Verification of Consistency Between Concurrent Program Designs and Their Requirements*

Marsha Chechik John Gannon

{chechik, gannon}@cs.umd.edu
Computer Science Department
University of Maryland
College Park, MD 20742
Tel. (301) 405-2671

Abstract

Writing requirements in a formal notation allows automatic assessment of such properties as ambiguity, consistency, and completeness. However, verifying that the properties expressed in the requirements are preserved in an implementation remains difficult. In our earlier work we described a technique for analyzing consistency of detailed program designs with their requirements. To ensure that our methods scale up to realistic systems, we need to develop compositional approaches. This paper describes a first step in this direction: a technique for analyzing concurrent program designs.

We present a language for specifying detailed designs of concurrent programs and an analysis tool, Analyzer, which uses this language to build a finite-state abstraction of the design. This abstraction is compared with properties derived from the set of requirements to determine if the former is consistent with the latter. Some restrictions on the model and on the specification language enable Analyzer to verify global user-specified properties effectively. After a design is verified to be consistent with its requirements, an implementation can be written around the design constructs to reduce the likelihood of implementation errors.

1. Introduction

The use of formal notations in specifying requirements can reduce their ambiguity and imprecision and enable computer-aided checking for inconsistency and incompleteness. For example, several requirements notations describe a system as a set of concurrently executing state machines ([2, 17, 21]) that respond to events in their environments. Formal notations can also be used to check that required behavior is preserved in detailed designs and implementations.

Designs are often expressed in special program design languages (PDLs)[4]. When a design is inspected, the reviewers seek to establish that it meets the requirements, and that it is complete and self-consistent[20]. In earlier work [5, 6], we defined a notation which can be used as a PDL. The PDL's outer syntax is comprised of C control constructs, and its inner syntax is a set of special comments, called *annotations*, which describe local properties of requirements variables' values. We have also developed a prototype tool called Analyzer which builds a finite-state abstraction of a sequential program design and automatically checks that this abstraction is consistent with properties derived from its requirements. The inputs to our tool consist of a requirements specification written in a formal notation and a design expressed in this PDL. We say that a design is consistent with its requirements if it implements those and only those transitions specified in the requirements and preserves all of the requirements' safety properties. Analyzer automatically generates a set of global checks from the requirements which are equivalent to temporal logic[9] formulas capturing this notion of consistency, builds a finite-state abstraction of the design using dataflow analysis techniques, and uses a special-purpose model checking algorithm to determine if the formulas hold in the abstraction. After a design is verified to be consistent with the requirements, an implementation can be developed from the design constructs. This process reduces the likelihood of implementation errors.

To ensure that our approach can scale to realistic systems, we have to develop compositional techniques enabling us to reduce the size of each state, both for building the model

*This work was supported by the Air Force Office of Scientific Research under contract F49620-93-1-0034

and for verifying properties. Our requirements notation naturally models concurrently executing processes. If we place certain restrictions on the channels of communication between these processes, we can analyze the system compositionally. Thus, we are interested in extending our analysis methods to designs of concurrent programs. This paper reviews our analysis techniques, discusses our synchronization paradigm, and presents extensions to Analyzer which enable us to verify the consistency of concurrent designs with their requirements.

Most of the previous approaches to analysis of concurrent programs have generally abstracted away data values and their effects on control flow to concentrate on task interactions (e.g., [10, 11, 22, 12, 25, 23]). We perform dataflow analysis on values of requirements variables and include them into the property verification. The goal of our technique is to avoid building a global reachability graph.

The rest of the paper is organized as follows: in Section 2, we use a simple example to introduce the requirements and program design languages and discuss the types of properties that we verify. Section 3 discusses how to compute the states that occur at each point in a design. In Section 4, we present techniques employed in Analyzer for verifying automatically-generated and user-defined safety properties. Section 5 discusses our conclusions and future work.

2. Requirements and Design Notations and the Notion of Consistency

Analyzer receives two inputs: a requirements specification and a detailed design written in our PDL. In this section, we present the requirements and the program design languages and define what we mean by consistency between a design and its requirements.

2.1. Requirements

The Software Cost Reduction (SCR) requirements notation[2, 15, 17] was developed by a research group at the Naval Research Laboratory. An SCR requirements specification models a system as a set of event-driven, state-transition machines. The machines' environment is abstracted as a set of *monitored state variables* and *controlled state variables* [24, 26]. A *state* is a mapping of variables to values. Changes to monitored variables may cause the system to change its mode or to alter the values of its controlled variables. A *mode class* defines a set of states, called *modes*, that partition the monitored environment's state space. One mode of each mode class is designated as the *initial* mode. Assumptions about the initial state of the environment are specified with the initial mode. Transitions between pairs of modes are activated by conditioned events. A *condition* is a predicate on monitored or mode class variables, and denotes an *event* when one of their values changes. For example, $@T(a)$ `WHEN` $[b]$ occurs if condition a changes value from False to True while condition b is True, or, formally (adopted from [16]),

$$@T(a)\ \texttt{WHEN}\ [b] = \sim a \wedge a\prime \wedge b,$$

where the unprimed version of condition a denotes a in the old state and the primed version denotes a in the new state, and $\sim$ and $\wedge$ indicate negation and conjunction, respectively. We refer to @T(a) and b as *Triggering* and *When* conditions, respectively. Similarly, $@F(a)$ occurs if condition a changes value from True to False. SCR has a formal underlying semantics [16] which permits creation of various CASE tools for creation and checking of SCR specifications[1].

Since conditions are predicates on monitored variables, we will assume that the system variables are boolean. Mode classes can typically be thought of an enumerated types whose values are modes of that mode class. Thus, we assume that the state space described by a requirements specification is finite.

One of the ways to create a system design is by implementing each mode class in a separate process. To be able to analyze each process in turn, we make certain restrictions on the original SCR model. Each mode class takes a set of *Inputs* (monitored variables) and produces *Outputs* (controlled variables). One mode class communicates with another by sending the values of some of its outputs to the other mode class which treats them as part of its set of inputs. Only some of the total inputs and outputs are used to communicate with the environment, as controlled or monitored variables (see Figure 1). A simple convention of using the same name for outputs of one process and inputs of another, identifies the variables used in communication. We restrict the model further by not allowing any inputs of a process to be among its outputs. Still, this model is more general than the one recently proposed in [16]. The latter assumes that there is a partial order of dependencies between variables and mode classes in every requirements document, and we make no such assumption.

We illustrate the SCR model using a simple example, abstracted from a Nuclear Control Rod system [19] to preserve communication problems. We call it a Mutual Exclusion system (MES). Assume that we want to specify a system where two identical processes have critical sections, and mutual exclusion is guaranteed by having a separate process, Manager, give out permissions to the other processes. The requirements for this system consist of three mode classes running in parallel: mode classes SubSys1 and SubSys2 describe two identical subsystems, and mode class

[1] We are using a slightly different (and at times simplified) version of SCR, and will point out some of the differences in this paper.

