Static Checking of System Behaviors Using Derived Component Assumptions

PAOLA INVERARDI
Universitá dell' Aquila
ALEXANDER L. WOLF
University of Colorado at Boulder
and
DANIEL YANKELEVICH
Universidad de Buenos Aires

A critical challenge faced by the developer of a software system is to understand whether the system's components correctly integrate. While type theory has provided substantial help in detecting and preventing errors in mismatched static properties, much work remains in the area of dynamics. In particular, components make assumptions about their behavioral interaction with other components, but currently we have only limited ways in which to state those assumptions and to analyze those assumptions for correctness. We have formulated a method that begins to address this problem. The method operates at the architectural level so that behavioral integration errors, such as deadlock, can be revealed early and at a high level. For each component, a specification is given of its interaction behavior. From this specification, assumptions that the component makes about the corresponding interaction behavior of the external context are automatically derived. We have defined an algorithm that performs compatibility checks between finite representations of a component's context assumptions and the actual interaction behaviors of the components with which it is intended to interact. A configuration of a system is possible if and only if a successful way of matching actual behaviors with assumptions can be found. The state-space complexity of this algorithm is significantly less than that of comparable approaches, and in the worst case, the time complexity is comparable to the worst case of standard reachability analysis.

This article is a major revision and expansion of a paper presented at COORDINATION '97.
The work of P. Inverardi was supported in part by a CNR/CONICET bilateral project and by the LOMAPS ESPRIT project. The work of A. L. Wolf was supported in part by the National Science Foundation under grants INT-95-14202 and CCR-97-10078, and by the Air Force Materiel Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Numbers F30602-94-C-0253 and F30602-98-2-0163. The content of the information does not necessarily reflect the position or the policy of the U.S. Government and no official endorsement should be inferred. The work of D. Yankelevich was supported in part by ANPCyT, project ARTE BID PICT 1856 and by UBACyT, project ARTE TW72.
Authors' addresses: P. Inverardi, Dipartimento di Matematica, Universitá dell' Aquila, I-67010 L'Aquila, Italy; email: inverard@univaq.it; A. L. Wolf, Department of Computer Science, University of Colorado, Boulder, CO 80309-0430; email: alw@cs.colorado.edu; D. Yankelevich, Departamento de Compuatción, Universidad de Buenos Aires, 1428 Buenos Aires, Argentina; email: dany@dc.uba.ar.

Categories and Subject Descriptors: D.2 [**Software**]: Software Engineering; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*modules and interfaces, state diagrams;* D.2.4 [**Software Engineering**]: Software/Program Verification—*assertion checkers, formal methods;* D.2.11 [**Software Engineering**]: Software Architectures—*languages;* F.3 [**Theory of Computation**]: Logics and Meanings of Programs; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*assertions, mechanical verification, specification techniques*

General Terms: Design, Theory, Verification

Additional Key Words and Phrases: Assumptions, Chemical Abstract Machine model, component-based systems, static analysis

1. INTRODUCTION

A critical challenge faced by the developer of a software system is to understand whether the system's components correctly integrate. While type theory has provided substantial help in detecting and preventing errors in mismatched static properties, much work remains in the area of dynamics. In particular, components make assumptions about their behavioral interaction with other components, but currently we have only limited ways in which to state those assumptions and to analyze those assumptions for correctness.

In previous work [Inveradi and Wolf 1995; Inverardi and Yankelevich 1996], we developed a specification and analysis method for software architectures based on the CHAM (Chemical Abstract Machine) formalism [Berry and Boudol 1992]. The CHAM formalism had, until then, been used primarily to describe the semantics of various models of concurrency and the semantics of various concurrent programming languages. We showed how it also could be used to describe actual software systems.

The method has proven to be useful for uncovering a variety of errors at the architectural level. One class of such errors involves mismatches in architectural components [Garlan et al. 1995a], where assumptions made by different components are in conflict. We showed how our method could be used to uncover architectural mismatch in component behavior [Compare et al. 1999].

