The Specification and Verified Decomposition of System Requirements Using CSP

ANDREW P. MOORE

Abstract—An important principle of building trustworthy systems is to rigorously analyze the critical requirements early in the development process, even before starting system design. Existing proof methods for systems of communicating processes focus on the bottom-up composition of component-level specifications into system-level specifications. Trustworthy system development requires, instead, the top-down derivation of component requirements from the critical system requirements. This paper describes a formal method for decomposing the requirements of a system into requirements of its component processes and a minimal, possibly empty, set of synchronization requirements. The Trace Model of Hoare's Communicating Sequential Processes (CSP) is the basis for the formal method. We apply the method to an abstract voice transmitter and describe the role that the EHDM verification system plays in the transmitter's decomposition. In combination with other verification techniques, we expect that the method defined here will promote the development of more trustworthy systems.

Index Terms—Automated theorem proving, communicating processes, formal specification, formal verification, process algebras, requirements definition, trusted systems, safety, security.

I. INTRODUCTION

A critical system is any system that can behave catastrophically. A critical requirement of a system is any requirement that if not satisfied can result in catastrophic behavior. In safety-critical systems, a catastrophe might be the unauthorized disclosure of information; in safety-critical systems, it might be the uncontrolled release of energy. Most critical systems require an extremely high degree of assurance that they meet their critical requirements. In combination with more conventional verification and validation techniques, formal methods can help attain this increased level of assurance.

An important step in applying formal methods to the development of a trustworthy system is to specify formally its critical requirements. Once these are specified and the design process has begun, methods are needed to derive lower-level component requirements from these critical requirements. Past work [1]–[4] suggests that process algebras, such as Hoare's Communicating Sequential Processes (CSP) [5], facilitate the formal specification of a system's critical requirements. Unfortunately, existing proof methods for these languages focus on the bottom-up composition of component-level specifications into system-level specifications, rather than a top-down derivation of component requirements from the (critical) system requirements. With this in mind, we define a formal method for decomposing requirements of a system into requirements of its component processes and a minimal, possibly empty, set of synchronization requirements. CSP is the basis for describing the functionality of systems; the Trace Model is the basis for reasoning about properties of systems described in CSP [6], [7].

The specification and decomposition of system requirements proceeds in successive stages of refinement and proof. Fig. 1 illustrates the three layers of specification required for a decomposition. As shown in the highest layer, requirements may exist that cannot be partitioned completely into requirements on an individual component; such requirements involve the synchronized behavior of two or more components. Since these synchronization requirements are typically more difficult to verify, the decomposition method promotes reducing their number and complexity as far as possible. The set of these requirements is minimal if, when each requirement is described in conjunctive normal form, each conjunct of each requirement depends on the behavior of two or more components.

Section II of this paper compares our method with other methods based on similar goals. Section III presents an overview of the relevant elements of the Trace Model of CSP, extends the CSP notation to facilitate hierarchical decomposition, and refines the method discussed above based on the model described. Section IV describes in detail the application of the method to an abstract voice transmitter. Finally, Section V discusses the use of the EHDM verification system [8], [9] to check mechanically the integrity of the decomposition process, both in general and as it was used in the decomposition of the voice transmitter. General conclusions of this effort and plans for future research are also presented. Proofs in the main body of the paper are written in the style used in [10], where justifications for each proof step are provided as hints enclosed in curly brackets. The complete proofs of the theorems used in this paper and the decomposition of the voice transmitter using EHDM are documented in [11], [12].

U.S. Government work not protected by U.S. copyright
II. COMPARISON TO RELATED WORK

Hoare’s work on CSP [5], [13] and Milner’s work on CCS [14], [15] have provided the basis for many of the methods described for the design and verification of concurrent systems. Previous comparisons [6], [16]–[18] characterize these methods by, among other things, the model of concurrency used, the types of properties provable, and the structure of systems specifiable. Due to the wealth of work done in this area we restrict our comparison primarily to methods based on CSP. Rather than explicitly describing each method and comparing it with our own, we describe how our method fits into the previously defined characterizations. This provides an implicit comparison with much of the previous work in a relatively short amount of space. We also compare our work with efforts to use traces for the abstract description of software and with more recent work not included in the past characterizations.

A. A Family of CSP Models

Olderog and Hoare [6] describe a family of increasingly sophisticated models for communicating processes: the Counter Model [19], [20], the Trace Model [7], [21], [22], the Divergences Model, the Readiness Model [19], [23], and the Failure Model [24]–[26]. Starting with the least sophisticated, the Counter Model involves the description of systems using separate channel histories. This model is shown to deal adequately only with acyclic or tree-like networks of processes. The Trace Model allows the description of arbitrary networks of processes. Both the Counter Model and the Trace Model can specify safety properties, but neither can deal adequately with diverging processes. The Divergences Model suffices to reason about systems that may diverge. The Readiness and Failure Models are required to reason about liveness in addition to safety. More recent work proposes the use of algebraic equations, rather than traces, to describe and prove properties about CSP [27].

Olderog and Hoare prove that the Trace Model is sufficient to reason about safety properties of nondivergent cyclic networks of processes. We choose this model for specifying and decomposing system requirements as a balance between its expressive power and the complexities involved with its use. For example, the Counter Model makes it difficult to specify certain relationships of values transmitted across different channels due to the fact that channel histories are recorded separately. The Gypsy Verification Environment [28], [29] exhibits this same difficulty. Although useful for the verification of a variety of concurrent applications, Gypsy has a number of limitations on the form of the specification and implementation of concurrent programs. At the time this paper was written, it was very difficult to specify exit conditions of the form “at the time a message is received over channel A, the history of channel B (distinct from A) satisfies property P.” The Trace Model alleviates this problem by interleaving the individual channel histories in the order in which the communications take place.

B. Characteristics of Trace-Driven Proof Systems

Hooman [17] and Barringer [18] describe a number of characteristics of proof systems for networks of processes. They distinguish between proof systems based on shared variables [17], [30], [31] versus those based on message passing, e.g., most systems based on CSP. Our method requires specifically that the only way one process may communicate with another process is through the transmission of messages over channels. Hooman distinguishes between proof systems based on a posteriori verification [32], [33] versus verification as part of the design process [21], [34], [35]. Since the design and implementation of nontrivial systems rarely, if ever, proceeds without error on the first attempt, an interactive verification/development process is an important part of our method.

A proof system is compositional if it is possible to derive a specification of a system from the specification of its components without knowledge of the component’s internal structure. The verify-while-develop paradigm of system development has led to the need for compositional proof systems [21], [30], [34], [35] versus noncompositional proof systems [31], [33]. We introduce a special compose operator to CSP allowing the description of a process based solely on the external channels over which the process communicates. By adapting proof rules from [5], a compositional proof method is defined similar to the approach used by van de Snepscheut for the verification of hardware designs [10]. However, unlike previous work, our method emphasizes the decomposition of high-level requirements, rather than the composition of low-level specifications.

In a slightly different vein, Bartussek and Parnas [36] and, more recently, Parnas and Wang [37] describe the use of trace theory for the abstract specification of software modules. Rather than reasoning about traces of communications, as does the Trace Model of CSP, this work reasons about software using traces of procedure calls. McLean extended the original work by providing a formal
foundation for specifying software requirements using traces [38] and defining the trace semantics of a simple sequential programming language [39]. This provides a general framework for specifying and verifying sequential program behavior. In combination with the method described in this paper for decomposing system-level requirements into component-level requirements, this framework may be useful for specifying and verifying the critical requirements of concurrent software systems.

C. Applications to Critical Systems

McCullough [40] and Jacob [1] use trace-based methods to reason about the multilevel security requirements of systems. McCullough introduces the notion of a composable property as one that if proven of the components of a system, is true of the system as a whole. He derives component requirements from arbitrary system requirements. Composability applies only to systems composed of components similar enough that the same critical requirements apply to all components and to the system as a whole. The goal of our method is to tackle the more general problem of deriving component requirements from arbitrary system requirements.