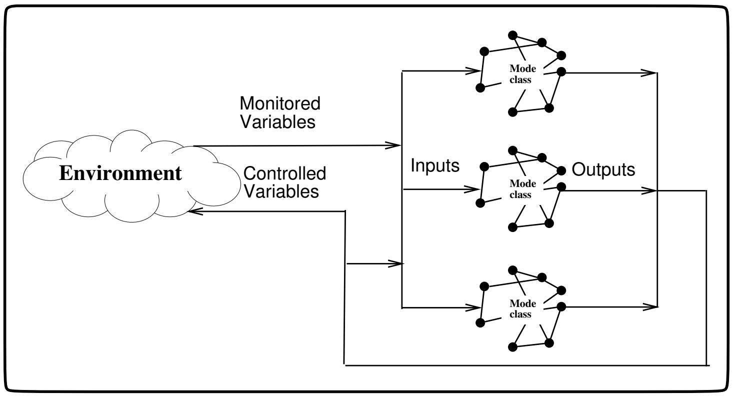

Figure 1. SCR requirements notation.

Current Mode	Time	In(Manager=Grant1)	Done	**New Mode**
S1Done	@T	–	–	S1Wait
S1Wait	–	@T	–	S1RecGrant
S1RecGrant	–	–	@T	S1Done

Initial: S1Done ($\sim$Time $\wedge$ $\sim$Done)

Table 1. Mode transition table for SubSys1.

Manager determines which subsystem has the permission to enter its critical section. So,

$$MSE = Manager \parallel SubSys1 \parallel SubSys2,$$

where || indicates a parallel composition of the processes. SCR requirements typically specify sequential systems. When each mode transition table is implemented in a separate process, race conditions can occur when tables are composed. We assume that subsystems cannot make simultaneous state changes, and this naturally gives us an interleaving model.

The subsystems communicate with the Manager by sending it a message (either requesting or returning the permission) and, in the case of a request, waiting until the permission is granted. Communication variables In(SubSys1=S1Wait), In(SubSys2=S2Wait), In(SubSys1=S1Done) and In(SubSys2=S2Done) are used by the Manager to determine which of the two processes is waiting to receive the permission. These variables are outputs of the respective subsystems and inputs to the Manager. In(Manager=Grant1) and In(Manager=Grant2) are used by the subsystems to determine which one is allowed to proceed; they are outputs of the Manager and inputs to the subsystems. All these variables are *state conditions*, i.e., conditions specifying that a particular mode class is in a particular mode. They become True when the system enters a corresponding state, and False when it leaves this state. A state condition is a simplified version of van Schouwen's InMode() function [26].

In addition, each subsystem monitors the variable Time to determine when it is time to ask for permission, and Done to determine when critical-section processing is complete. The requirements specification for mode classes SubSys1 and Manager are shown in Tables 1 and 2, respectively. The specification of each mode class's transition relation has tabular format. Each row in the table specifies a conditioned event that activates a transition from the mode on the left to the mode on the right. A table entry of "@T"(or "@F") under a column represents a triggering event for the condition represented by the variable labelling the column (the variable is changing value to True (or False)). A table entry of "t" (or "f") represents a `WHEN` condition. If the value of a condition does not affect the occurrence of a conditioned event, then the table entry is marked with a hyphen ("-"). For example, Manager transits from Stable to Grant1 (see Table 2) when subsystem 1 has requested a permission (In(SubSys1=S1Wait) changes value from False to True) while subsystem 2 is not waiting, and to Grant2 when subsystem 2 has requested a permission but subsystem 1 is not waiting. Both systems cannot request permission at the same time due to the interleaving model. Mode class Manager starts in mode Stable, and both subsystems start in mode Done.

The requirements designer may also specify a restricted set of system safety properties (see [6]). These properties can be local to one process or global to the whole system.

Current Mode	In(SubSys1 =S1Wait)	In(SubSys2 =S2Wait)	In(SubSys1 =S1Done)	In(SubSys2 =S2Done)	**New Mode**
Stable	@T	f	–	–	Grant1
	f	@T	–	–	Grant2
Grant1	–	–	@T	–	Stable
Grant2	–	–	–	@T	Stable

Initial: Stable (In(SubSys1=S1Done) $\wedge$ In(SubSys2=S2Done))

Table 2. Mode transition table for the Manager.

We assume that the requirements specification is correct, i.e., the specified properties have been verified with respect to the requirements (for example, using techniques in [3]). We defined a list of safety properties that we allow the user to specify. For example, $reach(f)$ indicates that the system should reach a state where f holds. For our example, we might want to prove properties that both grants are guaranteed to be given out at some point, i.e., the system can reach a state in which Manager=Grant1 holds, and a state in which Manager=Grant2 holds:

$$reach(Manager = Grant1)$$
$$reach(Manager = Grant2)$$

Typically reach properties are used to ensure that invariant properties are not satisfied vacuously. Another type of user-defined properties is an invariant, $inv(f)$, indicating that f holds in every node of the system. Mutual exclusion between subsystems, i.e. the two subsystems cannot be in RecGrant at the same time, can be expressed using inv property:

$$inv(\sim(SubSys1 = S1RecGrant \wedge SubSys2 = S2RecGrant))$$

Another type of properties is strict cause. $strcause(f_1, f_2)$, means that the next transition from each state in which f_1 holds is to a state in which f_2 holds. If we want to assure that the subsystems are able to return the grant, i.e., whenever the SubSys1 is in mode S1Done, the next transition of the system establishes a state in which Manager is Stable, we will express it using a $strcause$ property. A similar property is defined for subsystem 2:

$$strcause(SubSys1 = S1Done, Manager = Stable)$$
$$strcause(SubSys2 = S2Done, Manager = Stable)$$

The types of properties which we allow the user to specify (i.e., reach, inv, strcause) frequently appear in requirements documents. However, requirements of realistic size and complexity might contain other global properties which cannot be expressed using just these assertions.

2.2. Consistency

SCR requirements specify all transitions that the system is allowed to take. We want to make sure that the design implements these and only these transitions, and therefore give the following definition of consistency:

Definition 2.1 *A detailed design is consistent with its requirements if it implements all state transitions specified in the requirements, does not implement any state transitions which are not specified in the requirements, and satisfies all user-defined safety properties.*

A set of properties capturing this notion of consistency can be obtained by composing rows and columns of SCR tables. For example, we have a property asserting that there exists a state in which Manager is in mode Grant1 and an event @T(In(SubSys1=S1Done)) occurs, after which Manager is in mode Stable in one of its next states. In CTL[9], a computational tree logic, a branching-time logic which permits explicit quantification over all possible futures, this property is expressed as

$$EF(Manager = Grant1 \ \wedge \ @T(In(SubSys1 = S1Done)) \wedge EX(Manager = Stable))$$

Such properties ensure that all transitions specified in the requirements appear in the design, capturing the first part of our notion of consistency[2]. We call them "all legal transitions" (ALT) properties. We generate one ALT property for every row of transition tables for modeclasses and controlled variables.