There is, however, a significant shortcoming in the method as it was defined. This shortcoming limits the method's usefulness when one is developing a system by assembling existing architectural components. In particular, the method depends on a monolithic specification and analysis of a whole system's component interaction behavior. A more appropriate method would permit the *individual* specification of a component's interaction behavior together with a specification of the *assumptions* that the component makes about the expected interaction behavior of other components with which it might have to interact. The method would then use the specifications to discover mismatches among the components at system integration or configuration time.

We can illustrate this point through an analogy that we call the *guest analogy*. Suppose you are invited to a party. You expect the host to receive you at the door and to invite you in. You also expect your host's partner to take you to the living room and to offer you a drink. If your host's partner does not yet know you, then you expect your host to first introduce you to the partner. If both individual behaviors (host and partner) are satisfied, but your host disappears before introducing you to the partner, then you will be in an uncomfortable situation. From your perspective, it is therefore insufficient to have only the behavior of your interaction with the host and your interaction with the partner described, but your assumptions about the global party context—that the host will introduce you to the partner—must also be described.

We have begun to formulate a new method that takes this approach. The method proceeds in four basic steps.

(1) *Component Specification.* The first step is the specification of what we call *component* CHAMs, which are similar in form to normal "system" CHAMs, but involve the use of metavariables to act as placeholders for the components with which the specified component is expected to interact. This is consistent with a component-based engineering approach in which individual components are developed independently of the system or systems within which they might eventually be used.

(2) *System Configuration.* The second step is the *configuration* of a system from individual components, which involves an instantiation of the metavariables present in component CHAMs.

(3) *Representation Derivation.* The third step automatically derives from a component CHAM two finite representations, the first being a representation of the *actual* behavior of the specified component and the second being a representation of the *assumed* behavior of the specified component's external context (i.e., an expectation of the actual behaviors of other components).

(4) *Correctness Check.* Finally, the fourth step is a static correctness check of the configuration performed using the representations of actual and assumed behaviors. The check is based on an algorithm that we have developed to incrementally assess the *compatibility* between a component's context assumptions and the actual interaction behaviors of the other components in the configuration.

A configuration of a system is possible if and only if a successful way of matching actual behaviors with assumptions can be found. In effect, we are extending the usual notion of type checking to include the checking of behavioral compatibility.

One of the goals, and major contributions, of our work is to prove deadlock freedom of a system without building a complete finite-state model of the entire system, as is done in traditional approaches based on nondeterministic finite-state machines (e.g., Wright's use of CSP [Allen and Garlan 1997]). With respect to standard reachability analysis, our ap-

proach has the advantage that it dramatically reduces the state space. In particular, we only build (finite) representations of the individual components, which for each such component consists of representations of its actual behavior and of the assumed behavior of its context. For each component, the sizes of these two representations are the same. It is well known that the state space of a system composed of N concurrent components, each of size $O(K)$, is of size $O(K^N)$. In our approach, the state space used to check for deadlock of a system composed of N concurrent components is $O(KN)$. Furthermore, in the worst case, the time complexity is comparable to the worst case of standard reachability analysis. Although not explored in this paper, we foresee several effective heuristics to reduce the average time complexity. When our algorithm returns "true," then the system is proven to be deadlock free. The price we pay for achieving a reduced complexity, however, is that when the algorithm returns "false," then we do not know whether or not the system will deadlock. Notice that this is similar to the conservative trade-off made in traditional type checking.

The reasoning for compositional variants of existing approaches based on nondeterministic finite-state machines (e.g., Darwin's use of LTS [Magee et al. 1997]) is different, since in general they are effective only under certain restrictions on the style or structure of the systems being modeled. In particular, compositional techniques are incremental, since by assuming a hierarchical decomposition of a system into subsystems, they can compute the system graph by first applying a minimization operation on each subsystem and then composing them. In general, this technique does not guarantee avoidance of the state explosion problem at intermediate stages of composition. To make this technique more effective, one must typically provide information at the subsystem level to capture the supposed interaction between the subsystem and its context. This information is encapsulated in a so-called "interface" that serves the purpose of reducing the amount of visible information that the subsystem must export. The approach is effective only when a good knowledge of the context interaction can be embodied in the interface. Our work can be seen as an attempt to address this issue.