Jacob [1] demonstrates the expressive power of the Trace Model of CSP by specifying complex multilevel security requirements of systems. In later work, Woodcock [41] and Jacob [42] describe a method to derive an implementation from a specification not by intelligently guessing, but rather under guidance from the structure of the specification. Woodcock’s approach requires stating a set of fairly low-level requirements and generating an implementation consisting of a parallel composition of processes, each component of which implements a requirement. Jacob discusses problems with a (possibly nonterminating) method of generating more secure designs from less secure designs using Woodcock’s basic approach. Our method is different in that we are trying to derive low-level requirements from a given high-level requirement and (partial) implementation. We do not assume that we have the freedom to choose the implementation since that choice may depend on other considerations, e.g., the required noncritical functionality of the system or the physical realization of the system in hardware.

III. THE DECOMPOSITION METHOD

This section describes an iterative seven-step method for decomposing system requirements. We present an overview of relevant elements of the CSP Trace Model and describe a (primarily syntactic) extension for constructing systems in a hierarchical manner. In the course of this discussion, three rules are introduced that are used to justify the definition of the method. We use standard notation where possible and describe CSP-specific notation when first used. We assume universal quantification of variables in formulas unless otherwise indicated. Readers interested in a summary of the notation used or other details of CSP not covered in this paper should refer to [5].

A. CSP Preliminaries

A CSP process communicates with its environment through named communication channels. The Trace Model of CSP maps a process to an alphabet and a set of traces. The alphabet of a process $P$, denoted $\alpha P$, specifies all events in which $P$ is permitted to engage. A trace of a process is an observation of its execution. It consists of a finite sequence of events in which the process has engaged at some moment in time. The set of all traces of a process $P$, denoted traces ($P$), is a prefix-closed nonempty subset of $\alpha P^*$, where $\alpha P^*$ is the set of all finite traces formed from events in $\alpha P$.

The CSP notation allows the description of processes using a variety of process constructors. This paper primarily deals with the commutative concurrency operator ||. $P || Q$ describes a process executing process $P$ concurrently with process $Q$. $P \parallel Q$ requires $P$ and $Q$ to participate simultaneously in those events that occur in both $\alpha P$ and $\alpha Q$. Events occurring in $\alpha P$ but not $\alpha Q$ may be engaged in by $P$ independently of $Q$. Since either process of a concurrent composition may itself be a concurrent process, this operator supports the description of arbitrary networks of processes.

Processes executing concurrently communicate through channels. A communication event is a special type of event described by a pair $c.v$. The alphabet of process $P$ contains $c.v$ if and only if $P$ is permitted to communicate message $v$ over channel $c$. $c!v$ denotes the output of value $v$ on the channel $c$; $c?m$ denotes the input of any value $m$ communicable on the channel $c$. These operations are communication events defined by

Definition 1: ($c!v \rightarrow P) = (c.v \rightarrow P)$;

Definition 2: ($c?m \rightarrow P(m)) = (c.n!m \rightarrow P(n)))$;

where the prefix process $e \rightarrow P$ describes a process that first engages in the event $e$ and then behaves like process $P$; and the choice process $x:A \rightarrow P(x)$ describes a process that chooses $x$ from $A$ then behaves like $P(x)$. Although the CSP notation distinguishes between the input and output of values over channels, the Trace Model uses only the generic dot notation, $c.v$, to represent communications over channels. For example, traces ($c!v \rightarrow P$) = \{< >\} \cup \{<c.v\> | t \in traces (P)\},

where $\top$ is the trace append operation.

Henceforth, the term requirement refers to the statement of a property. The term specification refers to the statement that a system, or process, satisfies some property.
A communication of message \( m \) over channel \( c \) can occur between two processes running concurrently if and only if both processes have the communication event \( c.m \) in their alphabets and both processes simultaneously engage in that event. That is, whenever one process outputs a value onto the channel, the other process simultaneously inputs the same value from the channel.\(^\text{9}\) This implies that

\[
(c \not\in v \rightarrow P) \parallel (c \not\in m \rightarrow Q(m)) = (c.v \rightarrow P) \parallel (Q(v))
\]

where \( c.v \) occurs in \( \alpha P \) and \( \alpha Q \). If only one process in a system has a communication event in its alphabet, then that process may engage in that event independently of any other process. To simplify the theory involved, Hoare assumes that at most two processes in a system can access the same communication channel and that communication over a channel occurs in only one direction. If only one process within the system can access the channel, the channel is said to be external; if two processes can access the channel, the channel is said to be internal.

A requirement in CSP is viewed as a set of traces. Process \( P \) satisfies a requirement \( R \), denoted \( P \models R \), if and only if \( R \) contains every trace that may occur as an observation of \( P \).

\[\text{Definition 4: } (P \models R) \iff (\text{traces } (P) \subseteq R)\]

Clearly, a requirement is satisfiable by some process only if it contains a nonempty prefix-closed subset. For convenience we define a functional notation for requirements

\[\text{Definition 5: } R(trl) = \{trl \in R\}\]

and a predicate ValidTrace as

\[\text{Definition 6: } \text{ValidTrace} (trl, P) = \{trl \in \text{traces } (P)\}\]

Then, trivially,

\[\text{Lemma 1: } (P \models R) \iff \forall trl. (\text{ValidTrace} (trl, P) \Rightarrow R(trl))\]

\[\text{B. Concurrent Processes and Hiding}\]

The Trace Model requires relating each process to an alphabet and a set of traces. Since our goal is to decompose the requirements of a system as far as possible into requirements of its components, it is helpful to define the alphabet and set of traces of a concurrent process in terms of its component processes. Clearly,

\[\text{Definition 7: } \alpha(P \parallel Q) = \alpha P \cup \alpha Q\]

The valid traces of a concurrent process are defined as

\[\text{Definition 8: } \text{ValidTrace} (trl1, P \parallel Q) = (\text{ValidTrace} (trl1 \uparrow \alpha P, P) \wedge \text{ValidTrace} (trl1 \uparrow \alpha Q, Q) \wedge trl1 \in (\alpha P \cup \alpha Q)^*),\]

where the restriction operator \( \uparrow \) takes a trace and a set of events and returns the trace with the elements not in the set removed. Hoare justifies Definition 8 by arguing that if \( trl1 \in \text{traces } (P \parallel Q) \), then every event of \( trl1 \) must be an element of either \( \alpha P \) or \( \alpha Q \). For every event \( e \) in \( trl1 \), \( e \in \alpha P \) if and only if \( e \) occurs in the trace of \( P \). Likewise, for every event \( e \) in \( trl1 \), \( e \in \alpha Q \) if and only if \( e \) occurs in the trace of \( Q \). Therefore, \( (trl1 \uparrow \alpha P) \in \text{traces } (P) \) and \( (trl1 \uparrow \alpha Q) \in \text{traces } (Q) \). Definition 7 suggests that \( trl1 \) must be an element of \( (\alpha P \cup \alpha Q)^* \).

Definition 8 requires that any trace of a concurrent process \( P \parallel Q \) include every event in which \( P \) or \( Q \) engage. The visibility of the communications over internal channels in the traces of \( P \parallel Q \) reduces the amount of abstraction possible during the system design process. Hierarchical design, a proven method for managing the complexity of system design and verification, requires that the specification of a component be based solely on the sequence of external communications in which it may engage. To support this, we define the compose operator, denoted \( \parallel \), equivalent to the CSP concurrency operator except that the internal communications are hidden; consequently,

\[\text{Definition 9: } \alpha(P \downarrow Q) = (\alpha P + \alpha Q)\]

where \( (\alpha P + \alpha Q) = (\alpha P \cup \alpha Q) - (\alpha P \cap \alpha Q) \). Further, we define a new operator \( \dag \) which extends the definition of \( \parallel \) to operate on sets of traces, as

\[\text{Definition 10: } A \uparrow S = \{t \mid \exists t', (t' \in A \wedge (t' \downarrow S = t))\}\]

This extension allows the definition of the traces of \( P \downarrow Q \) in terms of the traces of \( P \parallel Q \) as

\[\text{Definition 11: } \text{traces } (P \downarrow Q) = (\text{traces } (P \parallel Q) \setminus (\alpha P \downarrow \alpha Q)).\]

Concealing communication over internal channels in this way is equivalent to that accomplished by the trace blend operation defined in [10].