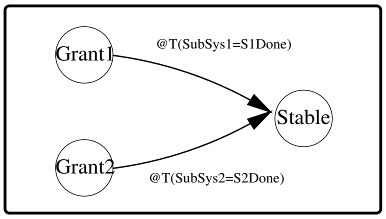

Figure 2. Transitions into mode Stable.

Another property asserts that if Manager is in mode Stable in its next state, then it either must be in that mode already, or one of the requirements' transitions is occurring: from Grant1 on @T(In(SubSys1=S1Done)) or from Grant2

[2] @T(a)(@F(a)) is a shorthand notation to express events. For a complete treatment of events in CTL, please see [7].

on @T(In(SubSys2=S2Done)) (see Figure 2). In CTL, this property is expressed as

$$AG(Manager = Stable \vee$$
$$(EX(Manager = Stable) \rightarrow (Manager = Stable \vee$$
$$(Manager = Grant1 \ \wedge \ @T(In(SubSys1 = S1Done))) \vee$$
$$(Manager = Grant2 \ \wedge \ @T(In(SubSys2 = S2Done))))))$$

This property was obtained by composing the rows of the Manager mode transition table which have Stable in their right columns. Such properties capture the second part of our notion of consistency and are called "only legal transitions" (OLT) properties. We generate one OLT property for each value of controlled variable and every mode appearing in the right columns of mode transition tables.

Both types of properties can be expressed as single-quantifier temporal logic formulae, with existential quantifiers for ALT properties and universal quantifiers for OLT properties. The last part of the notion of consistency is represented by user-specified properties, each of which can also be translated into a single-quantifier temporal logic formula (see [7]).

2.3. Detailed Design

Our PDL was motivated by two factors: we wanted designs that look close to real programs, i.e., have control flow information, yet reflect changes to requirements variables. So, designs written in our PDL contain a mixture of control-flow constructs and *annotations* - special statements reflecting communication information and changes to values of requirements variables. These changes are captured using three types of annotations: an *Initial annotation* indicates the starting state of each modeclass, unconditionally assigning values to variables; an *Update annotation* assigns values to variables, identifying points at which the program changes its state; an *Assert annotation* captures the designer's hypothesis that variables have particular values in the current system state, indicating information that the designer *believes* to be true. These constructs are the same for concurrent and sequential processing, although semantics of an Update annotation is such that only one Update can occur in the entire system at one time, to ensure the interleaving semantics. In addition, we define constructs to label communication points in the design. *Read* and *Write* annotations are used to identify points at which processes exchange information. This exchange and synchronization are done via shared buffers. Each buffer is a record consisting of variable-value pairs for all controlled variables used for communication. An annotation "Buf: Read" indicates that a process reads the values of its inputs from a buffer named Buf. The process sends a message to the buffer's Read() routine, which waits until there is some information in the buffer, removes it, and returns the values of relevant variables to the process. "Buf: Write" indicates that a process writes the values of its outputs to a buffer named Buf. The process sends a message to the buffer's Write() routine, specifying values of variables to be stored. The routine waits until the buffer is empty, stores the values of the variables and returns control to the process. Several processes can write to the same buffer or read from it, but not both.

We have chosen this semantics because it reduces the chances for starvation and deadlock, ensures that no events are missed, and that all events are "fresh" (see Section 4.1). This mechanism is similar to a hand-shaking protocol used in communication software: express the intent to write, wait for the reader's acknowledgement, send the data and wait for an acknowledgement that it has been received. Although it allows for correct interpretation of events, this semantics might not be the most appropriate for all applications. In some cases we want to prioritize messages, while in others we do not want the writer to wait. Possible semantics include (priority) queues of messages, asynchronous writes into one buffer, rendevous, and many others. Most of these alternative semantics result in weaker analysis. We might also remove restrictions on Updates to enable processes to change states simultaneously. Evaluation of design and analysis tradeoffs among different types of buffers and various semantics for process composition is part of our ongoing research.

A designer might also use other statements in addition to annotations and control-flow constructs. We do not process the former since they do (should) not reflect changes of values of requirements variables. To differentiate between statements and annotations, the later start with @@.

We use a detailed design of our specifications of the mutual exclusion protocol from Section 2 to illustrate the language and the design technique. Figure 3 shows the design of the Manager (a)) and SubSys1 (b)). This system has three buffers: A, B and C. Manager waits at a point "A: Read" until some information (the values of In(SubSys1=S1Wait) or In(SubSys2=S2Wait)) is written into buffer A at a point "A: Write" by either of the subsystems. Values of variables In(Manager=Grant1) and In(Manager=Grant2) are written into buffers B and C by the Manager, and read by the subsystems. Finally, the values of In(SubSys1=S1Done) and In(SubSys2=S2Done) are written into the buffer C and read by the Manager at a point "C: Read".

After a detailed design is complete and verified, it can be further refined to build an implementation. The implementor must choose to match the design semantics of the PDL constructs if he or she hopes to benefit from the analysis. It might also be feasible to develop techniques to automatically generate code from designs. We will explore this possibility in future research.

```
/* Manager */
{ @@ INITIAL Manager(Stable);
 while(1) {
   if (manager is free) {
     @@ Assert Manager(Stable);
     @@ A: READ;
     /* someone requests grant */
     if (this is system two) {
       @@ ASSERT In(SubSys2=S2Wait);
       @@ UPDATE Manager(Grant2);
     }
     else {
       @@ ASSERT In(SubSys1=S1Wait);
       @@ UPDATE Manager(Grant1);
     }
     /* tell processes */
     @@ B: WRITE;
   }
   else{
     /* one subsystem is done */
     @@ Assert ~Manager(Stable);
     @@ C: Read;
     @@ UPDATE Manager(Stable);
   }
 }
}
```

a)

```
/* subsystem 1 */

{ @@ INITIAL SubSys1(S1Done) &
      ~Time & ~Done;
  while (1) {
    /* wait for some time */
    @@ UPDATE Time;
    @@ UPDATE SubSys1(S1Wait)
       & ~Done;
    @@ A: WRITE;
    @@ B: READ;
    /* assume grant is given */
    @@ ASSERT In(Manager=Grant1);
    @@ UPDATE SubSys1(S1RecGrant)
       & ~Done;
    /* work for a while */
    @@ UPDATE Done;
    @@ UPDATE SubSys1(S1Done);
    @@ C: WRITE;
  }
}
```

b)

Figure 3. Detailed design of MES. Manager and SubSys1 processes.

3. Creating an Abstraction

The detailed design is used to create a Concurrent Design Flow Graph (*CDFG*) which is abstracted into a Finite-State Machine (*FSM*). Afterwards, *FSM* is used to check system properties.

