke, L. J. Osterweil, J. C. Wileden, and M. Young. Arcadia: a software development environment research project. In *Proceedings of the 2nd International Conference on Ada Applications and Environments*, pages 137–149, April 1986.

[31] J. C. Wileden. Applying event based analysis to specifications and designs. In H. Kugler, editor, *Information Processing 86*, pages 577–581, Elsevier Science Publishers, Amsterdam, September 1986.

[32] P. Zave. An operational approach to requirements specification for embedded systems. *IEEE Transactions on Software Engineering*, SE-8(3):250–269, May 1982.

[33] P. Zave and W. Schell. The PAISLey software tools: an environment for executable specifications. In *Proceedings of the Workshop on Software Engineering Environments for Programming-in-the-Large*, pages 54–63, June 1985.