Clearly from the above definitions, the alphabet and traces of a compose process contain only the external communication events in which the process may engage. Thus, each valid trace of a compose process corresponds to some valid trace of the related concurrent process as follows:

\[\text{Rule 1: } \text{ValidTrace} (trl1, P \downarrow aQ) = \exists trl2. (\text{ValidTrace} (trl2, P \parallel Q) \wedge trl1 = (trl2 \uparrow (\alpha P + \alpha Q)))\]

\[\text{Proof:}\]

\[\text{ValidTrace} (trl1, P \downarrow aQ) = \{\text{Definition 6}\}\]

\[trl1 \in \text{traces } (P \downarrow aQ) = \{\text{Definition 11}\}\]

\[trl1 \in \{\text{traces } (P \parallel Q) \uparrow (\alpha P + \alpha Q)\}\]

\[^{9}\text{To account for the time it takes to transmit information over physical wires we can associate with each interface between two processes an additional process that delays transmissions for some length of time. For ease of exposition, this paper does not consider issues regarding transmission delay; for details see [12].} \]
End of Proof.

Note that the compose operator is just a convenient notation for what could be defined using concealment in Hoare’s CSP. For example,

\[(P \parallel Q) = ((P \parallel Q) \setminus (\alpha P \cap \alpha Q))\]

where the CSP concealment operator \(\setminus\) is used to hide all communications over internal channels.

As mentioned previously, use of the Trace Model requires showing that the application of interest is nondivergent. Divergence arises from unguarded recursion. A recursion of the form \(\mu P.F(P)\) is guarded if \(F(P)\) starts with at least one event prefixed to all recursive occurrences of \(P\). Divergence can also arise from the hiding of events as is accomplished through the use of the compose operator. Any process that can engage in an infinite consecutive sequence of hidden events is divergent. In the case of a compose process, divergence leads to infinite internal chatter. Hoare defines a theory for reasoning about divergent processes. From this theory we have defined a method for stating requirements on component processes sufficient to guarantee nondivergence of a system [12]. This is a fairly mechanical process and is performed in isolation from the requirement decomposition process; we, therefore, do not further consider issues of divergence in this paper.

C. Decomposing System Requirements

We are now able to state and prove the primary inference rules for decomposing requirements of a compose process. The first reduces the problem of proving requirements of a compose process to proving requirements of a concurrent process:

**Rule 2:** \((P \parallel Q) \text{ sat } R\)

\[\forall tr1. (ValidTrace(tr1, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q))) \rightarrow \exists tr2. (ValidTrace(tr2, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q)))\]

\[= (P \parallel Q) \text{ sat } R\]

**Proof:**

\[\forall tr1. (ValidTrace(tr1, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q))) \rightarrow \exists tr2. (ValidTrace(tr2, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q)))\]

End of Proof.

The hypothesis of this rule requires proving properties about the requirements of a concurrent process. The following inference rule reduces the problem of proving requirements of a concurrent process to proving requirements of its components:

**Rule 3:**

\[((P \text{ sat } S) \wedge (Q \text{ sat } R)) \rightarrow \exists tr2. (R(tr2) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q))) \wedge (R(tr2) = R(tr2 \uparrow (\alpha P \div \alpha Q)))\]

\[= (P \parallel Q) \text{ sat } R\]

**Proof:**

\[\forall tr1. (ValidTrace(tr1, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q)) \wedge (R(tr1) = R(tr1 \uparrow (\alpha P \div \alpha Q))))) \rightarrow \forall tr1. (ValidTrace(tr1, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q)) \wedge (R(tr1) = R(tr1 \uparrow (\alpha P \div \alpha Q)))\]

\[\rightarrow \exists tr2. (R(tr2) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q))) \wedge (R(tr2) = R(tr2 \uparrow (\alpha P \div \alpha Q)))\]

\[\rightarrow \exists tr2. (R(tr2) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q))) \wedge (R(tr2) = R(tr2 \uparrow (\alpha P \div \alpha Q)))\]

\[\rightarrow (((S(tr1) \Rightarrow S(tr1)) \wedge (R(tr1) \Rightarrow R(tr1))) \wedge (S(tr1) \wedge R(tr1)) = T(tr1))\]

\[= (P \parallel Q) \text{ sat } T\]

\[\forall tr1. (ValidTrace(tr1, P \parallel Q) \wedge (tr1 = tr2 \uparrow (\alpha P \div \alpha Q)) \wedge (R(tr1) = R(tr1 \uparrow (\alpha P \div \alpha Q))) \wedge (((S(tr1) \Rightarrow S(tr1)) \wedge (R(tr1) \Rightarrow R(tr1))) \wedge (S(tr1) \wedge R(tr1)) = T(tr1)))\]

End of Proof.
→ \{ \text{Lemma 1} \}

\forall tr_1. \text{ValidTrace}(tr_1, P \parallel Q)
\Rightarrow (S(tr_1 \uparrow \alpha P) \land R(tr_1 \uparrow \alpha Q)
\land (S(tr_1 \uparrow \alpha P) \Rightarrow S(tr_1))
\land (R(tr_1 \uparrow \alpha Q) \Rightarrow R(tr_1))
\land ((S(tr_1), R(tr_1)) \Rightarrow T(tr_1)))

\{ \text{Lemma 1} \}

(P \parallel Q) \text{ sat } T

End of Proof.

Rules 1, 2, and 3 form the basis of our method for decomposing system requirements. Although relatively easy to prove, they encompass the conditions sufficient for justifying the method. The following describes and justifies the method using these rules:

1) Describe the architecture of the system as a composition of processes that can be arranged in a binary tree \( P_{i,j} \), for \( 0 \leq i < n \) and \( 0 \leq j < 2^n - 1 \). Define the alphabet of the system as the set of external communication channels, and only those communication channels, over which the system is permitted to communicate.

This step requires describing, as a binary tree, that part of the system architecture of interest. Each nonleaf vertex of the tree is a process representing the composition of its left son and its right son. Therefore, each subtree represents a subsystem of the entire system. The root of the tree \( P_{0,0} \) represents the system as a whole. The tree need not be complete, but if a vertex has one son then it must have both sons. Later refinement of the architecture specified in this step requires the straightforward application of Steps 1–7 to the new part of the architecture.

2) Specify the necessary requirements of the system in the form \( P_{0,0} \) sat \( R_{0,0} \).

The system specification is stated as a requirement \( R_{0,0} \) of \( P_{0,0} \). Subsequent decomposition will result in a requirement \( R_{i,j} \) for each process \( P_{i,j} \) of the system.

→ For \( 0 \leq i < n \) and \( 0 \leq j < 2^n - 1 \), let \( SR_{i,j} \) be the derived synchronization requirement for \( P_{i,j} \), defined as true initially. Traverse the tree in a breadth-first manner. At each nonleaf vertex \( P_{i,j} \), perform the following steps:

3) Define the alphabets of \( P_{i+1,2j} \) and \( P_{i+1,2j+1} \). Reduce the specification of the compose process \( P_{i,j} \) to the specification of the concurrent process \((P_{i+1,2j} \parallel P_{i+1,2j+1})\) by proving the Compose Restriction Condition.

\( R_{i,j}(tr_1) = R_{i,j}(tr_1 \uparrow \alpha P_{i,j}) \).

Definition 9 requires that the alphabets of the sons of \( P_{i,j}, P_{i+1,2j} \) and \( P_{i+1,2j+1} \), be defined such that their symmetric set difference equals the alphabet of \( P_{i,j} \). Under these restrictions, \( P_{i,j} \) sat \( R_{i,j} \) follows from the conditions

1) \((P_{i+1,2j} \parallel P_{i+1,2j+1})\) sat \( R_{i,j} \)
2) \forall tr_1. \text{ValidTrace}(tr_1, P_{i+1,2j} \parallel P_{i+1,2j+1})
   = 3tr_2. (ValidTrace(tr_2, P_{i+1,2j} \parallel P_{i+1,2j+1})
   \land \text{ValidTrace}(tr_1, \alpha P_{i+1,2j} \parallel \alpha P_{i+1,2j+1})
   \land (R_{i,j}(tr_2)
   \Rightarrow R_{i,j}(tr_2 \uparrow (\alpha P_{i+1,2j} \parallel \alpha P_{i+1,2j+1}))))

by Rule 2. The second condition follows from Rule 1 and the proof of the Compose Restriction Condition. The first will be decomposed in subsequent steps. In this, and subsequent steps, we require proof of stronger conditions than necessary, e.g., the Concurrent Restriction Condition, so as to proceed without any specific knowledge about the valid traces of the concurrent process \( P_{i+1,2j} \parallel P_{i+1,2j+1} \).