3.1. Constructing a Flow Graph

A *Concurrent Design Flow Graph with Buffers*(*CDFG*) of a detailed design D is a directed graph $\langle V, E, B, V_0\rangle$, where V is the set of nodes corresponding to decisions, joins and annotations of D, E is the set of edges representing sequential or synchronization control flow, B is the set of shared buffers used in communication between processes, and V_0 is a set of entry points, one for each process. A *sequential control flow edge* represents sequential flow of control within sequential parts of each process's design. A *synchronization edge* is an edge from a node containing a Write to the corresponding buffer, and from the buffer to a node with a Read. A node can have at most one synchronization edge, although a buffer can have several incoming and outgoing edges. Figure 4 contains a portion of *CDFG* corresponding to the code fragment in Figure 3. (SubSys2 is ommitted for clarity.) Sequential and synchronization edges are identified in the figure, following the convention of [13]. For clarity, we have ommited control-flow edges between buffers and the nodes using them. In the actual *CDFG*, there is a backward sequential edge for each synchronization edge.

Our computation of system states at each node of the *CDFG* is similar to that for reaching definitions via dataflow techniques[1, 13, 22]. `gen` sets capture new values generated at each node of the *CDFG*. `kill` sets represent values that variables cannot have after each node. These two sets are propagated throughout the graph, forming `ins` and `outs` for each node. `in` (`out`) information represents possible system states on entry to (exit from) a node. Each node contains nine sets: `ins`, `outs`, `gens` and `kills` for *reaching values* (`RVs`) and *conditions* (`Conds`), and an `info` set which is discussed below. The `RV` of a variable at a node is a set of values that the variable may attain if the control reaches the node. `RVs` are computed by interpreting all design statements except Asserts. The `Cond` of a variable at a node is a set of values the variable must attain if control reaches the node. `Conds` are computed by interpreting all design statements.

`gen` sets are empty except for nodes corresponding to Initial, Update and Assert annotations, where they contain variable-value pairs for each variable. These pairs are produced from values in these annotations. Variables which are unrelated to those appearing in these statements are assigned empty-set values in the `gen` set. Two different `gen` sets are calculated, $\texttt{gen}_u$ and $\texttt{gen}_a$, to compute `RVs` and `Conds`. These sets are identical except at nodes corresponding to Assert statements where $\texttt{gen}_u$ is empty and $\texttt{gen}_a$ contains variable-value pairs derived from the annotations. We also distinguish between $\texttt{kill}_u$ and $\texttt{kill}_a$ which contain the set of all possible values of a variable except those in a respective gen set. For example, an annotation @@Update Manager(Grant1) indicates that the Manager enters a mode Grant1, and sets an output variable In(Manager=Grant1) to True. This annotation produces the following gen and kill sets:

$$
\begin{aligned}
\texttt{gen}_u = \texttt{gen}_a &= \{(Manager, \{Grant1\}),\\
&\quad (In(Manager = Grant1), \{True\})\}\\
\texttt{kill}_u = \texttt{kill}_a &= \{(Manager, \{Stable, Grant2\}),\\
&\quad (In(Manager = Grant2), \{False\})\}
\end{aligned}
$$

We associate two sets of information for each buffer, RV(info) and Cond(info). These sets represent may and must values of variables written to the buffer.

Let $Pred_{seq}(n)$ and $Pred_{sync}(n)$ be sets of all predecessors of a node or a buffer n via sequential and synchronization edges, respectively. Our SCR model assumes that each process is data-independent, and explicit synchronization is used for data exchange. Thus, our treatment of synchronization edges is similar to that of sequential edges. Let $PN(n)$ be a process number of a node n, and $Read(buf, s, PN(n))$ and $Write(buf, s, PN(n))$ be variable-value pairs of set s that a process $PN(n)$ can respectively read from or write to a buffer buf. The equations to compute state information for each process are shown in Figure 5. To compute RV(in) for a node, we union the RV(out) information of its sequential predecessors. If a node represents a Read, we also union the RV(info) of the corresponding buffer. Each variable in RV(info) and Cond(info) for each buffer is initialized to its maximum ($\top$) value, i.e., a set containing all possible values for all variables, RV(out) for every node is equal to its $\texttt{gen}_u$, Cond(out) is equal to this node's $\texttt{gen}_a$, and RV(in) and Cond(in) are empty.

Consider calculating RV(in(6)) for SubSys1. The sequential predecessor of this node is 5 (same process), and the synchronization is via buffer B's Read operation. The only writer to this buffer is node 12 of the Manager. Its predecessor is node 11, and predecessors of node 11 are 8 and 10. Node 8 contains a transition to Manager(Grant1) and thus sets In(Manager=Grant1) to True and In(Manager=Grant2) to False. Node 10 contains a transition to Manager(Grant2), setting In(Manager=Grant2) to True and In(Manager=Grant1) to False. Information from these two nodes is combined in node 11, propagated to node 12 and then written to the buffer. So,

$$
\begin{aligned}
\texttt{RV(info}(B)) = \texttt{Cond(info}(B)) &=\\
\{(In(Manager = Grant1), &\{True, False\}),\\
(In(Manager = Grant2), &\{True, False\})\}.
\end{aligned}
$$

SubSys1 executes "B: Read" and receives $\{$(In(Manager= Grant1), $\{$True,False$\})\}$. This information is unioned with RV(out(5)) (SubSys1) to obtain

$$
\begin{aligned}
\texttt{RV}(in(6)) = &\{(SubSys1, \{S1Wait\}), (Time, \{False\}),\\
&(Done, \{False\}), (In(Manager = Grant1),\\
&\{True, False\})\}.
\end{aligned}
$$

Table 3 shows information computed for each buffer. Cond and RV sets for each buffer are the same, so only RV sets are listed. The *CDFG* with all sets calculated can be found in [8].

After a fixpoint is computed, Analyzer computes an intersection of RV and Cond values for each variable at every node, storing the result in this node's info set. The tool also attempts to remove unreachable nodes: those for which Cond values of some variables are not a subset of its RV values. Once such a node is discovered, Analyzer issues a warning that the designer's hypothesis cannot be satisfied, and markes it unreachable. Let us assume that the designer made a mistake on line 9 of the Manager process, changing the Manager state to Grant1. Then information written to buffer B is $\{$(In(Manager=Grant1), $\{$True$\}$), (In(Manager= Grant2), $\{$False$\})\}$, and this is what the subsystems read in node 6. SubSys2 assumes that it has been given a grant in node 7 (Assert In(Manager=Grant2)) which contradicts this node's RV information, making this node, as well as nodes 8, 9, 10, and 11 unreachable. This means that SubSys2 never writes to buffer C, which might have an effect on the Manager process, etc. Figure 6 summarizes the algorithm used by Analyzer to compute the system information and to eliminate unreachable paths.