In this paper we give an initial demonstration of the feasibility of our approach by describing its application to a system, the Compressing Proxy, first investigated by Garlan, Kindred, and Wing [Garlan et al. 1995b], and later by Compare, Inverardi, and Wolf [Compare et al. 1999]. The system contains incompatibilities between the assumptions and the interaction behaviors of two of its components. Our algorithm successfully reveals the known fact that these incompatibilities can result in a deadlock. Using a corrected version of one of the components, the system is then shown to be free of deadlock. Although currently based on the CHAM formalism, our method is likely to have wider applicability. In particular, our purpose in this paper is not to advocate a particular formalism, but rather to present the foundations of a potentially useful and practical static checking technique.

In the next section we briefly review related work. In Section 3 we provide some background on the CHAM formalism and our use of it for software architectures. In Section 4 we informally describe the Compressing Proxy problem. The specification, configuration, and derivation aspects of our method are presented in Section 5, while the checking algorithm is presented in Section 6. We conclude with some final comments on the method and our plans for future work.

2. RELATED WORK

Large software systems typically are seen as structures of individual components that behave independently and occasionally interact. Therefore, it is not unexpected that languages used to express concurrency semantics have been borrowed to describe the architectures of software systems. Besides CSP and CHAM, other models being explored include the Pi Calculus [Radestock and Eisenbach 1994] and Posets [Luckham et al. 1995]. We believe that our approach is independent of the specification language used, but one advantage of the CHAM formalism is that it does not embed within it any particular form of interaction. In most other languages, synchronous or asynchronous broadcast, or point-to-point communication are implicit and unavoidable.

From the perspective of module interconnection, informal or semiformal languages have been used to describe software architectures [Wolf et al. 1989]. In those cases, it is more difficult to prove properties of the systems. Perry [1989] presents an improved model in which the semantics of connections is taken into account to check when modules match. The semantic information in the modules, given as predicates, is used to verify some properties. One kind of predicate that can be associated with a module is a so-called "obligation," which is intended to be satisfied by the global system, not necessarily by the immediate modules with which the module interacts. These predicates can be seen as a nascent mechanism to record assumptions. Assumptions are also determined during construction of a system in the form of the semantic interconnections, which amount to the assumptions about the components used in the implementation that must be satisfied in any substitutions for those components. However, since both obligations and semantic interconnections are aimed at modules and assembly of modules, the dynamics of the system are not considered.

The use of sequences of actions associated with individual components to describe permissible interactions was introduced in Path Expressions [Campbell and Habermann 1974]. In that work, a description of potential behavior is given by a regular expression in which atomic elements represent operations on the component.

The idea of using behavioral equivalence to check the dynamics of a software system at the architectural level has been explored by Allen and Garlan [1996; 1997]. In their architectural description language Wright [Shaw and Garlan 1996], each component has one or more *ports* that represent points of interaction with other components. Rather than inter-

acting directly, however, components interact indirectly through special components called *connectors.* Connectors themselves have special ports called *roles.* Interaction occurs between two or more components by placing a connector between them and by associating each port in a component with a role in the connector. The semantics of ports and roles in Wright are given using a subset of the language CSP [Hoare 1985]. A notion of consistency is introduced via a behavioral equivalence between the CSP agents describing the semantics of corresponding ports and roles.

The main issues being studied in Wright are specifying and proving properties of individual connectors, and defining compatibility relationships through refinement. Provided that a connector is deadlock free, the compatibility relationship between roles and ports guarantees the preservation of deadlock freedom for that individual connector. Our concern is focused instead on proving global properties, initially system deadlock freedom. Given this difference in focus, we have developed an approach to proving deadlock freedom of a system without building a complete finite-state model of the entire system. To be more precise, we only build (finite) representations of the individual components, which for each such component consists of representations for its actual behavior and the assumed behavior of its context. To prove global deadlock freedom, the strategy followed in Wright is to first translate a Wright system configuration into a single CSP specification of the entire system and then to build the automata model from the CSP specification.