To ensure that the requirements found can be satisfied in subsequent steps. It is important to make the requirements as weak as possible. Although finding the weakest requirements may be difficult, the weaker the requirements found, the less that will be required of \( P_{i+1,2j} \) and \( P_{i+1,2j+1} \) to satisfy \( R_{i,j} \). This maximizes the amount of freedom in the design and implementation of the components while still meeting the system-level requirements defined.

5) Prove the Concurrent Restriction Condition,

\( R_{i+1,2j}(tr_1 \uparrow \alpha P_{i+1,2j}) = R_{i+1,2j}(tr_1) \)
\land \( R_{i+1,2j+1}(tr_1 \uparrow \alpha P_{i+1,2j+1}) = R_{i+1,2j+1}(tr_1) \).

If this proof fails, revise the requirements so that \( R_{i+1,2j} \) depends only on the events in the alphabet of \( P_{i+1,2j} \) and that \( R_{i+1,2j+1} \) depends only on events in the alphabet of \( P_{i+1,2j+1} \) and try again.

Once the component requirements are formulated, application of Rule 3 requires proving the condition

\text{ValidTrace}(tr_1, P_{i+1,2j} \parallel P_{i+1,2j+1})
\Rightarrow (R_{i+1,2j}(tr_1 \uparrow \alpha P_{i+1,2j}) = R_{i+1,2j}(tr_1))
\land (R_{i+1,2j+1}(tr_1 \uparrow \alpha P_{i+1,2j+1}) = R_{i+1,2j+1}(tr_1))

Proof of the Concurrent Restriction Condition is sufficient, although not necessary, to prove this requirement.
If it cannot be proven, \( R_{i+1,2,1} \) must be restricted so as to be independent of the events outside of \( aP_{r+1,2,1} \), and \( R_{i+1,2,1} \) must be restricted so as to be independent of the events outside of \( aP_{r+1,2,1} \).

6) Attempt to prove the Conjunction Condition,

\[ (R_{i+1,2,j}(tr1) \land R_{i+1,2,j+1}(tr1)) = R_{i,j}(tr1). \]

If successful, continue the tree traversal to the next vertex at step 3. Otherwise, specify the weakest condition \( C \) needed to complete the proof.

Application of Rule 3 requires proving that

\[ \text{ValidTrace}(tr1, P_{r+1,2,j} || P_{r+1,2,j+1}) \]

\[ = (R_{r+1,2,j}(tr1) \land R_{r+1,2,j+1}(tr1)) = R_{r,j}(tr1). \]

Proof of the Conjunction Condition is sufficient to satisfy this requirement. If it is provable, the decomposition at the current vertex is complete; in this case, \( P_{r,j} \) sat \( R_{r,j} \) provided that \( P_{r+1,2} \) sat \( R_{r+1,2} \) and \( P_{r+1,2,j+1} \) sat \( R_{r+1,2,j+1} \). Otherwise, Step 7 requires deriving a condition that allows completion of the proof. As in step 4, the weaker the condition found, the less that will be required of \( P_{r+1,2} \) and \( P_{r+1,2,j+1} \) to satisfy \( R_{r,j} \). The weakest condition must be found in order to guarantee the minimization of the set of synchronization requirements.

7) Describe \( C \) as a conjunction of simple conditions in conjunctive normal form. If no conjunct depends solely on the traces of either \( P_{r+1,2} \) or \( P_{r+1,2,j+1} \), then conjoint \( C \) to \( SR_{r,j} \) and continue the tree traversal to the next vertex at step 3. Otherwise, conjoint to \( R_{r+1,1} \), each conjunct of \( C \) that depends only on the traces of \( P_{r+1,1} \) (for \( k = 2j \) or \( k = 2j+1 \)) and continue at step 5.

Requirements that depend solely on the behavior of one component process are integrated into that process's specification whenever possible. Describing \( C \) as a conjunction of conditions, each captured in the simplest possible context, helps ensure that no requirement exists that could be stated as part of the specification of one of the component processes. This, in turn, maximizes the benefit gained from the decomposition process by minimizing the number and complexity of the synchronization requirements. Those requirements that depend on two or more component processes depend on specific knowledge about the valid traces of the concurrent process and are added to \( SR_{r,j} \).

D. Assumptions

The above method reduces the problem of formally verifying the requirements of a concurrent system into two separate, simpler problems: verifying that the system components meet their derived requirements and verifying that specific combinations of those components meet any derived synchronization requirements. Of course, the eventual implementation of the components of the system and their interconnections must be shown to satisfy the assumptions of the Trace Model of CSP on which our method depends. In summary, these are as follows:

1) The only way a process can communicate with another process executing concurrently is through CSP-like communication channels; no shared variables are permitted.

2) Exactly those external communication channels over which a process may pass data are included in its alphabet.

3) At most two processes of a system may communicate over a given channel; no broadcast capability exists. If the channel is external, exactly one process in the system must have the communication events associated with that channel in its alphabet. If the channel is internal, exactly two processes must have the communication events associated with that channel in their alphabets.

4) Communication over a given channel may take place in one direction only.

5) Any assumptions made of the communication media, e.g., in the definition of a transmission delay, must be validated.

Assuring the validity of these assumptions must take place throughout system development.

IV. An Example Decomposition

This section presents a description of the verified decomposition of an abstract voice transmitter using, as a guide, the iterative seven-step method discussed in Section III. The example, called the \( \mu \text{ASVT} \), is a simplified version of a voice terminal specified previously [2]. Although the notation \( P_{r,j} \) for processes, \( R_{r,j} \) for component requirements, and \( SR_{r,j} \) for synchronization requirements is convenient for describing the method for an arbitrary application, for a specific application this notation is cumbersome. Throughout the specification and decomposition of the \( \mu \text{ASVT} \), we use mnemonic names for processes so that, for example, the requirements for the process \( \mu \text{ASVT} \) are \( R_{\mu \text{ASVT}} \) and \( SR_{\mu \text{ASVT}} \).

A. The \( \mu \text{ASVT} \): An Informal Description

The \( \mu \text{ASVT} \) allows for the encrypted or plain text transmission of voice. It consists of three major components—the Voice Processor, the Modem Processor, and the Comsec Module. Fig. 2 illustrates the two external interfaces, Red Channel and Black Channel, and the three internal interfaces, Voice Modem, Voice Comsec, and Modem Comsec, through which voice transmissions may flow. The Red and the Black Channels are both analog interfaces. Conceptually, the Red Channel may be connected to telephone sets or intercoms; the Black Channel may be connected to radios or wireline appliances.

The \( \mu \text{ASVT} \) has a control panel allowing its users to choose between the cipher or the plain mode of operation. When in the cipher mode, the Voice Processor analyzes outgoing transmissions, the Comsec Module encrypts these transmissions, and the Modem Processor codes important bits and modulates the resulting bit stream. Transmissions in the plain mode are processed similarly except...
that the Comsec Module is bypassed so that information is transmitted through the Voice Modem channel without being encrypted. The terminal must be clear of all voice transmissions before a user may change the status of the control panel. The key used for encryption is pre-defined by the system and never changes. A user can receive transmissions from the μASVT only if he has access to a receiver that inverts the transformation performed by the terminal. This, of course, requires access to the decryption key. The details of the μASVT transformations have little or no bearing on the statement of its requirements and for ease of exposition are omitted.

The μASVT has one critical requirement, Red/Black Separation. More specifically, all information transmitted by the terminal when in the cipher mode of operation must be Black, encrypted, data. Thus, the only way for the μASVT to transmit Red, plain text, data is when the control panel is set in the plain mode. Because of its restricted functionality, the μASVT serves only as a demonstration of the method discussed in this paper—it is not intended as a real-world system. The practicality of this method and its demonstration on real systems is a topic of future research.

B. Specifying μASVT Architecture and Requirements

1) Describe the architecture of the μASVT as a composition of processes that can be arranged in a binary tree $P_{i,j}$ for $0 \leq i < n$ and $0 \leq j < 2^{n-1}$. Define the alphabet of the μASVT as the set of external communication channels, and only those communication channels, over which the μASVT is permitted to communicate.