3.2. Constructing a Finite-State Machine

The *CDFG* contains nodes which do not reflect state changes (i.e., splits, joins, Assert annotations, etc.). During the construction of a Finite-State Machine *FSM*, we remove all unreachable nodes, buffers and those nodes which do not correspond to Initial, Update, and Read annotations. The resulting *FSM* can then be used for the verification of the system properties. Nodes containing Read statements are just placeholders indicating that Analyzer should search for Triggering conditions in other processes. When a node is removed during the *FSM* construction, Analyzer searches for all of its immediate predecessors and successors remaining in *FSM*, and connects all predecessors to all successors. *FSM* corresponding to the *CDFG* on Figure 4 is shown in Figure 7, with Read statements shaded. Nodes 2, 3, 7, 11 and 13 of SubSys1 and SubSys2, and nodes 2, 3, 4, 6, 7, 9, 11, 12, 13, 16, and 17 of Manager are removed. Node 11 was a successor of node 8; but since 11, 12 and 17 were removed, nodes 5 and 14 became the successors of 8. The removed node 12 was connected via a synchronization edge to buffer B and from there to nodes 6 of the subsystems. Thus, node 8 is now connected to the subsystems via synchronization edges. For each node left in *FSM*, we keep $\texttt{gen}_u$, $\texttt{kill}_u$ and info sets, since we need them for property verification.

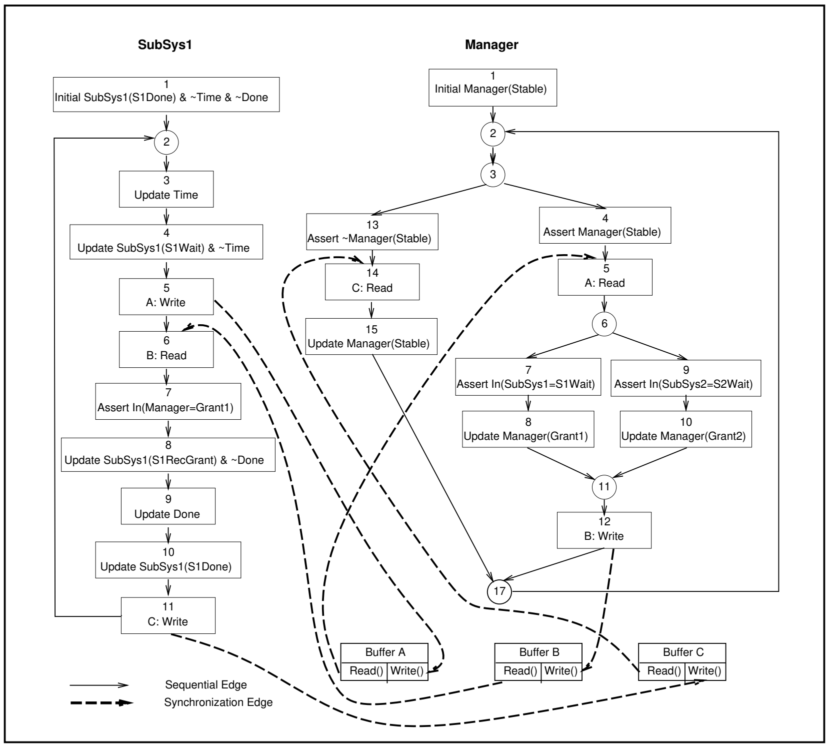

Figure 4. Fragment of CDFG of MES.

4. Verification of Properties

In this section we describe how automatically-generated and user-specified properties are verified. During the process described in Section 3, we took a detailed design and built a *FSM* from it, combining all possible values of each variable at every step of the design and removing nodes which do not correspond to state changes or which are clearly unreachable. Unfortunately, since our analysis is static, we are unable to determine which paths through the resulting *FSM* are feasible, and thus we cannot verify properties exactly, i.e., claim that a property is violated if and only if we found a violation. Thus, we will divide the properties into two categories. A property is checked *pessimistically* if all of its violations in the design are detected, but the analysis can result in some false negatives. A property is checked *optimistically* if all detected violations are present in the design, but the analysis might not be able to find all violations of this property.

OLT and most of user-specified properties, e.g., inv and strcause, involve quantification over every possible state, so a property violation might be reported on an infeasible path. Thus, we check these properties pessimistically. ALT and reach properties, on the other hand, are existentially quantified, i.e., if there is a state in the model which satisfies these properties, there might not be one in the design because such a state might be on an infeasible path. Thus, we check these properties optimistically.

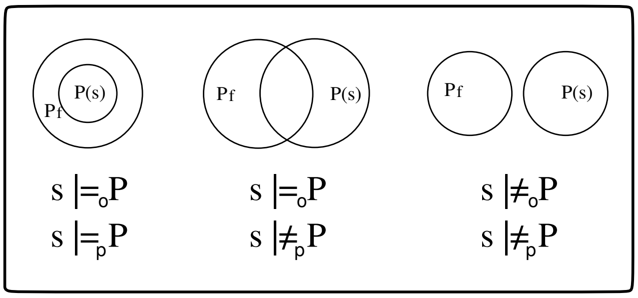

Figure 8. Interpretation of a proposition P holding in a state s.

We define two interpretations for a proposition P holding in a state s, one for pessimistic interpretation ($s \models_p P$) and another for optimistic interpretation ($\models_o P$). Variables in states have values represented by sets, so we treat atomic (or negated atomic) propositions in formulas as assertions about sets of values, e.g., A is treated as $A = \{True\}$ and $\sim M(M_1)$ is treated as $M = \{M_2, M_3\}$ (given that M is a modeclass containing modes M_1, M_2 and M_3). To check if an atomic proposition P holds in a state s, we compare P's asserted value (P_f) with its value in state s ($P(s)$ which is $\texttt{info}(s)$ or $\texttt{gen}_u(s)$) (see Figure 8). For

$$\texttt{RV}(\texttt{in}(n)) = \bigcup_{p \in Pred_{seq}(n)} \texttt{RV}(\texttt{out}(n)) \bigcup_{buf=Pred_{sync}(n)} Read(buf, \texttt{RV}(\texttt{info}(buf)), PN(n))$$

$$\texttt{Cond}(\texttt{in}(n)) = \bigcup_{p \in Pred_{seq}(n)} \texttt{Cond}(\texttt{out}(n)) \bigcup_{buf=Pred_{sync}(n)} Read(buf, \texttt{Cond}(\texttt{info}(buf)), PN(n))$$

$$\texttt{RV}(\texttt{out}(n)) = \texttt{RV}(\texttt{in}(n)) - \texttt{kill}_u(n) \cup \texttt{gen}_u(n)$$

$$\texttt{Cond}(\texttt{out}(n)) = \texttt{Cond}(\texttt{in}(n)) - \texttt{kill}_a(n) \cup \texttt{gen}_a(n)$$

$$\texttt{RV}(\texttt{info}(buf)) = \bigcup_{p \in Pred_{sync}(buf)} Write(\texttt{RV}(buf, \texttt{out}(p)), PN(p))$$

$$\texttt{Cond}(\texttt{info}(buf)) = \bigcup_{p \in Pred_{sync}(buf)} Write(\texttt{Cond}(buf, \texttt{out}(p)), PN(p))$$

Figure 5. Dataflow equations for processes with synchronization.