Although roles in Wright were introduced explicitly to support connector reuse, the idea is related to our notion of assumptions. Roles, in a sense, describe the expected behavior for a particular port. An important methodological difference from Wright is our automatic derivation of assumptions. In our approach we examine the specification of a component's actual behavior to reveal the assumptions it makes on the behaviors of other components. In the current Wright methodology, the designer is required to express those assumptions explicitly as roles in a connector. Thus, in using Wright, one anticipates a particular interaction situation by creating an appropriate connector to capture (and enforce) the assumptions of all the components involved in that specific interaction. In our approach, assumptions "come along with" each component so that each component can be used in multiple situations without requiring any additional specification effort beyond the usual configuration step inherent in any component-based architectural framework.

In the framework of specification languages for concurrent systems, Kobayashi and Sumii [Kobayashi 1998; Sumii and Kobayashi 1998] have recently proposed type systems that ensure certain kinds of deadlock freedom through static checking. Their approach is based on annotating each type of communication channel with an expression, its so-called *usage,* which specifies how the channel can and must be used. The annotations are based on a process calculus that allows the detection of certain erroneous situations arising from the incorrect use of channels. Their approach shares with us the idea of using behavioral specifications to statically check for

deadlock. It is different in the way the annotation is performed—that is, by having the programmer explicitly annotate communication channels. Thus, their approach can be seen as much closer to that of Wright and its notion of connector specifications.

The Kobayashi and Sumii approach is also strongly related to that of the earlier work of Liskov and Wing [1994], in the sense that behavioral information is being used to enhance a static type-checking mechanism. However, Liskov and Wing investigate this enhancement in the context of definitions for two particular subtype relationships, the idea being that properties demonstrated to hold for supertype objects should also hold for subtype objects. Synchronization among different components is not considered in their approach because their main concern is type and subtype compatibility in terms of input/output behavior.

Zaremski and Wing [1997] describe a technique called *specification matching* that is intended as a means to retrieve software components from a reuse library. They point out that their technique is currently limited in that it is based on simple input-output functional behavior. An enhancement that they propose to investigate would extend their formal framework to interaction protocols of architectural components, resulting in a technique for uncovering architectural mismatch.

Within the reuse community, there is an awareness of the need to enhance the behavioral description of components in order to "reason about how the behavior exhibited by a component affects the behavior of a system into which it is integrated" [Edwards and Weide 1997]. In particular, they are looking for ways to capture the assumptions made by components about the behaviors of other components. The work described here is a step in that direction.

3. BACKGROUND

The CHAM formalism was developed by Berry and Boudol in the domain of theoretical computer science for the principal purpose of defining a generalized computational framework [Berry and Boudol 1992]. It is built upon the chemical metaphor first proposed by Banâtre and Le Métayer to illustrate their Gamma (Γ) formalism for parallel programming, in which programs can be seen as multiset transformers [Banâtre and Métayer 1990; 1993]. The CHAM formalism provides a powerful set of primitives for computational modeling. Indeed, its generality, power, and utility have been clearly demonstrated by its use in formally capturing the semantics of familiar computational models such as the λ calculus and the CCS process calculus. Boudol [1994] points out that the CHAM formalism has also been demonstrated as a modeling tool for concurrent-language definition and implementation.

A CHAM is specified by defining *molecules* $m_1, m_2, \ldots$ defined as terms of a syntactic algebra that derive from a set of constants and a set of operations, and *solutions* $S_0, S_1, \ldots$ of molecules. Molecules constitute the basic elements of a CHAM, while solutions are multisets of molecules

interpreted as defining the *states* of a CHAM. A solution is denoted by a comma-separated list of molecules enclosed in braces. In a recursive fashion, solutions can be subsolutions, in which case they are considered molecules of the supersolution.