The μASVT is described as the composition of three CSP processes, VP representing the Voice Processor, CM representing the Comsec Module, and MP representing the Modem Processor. Let μASVT be the CSP process representing the terminal as a whole. Then,

**Definition 12:** μASVT = $(VP \parallel (CM \parallel MP))$.

Fig. 3 illustrates the μASVT architecture as a binary tree of processes. This view requires that we introduce a new process name CMP, for Comsec Modem Processor, representing the composition of CM and MP.

Define $M$ to be the set of all possible messages communicable over the message channels. The alphabet of the μASVT is defined to be those communication events in which it is permitted to engage at its external interface:

$\alpha_{μASVT} = \{VPCtl.c, RedChan.m, BlackChan.m | c \in \{cipher, plain\}, m \in M\}$.

2) Specify the necessary requirements of the system in the form $μASVT \models R_{μASVT}$.

Specifying the μASVT's critical requirement, Red/Black Separation, requires that every message transmitted while in the cipher mode of operation be encrypted. The Restriction operator $\mid$ provides a mechanism for specifying properties about the values transmitted over particular channels in isolation from other communications. Specifying Red/Black separation requires a mechanism stronger than this to determine, given an arbitrary trace, the sequence of values communicated over a channel when in the cipher mode of operation. We define an abstract mechanism for doing this since it will be useful in the specification of the components as well as the μASVT.

The mechanism is based on a function Filter.

**Definition 14:**

Filter $(tr1, set1, req1)$

$= \begin{cases} < > & \text{if } tr1 = < > \text{ then } < > \\ \text{elsif } \text{Last}(tr1) \in set1 \land req1(\text{NonLast}(tr1)) & \text{then } \text{Append}(\text{Last}(tr1)) \\ & \text{Filter}(\text{NonLast}(tr1), set1, req1)) \\ \text{else } \text{Filter}(\text{NonLast}(tr1), set1, req1)) \end{cases}$

where $< >$ denotes the empty sequence, Last returns the last element of a sequence, NonLast returns all but the last element of a sequence, and Append appends an element to the end of a sequence. Intuitively, Filter returns the sequence of elements of trace $tr1$ that are in set $set1$ such that the sequence of events leading up to each element satisfies requirement $req1$.

*Note that no attempt is made to formally specify or decompose the noncritical functionality requirements in this paper.*
To use Filter, we define a function, CipherMode, that determines whether the terminal is in the cipher mode after an arbitrary sequence of communications takes place.

**Definition 15:**

\[
\text{CipherMode}(\text{ctlch1})(\text{trl}) = \begin{cases} 
\text{InitialMode} = \text{cipher} & \text{if trl} < > \text{ctlch1.c} \\
\text{ctlch1.c} & \text{else}
\end{cases}
\]

where InitialMode is the position of the control panel on start-up and \(\text{ctlch1} \) is the control channel over which the cipher/plain mode signals are sent. Now, if \(\text{chl: M} \) denotes the set of all possible communications of values in \(\text{M} \) over channel \(\text{chl} \),

\[
\text{Filter}(\text{trl}, \text{chl: M}, \text{CipherMode}(\text{ctlch1})) \]

returns the sequence of communications over \(\text{chl} \) that occur in \(\text{trl} \) either 1) when the last value sent over control channel \(\text{ctlch1} \) was cipher, or 2) if there is no such value, when InitialMode is cipher.

Red/Black separation requires that every message output over BlackChan correspond to the per transformation, including an encryption, of a message input over RedChan. Let Key be the key used by the \(\mu\text{ASVT} \) for encryption of voice and the functions Analyze, Encrypt, and Modulate be the functions representing the transformations performed by the Voice Processor, Comsec Module, and Modem Processor, respectively. Using the Filter function as described above, Red/Black separation is captured in the specification of the \(\mu\text{ASVT} \):

**Specifcation 1:** \(\mu\text{ASVT} \) sat \(\text{R}_{\mu\text{ASVT}}(\text{trl}) \)

\[
\text{R}_{\mu\text{ASVT}}(\text{trl}) = \forall m1. \text{(BlackChan.m1 in Filter}(\text{trl}, \text{BlackChan: M}, \text{CipherMode}(\text{VPCtl})) \rightarrow \exists m2. \text{(RedChan.m2 in Filter}(\text{trl}, \text{RedChan: M}, \text{CipherMode}(\text{VPCtl})) \wedge \text{Modulate}(\text{Encrypt}(\text{Analyze}(m2), \text{Key}) = m1))
\]

where \(\text{in} \) is the sequence membership symbol.

**C. Decomposing \(\mu\text{ASVT} \) Requirements: The First Attempt**

The top level CSP specification of the \(\mu\text{ASVT} \) composes VP with \((\text{CM} \parallel \text{MP}) \). Although a full analysis requires a complete traversal of the tree illustrated in Fig. 3, the following describes only the decomposition of \(\mu\text{ASVT} \) into two components, VP and CMP, where

**Definition 16:** \(\text{CMP} = (\text{CM} \parallel \text{MP}) \);

the decomposition of CMP proceeds similarly. Section IV-E describes the final results of the full decomposition of the \(\mu\text{ASVT} \). The Appendix to this paper presents an example CSP implementation of VP, CM, and MP.

3) Define the alphabets of VP and CMP. Reduce the specification of the compose process \(\mu\text{ASVT} \) to the specification of the concurrent process \((\text{VP} \parallel \text{CMP}) \) by proving the **Compose Restriction Condition**,

\[
\text{R}_{\mu\text{ASVT}}(\text{trl}) = \text{R}_{\mu\text{ASVT}}(\text{trl} \parallel \alpha\mu\text{ASVT}).
\]

The alphabets of the \(\mu\text{ASVT} \) component processes are defined as

**Definitions 17:**

\[
\alpha\text{VP} = \{\text{VPCtl.c, CMCtl.c, RedChan.m, VoiceComsec.m, VoiceModem.m | c} \in \{\text{cipher, plain} \}, m \in \text{M}\}
\]

\[
\alpha\text{CM} = \{\text{CMCtl.c, MPCtl.c, VoiceComsec.m, ModemComsec.m | c} \in \{\text{cipher, plain} \}, m \in \text{M}\}, and
\]

\[
\alpha\text{MP} = \{\text{MPCtl.c, BlackChan.m, ModemComsec.m, VoiceModem.m | c} \in \{\text{cipher, plain} \}, m \in \text{M}\}.
\]

CM is notified of changes in the control panel by VP through the CMCtl channel. Likewise, MP is notified of changes in the control panel by CM through the MPCtl channel. The alphabet of CMP follows from Definition 9 and Definitions 17.

**Lemma 2:** \(\alpha\text{CMP} = (\alpha\text{CM} + \alpha\text{MP}) \).

The CSP process architecture for the \(\mu\text{ASVT} \) is shown in Fig. 4.

Proof of the Compose Restriction Condition requires showing that the truth of \(\text{R}_{\mu\text{ASVT}} \) depends only on the occurrence of events in \(\mu\text{ASVT} \)’s alphabet. Towards this goal, let \(\leq \) be the trace prefix relation such that, for example, \(tr2 \leq tr1 \) states that \(tr2 \) is a prefix of \(tr1 \). The following lemma states sufficient conditions for proving that the value returned by Filter depends only on some restricted set of events:

**Lemma 3:**

\[
\left(\begin{array}{c}
\text{set1} \subseteq \text{set2} \\
\wedge (\forall tr2. tr2 \leq tr1) \\
\Rightarrow (\text{Filter}(tr1 \parallel \text{set2}, \text{set1}, \text{req1}))
\end{array} \right)
\]

**Proof:** (by mathematical induction on the length of \(tr1 \))

**Base Case:** (definition of \(\parallel\))

\[
\text{Filter}(\text{trl} \parallel \text{set2}, \text{set1}, \text{req1}) = \text{Filter}(\text{trl}, \text{set1}, \text{req1})
\]
**Induction Step:**

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \ tr_1) \Rightarrow (\text{req} \ (\tr_2) = \text{req} \ (\tr_2 \ \set 2)) \\
& \wedge (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1) \Rightarrow (\text{req} \ (\tr_2) = \text{req} \ (\tr_2 \ \set 2))) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

**Case:** \( \text{Last}(\tr_1) \in \set 2 \) \{definition of \( \uparrow \) \}

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1)) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

**Case:** \( \text{Last}(\tr_1) \notin \set 2 \) \{definition of \( \uparrow \) \}

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1)) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

End of Proof.