Name	Writers	Readers	RV
A	node 5, SubSys1 & SubSys2	node 5, Manager	{(In(SubSys1=S1Wait), {True,False}), (In(SubSys1=S1Done), {False}), (In(SubSys2=S2Wait), {True,False}), (In(SubSys2=S2Done), {False})}
B	node 12, Manager	node 6, SubSys1 & SubSys2	{(In(Manager=Grant1), {True,False}), (In(Manager=Grant2), {True,False})}
C	node 11, SubSys1 & SubSys2	node 14, Manager	{(In(SubSys1=S1Wait), {False}), (In(SubSys1=S1Done), {True,False}), (In(SubSys2=S2Wait), {False}), (In(SubSys2=S2Done), {True,False})}

Table 3. Information computed for buffers.

optimistic analysis, if there are common values between P_f and $P(s)$, i.e., $P_f \cap P(s) \neq \emptyset$ (see Figure 8a), we assume that P_f holds in s. In pessimistic analysis we do not want to overlook any errors, and thus we consider P_f to hold in a state s only if $(P(s) \neq \{\}) \wedge (P(s) \subseteq P_f)$, i.e., all values of $P(s)$ are present in P_f.

Automatically-generated and user-specified properties are verified separately. The rest of this section describes the verification steps.

4.1. Verification of Automatically-Generated Properties

To verify automatically-generated properties, we take advantage of their rather restricted forms and verify all of them in a single traversal of the *FSM*.

Event Computation. Automatically-generated properties involve state transitions which occur only in response to events. To calculate a node's event, we traverse the *FSM* arcs backwards until an Update node (n) is found on each branch. Some of Update nodes can come via synchronization edges, and we assume that these nodes can also mark potential events. For each found node, we look for its predecessors to check that the Update actually marks an event. A variable v in $\texttt{gen}_u(n)$ is a part of the Triggering condition under optimistic analysis if there is *a branch* on which n reflects a change of value of v. v is a part of the Triggering condition under pessimistic analysis if n reflects a change of value of v on *every branch*. If a modeclass variable mc has a value m in $\texttt{kill}_u(n)$, @F($mc(m)$) is part of the Triggering condition under optimistic (pessimistic) analysis if mc is in mode m in some (all) predecessors of n.

We form conjunctions of the Triggering and When conditions using Update nodes which mark an event. For example, consider calculating an event which results in the transition to Manager(Stable) at node 15 of the Manager process (Figure 7). The predecessor of this node is a specially marked Read node (node 14) indicating that the events should come from other processes. Thus, Analyzer looks for predecessors of 14, finding node 10 in SubSys1 and SubSys2. These nodes contain Update annotations, changing the mode of SubSys1 and SubSys2, respectively. The Triggering condition for a transition from node 10 (SubSys1) to 14 (Manager) is

$$@T(SubSys1 = S1Done)) \wedge @T(In(SubSys1 = S1Done)) \wedge @F(SubSys1 = S1RecGrant))$$

under pessimistic and optimistic analysis, since

Algorithm

```
initialize each buffer's RV(info) and Cond(info) to ⊤ for each variable
mark all states as reachable
for each process p
    change[p] = 1
repeat
    while (there is process p s.t. change[p] = 1)
        for each p
            if change[p]
                change[p] = 0
                compute fixpoint state information for this process, considering
                only reachable states
                for each node n
                    compute info(n) = RV(n) ⊓ Cond(n)
                    determine if this node is unreachable
                if found new unreachable states
                    notify user
                    change[p] = 1
    for each buffer b
        compute RV(info(b)) and Cond(info(b)) - union of RV(out) and
        Cond(out), respectively, of all reachable writers to b
        if RV(info(b)) or Cond(info(b)) has changed
            for all processes p reading from b
                change[p] = 1
until there are no changes in buffers
```

Figure 6. Algorithm to compute system information.

$$
\begin{aligned}
\mathtt{gen}_u(10) &= \{(SubSys1, \{S1Done\}),\\
&\quad (In(SubSys1 = S1Done), \{True\})\}\\
\mathtt{kill}_u(10) &= \{(SubSys1, \{S1Wait, S1RecGrant\}),\\
&\quad (In(SubSys1 = S1Done), \{False\}),\\
&\quad (In(SubSys1 = S1Wait), \{True\})\}\\
\mathtt{info}(9) &= \{(SubSys1 = \{S1RecGrant\}),\\
&\quad (In(SubSys1 = S1Done), \{False\}),\\
&\quad (In(SubSys1 = S1Wait), \{False\}),\\
&\quad (Done, \{True\}), (Time, \{False\}),\\
&\quad (In(Manager = Grant1), \{True\})\}
\end{aligned}
$$

(Node 9 is a predecessor of 10). SubSys1(S1Wait) is not part of `info`(9), so @F(SubSys1 (S1Wait)) is not in the Triggering condition, and neither is @F(In(SubSys1=S1Wait)). The Triggering condition for a transition from node 10 (SubSys2) to 14 (Manager) is

$$
\begin{aligned}
@T(SubSys2 = S2Done) \wedge @T(In(SubSys2 = S2Done))\\
\wedge\ @F(SubSys2 = S2RecGrant))
\end{aligned}
$$

under both types of analysis. Only a portion of these conditions is relevant for the Manager, namely, $@T(In(SubSys1 = S1Done))$ and $@T(In(SubSys2 = S2Done))$. The When conditions are the values of variables relevant to Manager in `info` sets of node 10 of each subsystem unioned with the `info` set of node 14 (the Read node). Events discovered by Analyzer for mode transitions between (Manager, {Grant1, Grant2}) and (Manager, {Stable}) are shown in Figure 9.

Our event calculations change the semantics of SCR events in several ways, although none of them invalidates the analysis. First of all, we change the time at which a When condition is supposed to hold. Recall that in SCR, $@T(a)$ WHEN $[b]$ means that in the old state the value of a is False and the value of b is True, whereas in the new state the value of a is True. We get the value for b at the same node as the Triggering condition ("new" state). However, since the When and the Triggering conditions are disjoint, b is not part of the Update statement changing the value of a, and thus b had to hold prior to that statement, i.e., in the "old" state. Secondly, SCR mode transitions occur simultaneously with the events triggering these transitions. We assume that Triggering conditions and mode transitions occur in different states. However, we require that no events occur between these states.