A CHAM specification contains *transformation rules* T_1, T_2, . . . that define a *transformation relation* $S_i \rightarrow S_j$ dictating the way solutions can evolve (i.e., states can change) in the CHAM. Following the chemical metaphor, the term *reaction rule* is used interchangeably with the term *transformation rule*. Transformation rules can be applied depending on the satisfaction of a condition by the current state. Conditions are expressed as *premises* in the rule, with the meaning that the rule can be applied if and only if the current state satisfies the condition expressed by the premises.

At any given point, a CHAM can apply as many rules as possible to a solution, provided that their premises do not conflict—that is, no molecule is involved in more than one rule. In this way it is possible to model parallel behaviors by performing parallel transformations. When more than one rule can apply to the same molecule or set of molecules, we have nondeterminism, in which case the CHAM makes a nondeterministic choice as to which transformation to perform. Thus, we may not be able to completely control the sequence of transformations; we can only specify when rules are enabled. Finally, if no rules can be applied to a solution, then that solution is said to be *inert*.

In our original formulation for software architectures [Inverardi and Wolf 1995] we structured CHAM specifications of a system into three parts:

(1) a description of the syntax by which components of the system (i.e., the molecules) can be represented;
(2) a solution representing the initial state of the system; and
(3) a set of reaction rules describing how the components interact to achieve the dynamic behavior of the system.

Here, we add a fourth part:

(4) a set of solutions representing the intended final states of the system.

The syntactic description of the components is given by an algebra of molecules or, in other words, a syntax by which molecules can be built. Following Perry and Wolf [1992], we distinguish three classes of components: data elements, processing elements, and connecting elements. The processing elements are those components that perform the transformations on the data elements, while the data elements are those that contain the information that is used and transformed. The connecting elements are the "glue" that holds the different pieces of the architecture together. For example, the elements involved in effecting communication among components are considered connecting elements. This classification is reflected in the syntax, as appropriate.

We model components as elements of a syntactic category, thus completely abstracting away from their internal behavior. In other words, a component is represented by a name; the only structure that we add refers to the state of the component with respect to its interaction with other components in the system. Thus, a complex molecule can represent a specific state of a component in terms of its interaction with the external context. This reflects a precise choice in the level of abstraction we have chosen to model software architectures.

The initial solution is a subset of all possible molecules that can be constructed using the syntax. It corresponds to the initial, static configuration of the system. We require the initial solution to contain one molecule for each component, thus modeling the initial state of each component. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration.

The set of final solutions represents the different possible states of the system in which the computation is considered to have completed. These solutions may or may not be inert. For example, a legitimate final solution for an iterative system would be the initial solution. If a final solution is inert, then the explicit specification of that final state serves to distinguish it from an unintended deadlock state, which is also inert. The specification of final states is common in behavioral models, such as finite-state machines.

We now present a simple example of a CHAM specification as a way to illustrate the concepts mentioned above. The example is a highly abstract client-server system. It consists of a single server and a single client. The server provides a single piece of data, and the client requests that piece of data. The specification of the Compressing Proxy system given in Section 5 is, of course, more complex. Detailed examples and explanations of our use of the CHAM formalism to model software architectures are also presented elsewhere [Compare et al. 1999; Inverardi and Wolf 1995].

The first step is to define the syntax Σ of its molecules M.

$$
\begin{array}{lcl}
M & ::= & P \mid C \mid D \mid M \diamond M \\
P & ::= & \mathbf{Server} \mid \mathbf{Client}_1 \\
C & ::= & serve(D) \mid request(D) \\
D & ::= & \mathbf{data}
\end{array}
$$

The syntax consists of the set P representing the two kinds of processing elements and of an infix operator "$\diamond$" used to express the status of a processing element. The connecting elements for the architecture are given by a second set C consisting of two operations, *request* (for input by the client) and *serve* (for output by the server). The third set D defines the data, in this case trivially the constant "**data**."