Since the only place that \( \tr_1 \) is referenced in Specification 1 of \( R_{\mu ASVT} \) is as a parameter to Filter, the Compose Restriction Condition follows from Theorem 1 below. We state and prove this theorem after proving a useful lemma.

**Lemma 4:** \( \{ ctlch1.cipher, ctlch1.plain \} \subseteq \set 1 \Rightarrow \text{CipherMode} (ctlch1)(tr1 \ \uparrow \ \set 1) = \text{CipherMode} (ctlch1)(tr1) \)

**Proof:** \{mathematical induction on the length of \( tr1 \) \}

**Base Case:** \{definition of \( \uparrow \) \}

\[
\begin{align*}
& \text{CipherMode} (ctlch1)(< \uparrow \ \set 1) \\
& = \text{CipherMode} (ctlch1)(< >)
\end{align*}
\]

**Induction Step:**

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1)) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

**Case:** \( \text{Last}(\tr_1) \notin \set 1 \)

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1)) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

**Case:** \( \text{Last}(\tr_1) \in \set 1 \)

\[
\begin{align*}
& (\forall \ tr_2. \ tr_2 \leq \text{NonLast} (\tr_1)) \\
& \Rightarrow (\text{Filter} (\text{NonLast} (\tr_1) \ \set 2, \ \set 1, \ \text{req} 1) = \text{Filter} (\text{NonLast} (\tr_1), \ \set 1, \ \text{req} 1))
\end{align*}
\]

End of Proof.

**Theorem 1:**

\[
\begin{align*}
& (\text{Filter} (\tr_1, \ BlackChan:M, \ \text{CipherMode} (\text{VPCtl}))) \\
& = \text{Filter} (\tr_1, \ \alpha_{\mu ASVT}. \ BlackChan:M, \ \text{CipherMode} (\text{VPCtl})))
\end{align*}
\]
\[ (\text{Filter}(tr1, \text{RedChan}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1 \uparrow \alpha_\mu \text{ASVT}, \text{RedChan}:M, \text{CipherMode}(\text{VPCtl}))) \]

**Proof:** We prove the first conjunct; the proof of the second conjunct proceeds similarly. By Lemma 3, 
\[ (\text{BlackChan}:M \leq \alpha_\mu \text{ASVT} \land (\forall tr2. tr2 \leq tr1) \land (\text{Filter}(tr1 \uparrow \alpha_\mu \text{ASVT}, \text{BlackChan}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1, \text{BlackChan}:M, \text{CipherMode}(\text{VPCtl}))) \] 
\[ \to \{\text{Definition 13 of } \alpha_\mu \text{ASVT}, \text{Lemma 4}\} \]

\[ \text{Filter}(tr1 \uparrow \alpha_\mu \text{ASVT}, \text{BlackChan}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1, \text{BlackChan}:M, \text{CipherMode}(\text{VPCtl})) \]

**End of Proof.**

4) Derive requirements \( R_{VP} \) for VP and \( R_{CMP} \) for CMP with the goal of proving \( VP \parallel CMP \) satisfies \( R_{\mu \text{ASVT}} \). \( R_{\mu \text{ASVT}} \) states that every message transmitted by the terminal when in the cipher mode is a proper transformation of some message received by the terminal. A natural requirement decomposition is to require that, when in the cipher mode, every message transmitted by a component be a proper transformation of some message received by the component. A proper transformation for VP is to Analyze the message; a proper transformation for CMP is to Encrypt, using the encryption key Key, and then Modulate the message. This suggests the following specifications for VP and CMP, respectively:

**Specification 2:** VP sat \( R_{VP} \).

\[ R_{VP}(tr1) = \] 
\[ \forall m1. (\text{VoiceComsec}. m1 \in \text{Filter}(tr1, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl})) \land (\exists m2. (\text{RedChan}. m2 \in \text{Filter}(tr1, \text{RedChan}:M, \text{CipherMode}(\text{VPCtl})) \land \text{Analyze}(m2) = m1)) \]

**Specification 3:** CMP sat \( R_{CMP} \).

\[ R_{CMP}(tr1) = \] 
\[ \forall m1. (\text{BlackChan}. m1 \in \text{Filter}(tr1, \text{BlackChan}:M, \text{CipherMode}(\text{CMCtl})) \land (\exists m2. (\text{VoiceComsec}. m2 \in \text{Filter}(tr1, \text{VoiceComsec}:M, \text{CipherMode}(\text{CMCtl})) \land \text{Modulate}(\text{Encrypt}(m2, \text{Key})) = m1)) \]

\[ \begin{align*} \text{R}_{VP}(tr1) & \land \text{R}_{ CMP}(tr1) = \text{R}_{\mu \text{ASVT}}(tr1). \end{align*} \]

Notice that communications over \( \text{VPCtl} \) are used to determine whether the terminal is in the cipher mode for VP, whereas communications over \( \text{CMCtl} \) are used for CMP.

5) Prove the **Concurrent Restriction Condition**, 
\[ R_{VP}(tr1 \uparrow \alpha_{VP}) = R_{VP}(tr1) \land R_{CMP}(tr1 \uparrow \alpha_{CMP}) = R_{CMP}(tr1). \]

If this proof fails, revise the requirements so that \( R_{VP} \) depends only on the events in the alphabet of VP and that \( R_{CMP} \) depends only on events in the alphabet of CMP and try again.

We prove the first conjunct of the Concurrent Restriction Condition; the proof of the second conjunct proceeds similarly. Since the only place that \( tr1 \) is referenced in Specification 2 of VP is as a parameter to Filter, the first conjunct of the Concurrent Restriction Condition follows from Theorem 2 below. This proof has the same general structure as the proof of Theorem 1.

**Theorem 2:**
\[ (\text{Filter}(tr1, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1 \uparrow \alpha_{VP}, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl}))) \]
\[ \land (\text{Filter}(tr1, \text{RedChan}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1 \uparrow \alpha_{VP}, \text{RedChan}:M, \text{CipherMode}(\text{VPCtl}))) \]

**Proof:** We prove the first conjunct; the proof of the second conjunct proceeds similarly. By Lemma 3,
\[ (\text{VoiceComsec}:M \leq \alpha_\mu \text{ASVT} \land (\forall tr2. tr2 \leq tr1) \land (\text{Filter}(tr1 \uparrow \alpha_\mu \text{ASVT}, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl}))) \]
\[ \to \{\text{Lemma 4, Definitions 17 of } \alpha_\mu \text{ASVT}\} \]
\[ \text{Filter}(tr1 \uparrow \alpha_\mu \text{ASVT}, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl})) = \text{Filter}(tr1, \text{VoiceComsec}:M, \text{CipherMode}(\text{VPCtl})) \]

**End of Proof.**

6) Attempt to prove the **Conjunction Condition**, 
\[ (R_{VP}(tr1) \land R_{CMP}(tr1)) = R_{\mu \text{ASVT}}(tr1). \]

If successful, continue the tree traversal to the next vertex at step 3. Otherwise, specify the weakest condition \( C \) needed to complete the proof.