Verification of Properties. All automatically-generated properties are verified in a single traversal of the *FSM*. When

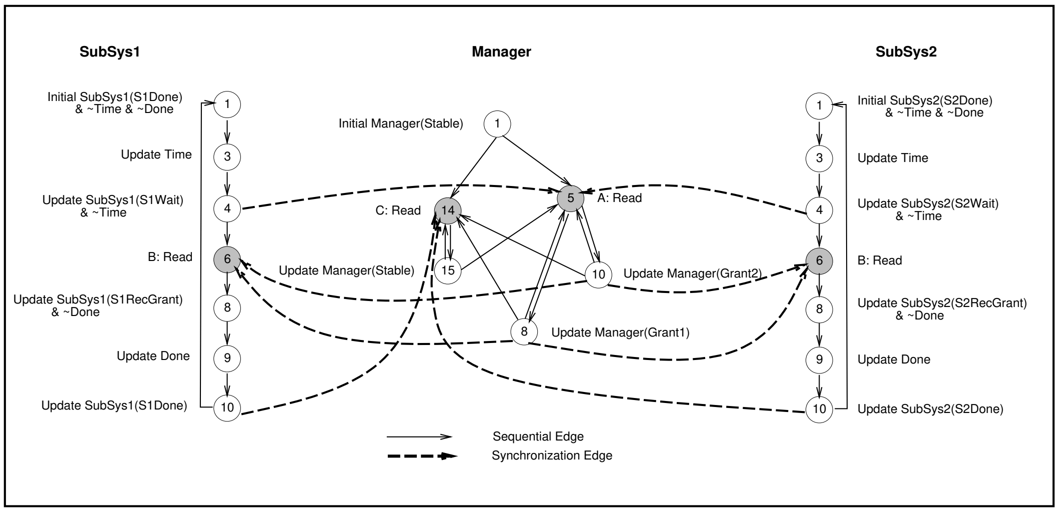

Figure 7. FSM of MES.

$$I_1:\ @T(In(SubSys1 = S1Done))\ \texttt{WHEN}\ [In(SubSys1 = S1Wait) = \{False\}\ wedge\ Manager = \{Grant1, Grant2\} \wedge In(Manager = Grant1) = \{True, False\} \wedge In(Manager = Grant2) = \{True, False\}]$$

$$I_2:\ @T(In(SubSys2 = S2Done))\ \texttt{WHEN}\ [In(SubSys2 = S2Wait) = \{False\} \wedge Manager = \{Grant1, Grant2\} \wedge In(Manager = Grant1) = \{True, False\} \wedge In(Manager = Grant2) = \{True, False\}]$$

Figure 9. Events discovered for a transition to Manager(Stable), node 15.

Analyzer finds a node containing an annotation representing a mode transition or a change of value of a controlled variable, it looks for its predecessors, computes events and checks a corresponding OLT property, reporting violations as soon as they are discovered. It also marks satisfied ALT properties, reporting those unmarked at the end of the traversal.

For example, one of OLT properties asserts that if Manager is in mode Stable in its next state, then it either must be in that mode already, or one of the requirements' transitions is occurring: from Grant1 on an event $@T(In(SubSys1 = S1Done))$ or from Grant2 on $@T(In(SubSys2 = S2Done))$. During the *FSM* traversal, Analyzer discovers a node with an @@ Update Manager(Stable) annotation (node 15, Manager) and determines that in the previous state (node 14) the value of Manager was $\{$Grant1, Grant2$\}$. Then the tool looks for predecessors of node 15 containing valid Triggering events, which are nodes 10 (SubSys1 and SubSys2), calculates events (see above) and checks that both events establish the property. So, Analyzer checks if there is a transition from Manager(Grant1) events I_1 and I_2, and from Manager(Grant2) on the same events. Since two of these transitions are not specified in the requirements, the tool reports a number of violations (see Figure 10). Of course, transitions from Manager(Grant1) on I_2 and from Manager(Grant2) on I_1 cannot occur in the actual design. So, in some cases Analyzer reports more OLT violations than present in the design, leaving it up to the user to interpret which of the messages actually correspond to errors.

One of the ALT properties automatically generated from the requirements indicates that there exists a state in which Manager is in Grant1 and an event $@T(In(SubSys1 = S1Done))$ occurs, and Manager is Stable in one of the next states. Another ALT property is that there exists a state in which Manager is in Grant2 and an event $@T(In(SubSys2 = S2Done))$ occurs, and Manager is Stable in one of the next states. These properties hold in the model because events calculated for node 15 of the Manager process satisfy them.

4.2. Verification of User-Defined Properties

To verify user-specified properties, we remove nodes corresponding to Read annotations and concurrent edges from *FSM*, breaking the global *FSM* into a set of *FSM*s, one for each process. Each user-defined property is verified in a separate traversal of one or more *FSM*s. First, Analyzer determines processes on which it is sufficient to verify a particular property (see [8]). If a property can be verified on just one process, it is considered *local*. Otherwise, it is *global*.

```
Found an event on line 13, process SubSys1
The triggering condition for this transition is
@T(In(SubSys1=S1Done)) WHEN [In(SubSys1=S1Wait)=FALSE & Manager={Grant1,Grant2} &
                            In(Manager=Grant1)=EITHER & In(Manager=Grant2)=EITHER]

Found rule:
    GRANT1   @T(IN(SUBSYS1=S1DONE))   STABLE

Error on line 22, process Manager, in mode class MANAGER:
    no transition from mode(s) {Grant2} to mode(s) {STABLE}.

Found an event on line 13, process SubSys2
The triggering condition for this transition is
@T(In(SubSys1=S1Done)) WHEN [In(SubSys1=S1Wait)=FALSE & Manager={Grant1,Grant2} &
                            In(Manager=Grant1)=EITHER & In(Manager=Grant2)=EITHER]

Found rule:
    GRANT2   @T(IN(SUBSYS2=S2DONE))   STABLE

Error on line 22, process Manager, in mode class MANAGER:
    no transition from mode(s) {Grant1} to mode(s) {STABLE}.
```

Figure 10. Violations reported by the tool for transition in Manager(Stable).

Verification of User-Defined Local Properties. Algorithms for verifying local user-defined properties are the same as for sequential designs and appear in [7]. For example,

$$reach(Manager = Grant2)$$

involves only the Manager process, and thus is local to that process (see [8]). To verify this property, Analyzer searches *FSM* of the Manager process for a state where it is in mode Grant2, finds one (say, node 10) and decides that this property holds. As noted earlier, reach properties are verified optimistically, so it is possible that Analyzer reports no violation although the property does not hold in the design. However, when we try to verify the same property on the incorrect design (see Section 3.1, where we changed the annotation at node 10 to @@Update Manager(Grant1)), Analyzer reports an error. This means that the error is present in the actual design.

Verification of User-Defined Global Properties. During verification of global properties, we need to check each property on combinations of reachable states of each participating process. Consider verifying the invariant property

$$inv(\sim(SubSys1 = S1RecGrant \wedge SubSys2 = S2RecGrant))$$

This property involves the two subsystems. To verify the property, we compose `info` sets of all reachable states of one process with those of the other. A *composition* of nodes n_1 and n_2, written as $(n_1,\ n_2)$, respectively, consists of all variable-value pairs of `info` sets of these nodes for variables local for $PN(n_1)$ and $PN(n_2)$, and unions of values of variables shared between $PN(n_1)$ and $PN(n_2)$. In our example, there are no variables shared between these processes. So, the composition just includes local variables for each process. Analyzer goes through composed states and looks for those satisfying SubSys1=S1RecGrant $\wedge$ SubSys2=S2RecGrant namely, $(8, 8)$, $(8, 9)$, $(9, 8)$ and $(9, 9)$, and reports possible the violations. Since we do not know which of the composed states can exist in the actual design, the analysis is overly pessimistic (or, for reach properties, overly optimistic). We assume, however, that if these user-specified properties hold in the requirements and the design corresponds to its requirements, there is a strong reason to believe that these properties hold in the design.

5. Conclusion and Future Work

We have defined a notion of consistency between a set of SCR-style requirements and a detailed design of a set of concurrently executing programs implementing these requirements. We have also presented a technique to ensure this notion of consistency, implemented in a tool called Analyzer. Analyzer creates concurrent design flow graphs of each program and uses data-flow analysis to compute state information at each node of these graphs. The graphs are further abstracted into finite-state machines which are used to check a set of properties automatically generated from the requirements or specified by the user. The approach is applicable to small but realistic systems.

We need to make several modifications to our approach to be able to verify a wider range of systems. For example, many concurrent applications consist of several identical processes. We need to be able to specify requirements for these systems using SCR format without enumerating all of them (e.g., in the case of many subsystems and one manager, the manager need not be specified to go to Grant1 when subsystem 1 is waiting, to Grant2 when subsystem 2 is waiting, etc.). It would be more convenient to use arrays of modes and mode classes, as in [18, 14]. We might also need to specify exactly which variables are to be read from and written to each buffer, rather than defaulting to all variables passed between processes. Also, a useful and general semantics of communication between processes needs further investigation.

We envision Analyzer being used in a software development process in which SCR specifications are first written and checked for consistency and completeness. Then, a design is developed using a PDL. A detailed design is automatically verified for consistency with the requirements, also ensuring that global requirements properties are preserved. Afterwards, a real implementation is manually written around the PDL statements or automatically generated from them. This process gives a developer some assurance that the code implements the system behavior specified in the requirements. Finally, the implementation is thoroughly tested.

Acknowledgements

We would like to thank Connie Heitmeyer and Stuart Faulk for their suggestions on specifying the program implementation process and for their help in understanding the SCR model. We are also grateful to Rich Gerber for his time and his ideas.

References

[1] A. Aho, R. Sethi, and J. Ulman. *Compilers: Principles, Techniques, and Tools, Chapter 10*. Addison Wesley, 1988.

[2] T. Alspaugh, S. Faulk, K. Britton, R. Parker, D. Parnas, and J. Shore. "Software Requirements for the A-7E Aircraft". Technical report, Naval Research Laboratory, March 1988.

[3] J. Atlee. *"Automated Analysis of Software Requirements"*. PhD thesis, University of Maryland, College Park, Maryland, December 1992.

[4] S. Caine and E. Gordon. "PDL: A Tool for Software Design". In *Proceedings of the National Computer Conference*, volume 44, pages 271–276, 1975.

[5] M. Chechik and J. Gannon. "Automatic Verification of Requirements Implementations". In *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–14, Seattle, Washington, August 1994.

[6] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Implementations and Requirements: A Case Study". In *Proceedings of 10th Annual Conference on Computer Assurance*, pages 123–131, June 1995.

[7] M. Chechik and J. Gannon. "Automatic Analysis of Consistency Between Requirements and Designs". Technical report CS-TR-3394.1, Dept. of CS, University of Maryland, College Park, October 1995.

[8] M. Chechik and J. Gannon. "Verification of Consistency between Concurrent Program Designs and Their Requirements". Technical Report CS-TR-3554, Dept. of CS, University of Maryland, College Park, January 1996. (in preparation).

[9] E. Clarke, E. Emerson, and A. Sistla. "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications". *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[10] M. B. Dwyer and L. A. Clarke. "Data Flow Analysis for Verifying Properties of Concurrent Programs". In *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 62–75, December 1994.

[11] J. Fischer and R. Gerber. "Compositional Model Checking of Ada Tasking Programs". In *Proceedings of the Ninth Annual IEEE Conference on Computer Assurance*, June 1994.

[12] R. Gerber and I. Lee. "A Layered Approach to Automating the Verification of Real-Time Systems". *IEEE Trans. on Software Eng.*, 18(9):768–784, 1992.

[13] D. Grunwald and H. Srinivasan. "Data Flow Equations for Explicitly Parallel Programs". In *Proceedings of 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 159–167, May 1993.

[14] D. Harel. "StateCharts: A Visual Formalism for Complex Systems". *Science of Computer Programming*, 8:231–274, 1987.

[15] C. Heitmeyer and B. Labaw. "Consistency Checks for SCR-Style Requirements Specifications". Technical Report NRL Report 93-9586, Naval Research Laboratory, November 1993.

[16] C. Heitmeyer, B. Labaw, and D. Kiskis. "Consistency Checking of SCR-Style Requirements Specifications". In *Proceedings of RE'95 International Symposium of Requirements Engineering*, March 1995.

[17] K. Heninger. "Specifying Software Requirements for Complex Systems: New Techniques and Their Applications". *IEEE Transactions on Software Engineering*, SE-6(1):2–12, January 1980.

[18] M. S. Jaffe, N. G. Levenson, M. P. Heimdahl, and B. E. Melhart. "Software Requirements Analysis for Real-Time Process-Control Systems". *IEEE Transactions on Software Engineering*, 17(3), March 1991.

[19] F. Jahanian and A. K.-L. Mok. "A Graph-Theoretic Approach for Timing Analysis and Its Implementation". *IEEE Transactions on Computers*, 8:961–975, August 1987.

[20] D. Lamb. *Software Engineering: Planning for Change*. Prentice-Hall, 1988.

[21] N. Levenson, M. Heimdahl, H. Hildreth, and J. Reese. "Requirements Specification for Process-Control Systems". *IEEE Transactions on Software Engineering*, 20(9):684–707, September 1994.

[22] D. Long and L. A. Clarke. "Data Flow Analysis of Concurrent Systems That Use the Rendezvous Model of Synchronization". Technical Report COINS 91-31, Univ. Mass. Amherst, July 1991.

[23] S. Masticola and B. Ryder. "A Model for Ada Programs for Static Deadlock Detection in Polynomial Time". *Proceedings of ACM Sigplan Programming Languages Design and Implementation, ACM Sigplan Notices*, 26(12):97–107, 1991.

[24] D. Parnas and J. Madey. "Functional Documentation for Computer Systems Engineering(Version 2)". Technical Report CRL Report 237, McMaster University, Department of Electrical and Computer Engineering, 1991.

[25] R. N. Taylor. "A General-Purpose Algorithm for Analyzing Concurrent Programs". *Communications of the ACM*, 26(5):362–376, May 1983.

[26] A. J. van Schouwen. "The A-7 Requirements Model: Re-examination for Real-Time Systems and an Application to Monitoring Systems". Technical Report TR-90-276, Queen's University, Kingston, Ontario, May 1990.