This step is difficult, partly due to the fact that the proof is dependent on the proper synchronization of VP and CMP. We derive the necessary synchronization require-
ment through an attempt to prove the Conjunction Condition. Analyzing \( R_{VP} \) and \( R_{CMC} \) of Specifications 2 and 3 shows that the Conjunction Condition can be deduced from the following theorem:

**Theorem 3:**

\[
\begin{align*}
\text{Theorem 3:} & \quad \text{Filter}(rrl, \text{VoiceComsec}:M, \text{CipherMode}(VPCtl))
\end{align*}
\]

\[
\begin{align*}
&= \text{Filter}(r1, \text{VoiceComsec}:M, \text{CipherMode}(CMCtl)) \\
&\wedge \text{Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(VPCtl)) \\
&= \text{Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(CMCtl)) \\
&\Rightarrow (R_{VP}(r1) \wedge R_{CMC}(r1)) = R_{\mu ASVT}(r1)
\end{align*}
\]

**Proof:**

\[
\begin{align*}
&\text{Filter}(r1, \text{VoiceComsec}:M, \text{CipherMode}(VPCtl)) \\
&= \text{Filter}(r1, \text{VoiceComsec}:M, \text{CipherMode}(CMCtl)) \\
&\wedge \text{Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(VPCtl)) \\
&= \text{Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(CMCtl)) \\
&\wedge R_{VP}(r1) \wedge R_{CMC}(r1)
\end{align*}
\]

\[
\Rightarrow \{\text{Specification 2 of } R_{VP} \text{ and Specification 3 of } R_{CMC}\}
\]

\[
\forall \ m1. \ (\text{VoiceComsec.} \ m1 \ \text{in Filter}(r1, \text{VoiceComsec}:M, \text{CipherMode}(VPCtl))
\]

\[
\Rightarrow \exists m2. \ (\text{RedChan.} \ m2 \ \text{in Filter}(r1, \text{RedChan}:M, \text{CipherMode}(VPCtl))
\]

\[
\wedge \text{Analyze}(m2) = m1)
\]

\[
\wedge \forall \ m1. \ (\text{BlackChan.} \ m1 \ \text{in Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(VPCtl))
\]

\[
\Rightarrow \exists m2. \ (\text{VoiceComsec.} \ m2
\]

\[
\text{in Filter} \cdot (r1, \text{VoiceComsec}:M, \text{CipherMode}(VPCtl))
\]

\[
\wedge \text{Modulate}(\text{Encrypt}(m2, \text{Key})) = m1)
\]

\[
\Rightarrow \{\text{Calculus}\}
\]

\[
\forall \ m1. \ (\text{BlackChan.} \ m1 \ \text{in Filter}(r1, \text{BlackChan}:M, \text{CipherMode}(VPCtl))
\]

\[
\Rightarrow \exists m2. \ (\text{RedChan.} \ m2 \ \text{in Filter}(r1, \text{RedChan}:M, \text{CipherMode}(VPCtl))
\]

\[
\wedge \text{Modulate}(\text{Encrypt}(m2, \text{Key})) = m1)
\]

\[
\Rightarrow \{\text{Specification 1 of } R_{\mu ASVT}\}
\]

\[
R_{\mu ASVT}(r1)
\]

**End of Proof.**

Theorem 3 states sufficient conditions for proving the Conjunction Condition. Provided these conditions are true, when the terminal is in the cipher mode, the origination of a message transmitted through BlackChan can be traced through CMP from VoiceComsec. Likewise, the origination of a message transmitted through VoiceComsec can be traced through VP from RedChan. Unfortunately we cannot prove the antecedent of Theorem 3 without additional information about the valid traces of the \( \mu ASVT \). Therefore, set \( C \) equal to this antecedent. 7)

Recall that CipherMode\( (ctlchl) \) is true if and only if the terminal is in the cipher mode after engaging in the sequence of events \( r1 \). The mode is cipher if and only if either the most recent transmission over \( ctlchl \) was cipher or, if there is no such transmission, the initial mode is cipher. Since VPCtl and CMCtl are distinct channels, there is no way to determine the truth of \( C \) from only the definition of CipherMode.

The truth of the first conjunct of \( C \) depends only on communication over VoiceComsec, VPCtl, and CMCtl; the truth of the second conjunct depends on communication over BlackChan, VPCtl, and CMCtl. Since VP must participate in any events of \( r1 \) that occur in its alphabet and since, by Definitions 17 of \( \alpha VP \),

\[\text{Note that since this instantiation of } C \text{ is not necessary to establish the truth of the Conjunction Condition, we cannot guarantee that the set of synchronization requirements derived is minimal.}\]
Lemma 5: 
\[
\{ \text{VoiceComsec.m, VPCtl.c, CMCtl.c} \\
| c \in \{\text{cipher, plain}\}, m \in M \} \subseteq \alpha_{VP},
\]
the truth of the first conjunct depends solely on the traces of VP. Defining R_{VP} as this conjunct results in Definition 18:
\[
R_{VP}(trl) = \left\{ \begin{array}{ll}
(trl \neq < > \\
\text{Last}(trl) \in \text{VoiceComsec:M})
\end{array} \right.
\Rightarrow (\text{CipherMode}(\text{VPCtl})(trl) \\
= \text{CipherMode}(\text{CMCtl})(trl))
\]

Therefore, we can conjoin R_{VP} to R_{VP}. Unfortunately, the second conjunct of C does not depend on the traces of a single component. By Definitions 17, communications over BlackChan and VPCtl do not occur in the alphabet of any single component. Nevertheless, since the specification of VP has been extended, we must go back to step 5 and reprove the Concurrent Restriction Condition for VP.

D. Justifying the Revised Decomposition

During the first attempt to decompose the requirement of \( \mu \text{ASVT} \), we proved the Concurrent Restriction Condition for the original definitions of R_{CMp} and R_{VP}. Conjoining R_{VP} to R_{VP}, requires proof that R_{VP} depends only on the events in the alphabet of VP to reestablish the truth of the Concurrent Restriction Condition:
\[
\text{Theorem 4: } R_{VP}(trl \uparrow \alpha_{VP}) \Rightarrow R_{VP}(trl).
\]

Proof:
\[
R_{VP}(trl \uparrow \alpha_{VP})
\Rightarrow \{ \text{Definition 18 of } R_{VP} \}
\Rightarrow \{ \text{Definition 17 of } \alpha_{VP} \}
\Rightarrow \{ \text{Lemma 4, Definitions 17 of } \alpha_{VP} \}
\Rightarrow \{ \text{Definition 18 of } R_{VP} \}
\Rightarrow R_{VP}(trl)
\]

End of Proof.

Since R_{VP} has been modified, the next step is to reprove the Conjunction Condition. Fortunately, these proofs proceed exactly as before except, this time, the proof depends only on the second conjunct of Condition 1. Since it does not depend on any one component of the \( \mu \text{ASVT} \), the decomposition at this vertex is finished.

E. Analyzing the Final Results

Decomposing Specification 1 of \( \mu \text{ASVT} \) results in a specification for each component and a number of synchronization requirements on multiple components. In summary, the derived component specifications are as follows.

Specification 4: VP sat R_{VP}.
\[
R_{VP}(trl) = (\forall m1. \{\text{VoiceComsec.m in Filter \cdot (trl, VoiceComsec:M, CipherMode(\text{VPCtl}))} \\
\exists m2. \{\text{RedChan.m2 in Filter \cdot (trl, RedChan.m2 in Filter (trl, VoiceComsec:M, CipherMode(\text{VPCtl}))} \\
\text{Analyze(m2, Key) = m1}\})
\]

Specification 5: CM sat R_{CM}.
\[
R_{CM}(trl) = (\forall m1. \{\text{ModemComsec.m in Filter \cdot (trl, ModemComsec:M, CipherMode(\text{CMCtl}))} \\
\exists m2. \{\text{VoiceComsec.m2 in Filter \cdot (trl, VoiceComsec:M, CipherMode(\text{CMCtl}))} \\
\text{Encrypt(m2, Key) = m1}\})
\]

Specification 6: MP sat R_{MP}.
\[
R_{MP}(trl) = (\forall m1. \{\text{BlackChan.m in Filter \cdot (trl, BlackChan:M, CipherMode(\text{MPCtl}))}
\]
The first conjunct of RVP in Specification 4, the first conjunct of RC in Specification 5, and all of RMP in Specification 6 represent the proper transformation of messages transmitted through each component when in the cipher mode. Each processor uses its own control channel for determining whether the terminal is in the cipher mode. At the system level, however, only VPCtl is used to determine whether the system is in the cipher mode. As suggested by the analysis of Sections IV-C and IV-D this fact requires that VP and MP satisfy additional conditions, defined by the second conjuncts of RVP and RC in Specifications 4 and 5. These conditions were derived during the effort to prove the Conjunction Conditions during the first and second level decompositions.

The second conjuncts of RVP and RC are somewhat obscure. Their purpose is to help ensure that the position of the mode selector dial is seen consistently by all three processes. A notification of a change in mode for the component processes and the minimal set of synchronization requirements of the \( \mu \)ASVT as referenced in the introduction of this paper. Although it is outside the scope of this paper to prove the requirements described in this section, it is interesting to look at them in light of the example functional refinement presented in the Appendix to this paper. Through some analysis of these definitions, the truth of Specifications 4-6, and Requirements 1 and 2, becomes apparent. Proofs of the component requirements are relatively easy to formalize using the proof methods of the Trace Model of CSP. Unfortunately, however, when one tries to formalize the proofs of Requirements 1 and 2, the arguments explode into an extremely large number of cases. Future work requires determining methods for practically dealing with such proofs.

V. SUMMARY AND CONCLUSIONS

This paper describes and demonstrates a method for formalizing and decomposing critical requirements of systems using the Trace Model of CSP. The method proceeds iteratively until the appropriate requirements for the component processes and the minimal set of synchronization requirements are found. An extension to the CSP notation, involving process composition with hidden internal structure, promotes hierarchical system design and decomposition. A goal of the decomposition process is to minimize the number and complexity of the synchronization requirements since these are the most difficult to verify in later system development. We capture these requirements in the simplest possible context and ensure that each depends on the behavior of at least two component processes. The method described reduces the problem of assuring that the system meets its critical requirements to the simpler, although possibly nontrivial, problem that each component meets its derived requirements and the system satisfies the derived synchronization requirements.

Experience with the \( \mu \)ASVT decomposition confirms our belief that the CSP Trace Model is a valuable formalism for specifying and reasoning about systems and their requirements. Although the Trace Model is restricted to reasoning about the safety of nondivergent processes, within this domain the power of trace theory eases the description of critical properties over less sophisticated models while avoiding the complexities of the more sophisticated models. For example, the use of separate channel histories, as in the Counter Model and Gypsy,
complicates the statement of relationships between the communication histories of different channels at specific times during a process's execution. The μASVT's Red/Black separation requirement is difficult, if not impossible, to specify within this paradigm due to the need to determine the mode in which the terminal is operating when messages are transmitted; this determination requires finding out the last value transmitted over VPCtl for each message transmitted over BlackChan. The Trace Model allows the statement of such properties by describing process behavior in terms of interleavings of distinct channel histories. The ability to decompose critical liveness properties of potentially diverging processes using the more sophisticated models is an area of future research.

Stages of development, subsequent to system decomposition, require methods to verify sequential component and synchronization requirements. Verifying properties of sequential components does not present any inherent difficulties. Unfortunately, we have not yet found a practical approach applicable to nontrivial systems for formally proving the derived synchronization requirements. Although proof methods for reasoning about global properties exist within the CSP Trace Model, they are intractable for most real-world systems. This problem is not unique to the model used here; the other models of CSP discussed, as well as the proof methods of Gypsy, exhibit these same difficulties. In general, if it is impossible to derive component process requirements that are sufficient to prove the system requirements, reasoning about global properties is required. This condition arose in the decomposition of even the relatively simple μASVT example as discussed in Section IV.

The EHDM verification system [8] is proving useful for mechanically checking the proofs required of the decomposition process. We have encoded within the logic of EHDM the relevant portion of the CSP Trace Model allowing system architecture and requirements to be specified. Directed by an informal proof outline derived separately, we can guide EHDM through the seven-step method. When applied to the μASVT decomposition, EHDM supplied many of the low-level proof details, e.g., variable instantiations, and, at times, caught errors in the informal proof. The final EHDM proof provided necessary information for revising and refining the informal proofs. This paper presents a subset of these revised proofs.

Although the use of EHDM is beneficial towards checking the integrity of a decomposition, it is clear that the availability of tools specifically oriented towards specifying and verifying systems involving concurrency are needed to facilitate the decomposition process. Verification systems with much of the concurrency model built in would allow the automation of large portions of the proof process. Alternatively, verification systems that allow the user to define automatically invoked decision procedures would permit developers to "program" the theorem prover with CSP-specific proof strategies.

Experiences such as those described in this paper may prove beneficial for identifying useful characteristics of verification tools and helpful decision procedures for verifying concurrent systems.

The decomposition method proposed in this paper applies to systems with critical requirements implemented in either hardware or combinations of software and hardware. Executable languages supporting concurrency [43]-[45] may be useful for prototyping systems described in CSP [46]-[48], for testing the validity of formalisms specified in the Trace Model [49], and for continuing the formal verification to lower levels of system design and implementation [43]. Others are investigating the incorporation of CSP into formal methods that do not explicitly deal with concurrency such as the Z specification language [4]. If decompositions are taken to a sequential level, more conventional procedural languages may be used to implement systems in software or as design languages for hardware. Formal reasoning can proceed in such languages if a trace semantics can derived for programs written, e.g., [39]. Since our method focuses on requirement decomposition rather than physical decomposition, it can be used during structural hardware design to decompose requirements for hardware in a top-down fashion as primitive hardware elements are interconnected to realize higher-level blocks in a bottom-up fashion.

Decomposing requirements using formal methods enhances the role of testing and simulation in the design of trustworthy systems. A designer tests a system description to determine whether it meets the requirements. Complex systems lead to complex test suites. The greater the complexity of the test suite, the more difficult it is to determine whether the system meets its requirements. A requirement decomposition allows the designer to break the test suite for a system into smaller, less complex test suites for its components. This simplifies the requirements that must be assured and decreases the number of execution paths that must be traversed. Simplifying the testing process in this way increases the probability of finding problems that exist in the design. Of course, each synchronization requirement derived from a requirement decomposition requires testing of those components on which it depends.

Further work is proceeding along two complementary lines: the application of the decomposition method to a security-critical portion of a full-scale communication system and the extension of the method to handle the verification of the component-level and synchronization requirements. From derived component-level requirements, we expect to generate and verify implementations of the components using some formally verifiable programming language such as Gypsy [29] or m-Verdi [50]. Unfortunately, due to the immature state of formally verifiable languages, the actual source code will likely be written in a more conventional language with better run-time support. Nevertheless, the verified implementations will act as low-level functional descriptions from which to derive the source code. Using strict coding conventions and re-
view procedures this derivation should be fairly straightforward so as to preserve the formal verification accomplished at the higher levels.

Appendix

A CSP IMPLEMENTATION OF THE µASV T

The following functional description reflects the operation of the terminal as discussed in Section IV. Note that the CSP recursive process µX.F(X) represents the process X such that X = F(X). The choice process (e1 → P) | (e2 → Q) represents a process that first engages in either e1 or e2. If it engages in e1 then it, subsequently, behaves like P; if it engages in e2 then it, subsequently, behaves like Q.

VP

= VPCtl ? mode → µY.(CMCtl ! mode
→ if mode = cipher
then µX. ((RedChan ? msg
→ VoiceComsec ! Analyze(msg)
→ X) | VPCtl ? mode → Y)
else µX. ((RedChan ? msg
→ VoiceModem ! Analyze(msg)
→ X) | VPCtl ? mode → Y))

CM

= CMCtl ? mode → µY.(MPCtl ! mode
→ if mode = cipher
then µX. ((VoiceComsec ? msg
→ ModemComsec ! Encrypt(msg, Key)
→ X)
| CMCtl ? mode → Y)
else CMCtl ? mode → Y)

MP

= MPCtl ? mode → µY.(if mode = cipher
then µX. ((ModemComsec ? msg
→ BlackChan ! Module(msg)
→ X | MPCtl ? mode → Y))
else µX. ((VoiceModem ? msg
→ BlackChan ! Module(msg)
→ X) | MPCtl ? mode → Y)

Acknowledgment

The author is grateful to C. Landwehr, J. McLean, C. Payne, and J. Gray of NRL; to K. Hayman of the Australian DSTO; and to the referees, for helpful comments on earlier drafts of this paper.

References


Andrew P. Moore received the B.A. degree in mathematics and computer science from the College of Wooster, Wooster, OH, in 1984 and the M.A. degree in computer science from Duke University, Durham, NC, in 1986. He is a Computer Scientist at the Naval Research Laboratory's Center for Secure Information Technology where he is currently principle investigator for a project involving the use of formal methods to prove security properties of hardware designs. He previously served as principle investigator for a research project involving the application of formal software verification techniques and tools to assure the security of communication system software. His research interests include formal specification and verification, trusted system development, and computer and communications security. Prior to joining NRL, he worked for Research Triangle Institute of North Carolina and Software Architectures and Engineering of Arlington, VA.

Mr. Moore is a member of the Association for Computing Machinery and the IEEE Computer Society.