We use the two operations *serve* and *request* to represent the communication over the channel between the components. The infix operator "$\diamond$" is used to express the status of a processing element with respect to its communication behavior. In particular, the status is understood by "read-

ing" the molecule from left to right. Consider, for example, the possible server molecule $serve(\mathbf{data}) \diamond \mathbf{Server}$. This is interpreted to mean that the server is prepared to serve a client, while the molecule $\mathbf{Server} \diamond serve(\mathbf{data})$ is interpreted to mean that the server is in an idle or "wait" state with respect to its communication behavior.

The next step in specifying the client-server system is to define an initial solution S_0.

$$S_0 = serve(\mathbf{data}) \diamond \mathbf{Server},\ request(\mathbf{data}) \diamond \mathbf{Client}_1$$

This solution establishes that, in the initial configuration of the system, both the server and the client are ready to communicate.

The transformation rules define how the system can evolve from the initial solution. Consider the following simple rule, which describes the communication between the components.

$$T_1 \equiv serve(d) \diamond p_1,\ request(d) \diamond p_2 \ \rightarrow\ p_1 \diamond serve(d),\ p_2 \diamond request(d)$$

Notice the generic form of the rule, which makes use of the variables p_1 and p_2, which range over P, and the variable d, which ranges over D. With this rule we can model a deterministic behavior that evolves the system into the following state:

$$S_1 = \mathbf{Server} \diamond serve(\mathbf{data}),\ \mathbf{Client}_1 \diamond request(\mathbf{data})$$

In this state rule T_1 cannot be applied, so the system is inert. If the system is to exhibit further behavior, then we need another rule, one that describes an iterative behavior of the components:

$$T_2 \equiv p \diamond c \ \rightarrow\ c \diamond p$$

Again, notice the generic form of the rule; using variable p ranging over P and variable c ranging over C, the rule applies to both **Server** and $\mathbf{Client}_1$. Together the two rules now describe a deterministic, but infinite, behavior of the system in which every request is served. That is, the only possible behavior of the system is described by the following (infinite) sequence of rule applications

$$S_0 \overset{T_1}{\rightarrow} S_1 \overset{T_2}{\rightarrow} S_2 \overset{T_2}{\rightarrow} S_3 \overset{T_1}{\rightarrow} \ldots$$

where $S_0 = S_{3n}$, for $n = 0, 1, \ldots$.

With only a slight change to the specification we can obtain a more interesting system behavior. Consider what happens if we add a second client $\mathbf{Client}_2$ to Σ and use the following as the initial solution:

$$S_0' = serve(\mathbf{data}) \diamond \mathbf{Server},\ request(\mathbf{data}) \diamond \mathbf{Client}_1,$$
$$request(\mathbf{data}) \diamond \mathbf{Client}_2$$

We have now introduced an element of nondeterminism, as well as enriched the set of different states that can occur, because in certain states both clients can request the service and only one can be served at a time.

If we add the following rule

$$T_3 \equiv p \diamond c \ \rightarrow \ p$$

where p ranges over P and c ranges over C, then we have introduced a second element of nondeterminism, in this case with respect to the choice of rule to apply in a given state, since both T_2 and T_3 have the same premise. We have also reintroduced the possibility of a terminating, or finite, sequence of rule applications because, through T_3, molecules can enter a state in which no rule can be applied. For example, if **Server** is transformed by this rule, then no further requests can be served. Whether this is a desirable situation or not depends on whether or not the designer has identified this state as an intended final state. A reasonable choice for an intended final state might be the following:

$$S_f = serve(\mathbf{data}) \diamond \mathbf{Server}, \mathbf{Client}_1, \mathbf{Client}_2$$

This solution represents the fact that the clients no longer require service, although the server is still willing to offer the service. This solution is not inert, however, since rule T_3 can still be applied to **Server.** An inert solution that is not an intended final state can be characterized as a *deadlock* state. Note that in general a designer can indicate more than one state as an intended final state.

The preceding discussion of basic concepts is given in terms of a monolithic specification of a whole system. One of the contributions of the work described in this paper is to develop a method for breaking apart the specifications along the lines of components. This is discussed in Section 5, where we introduce the concept of the *component* CHAM. First, however, we introduce our main example, the Compressing Proxy.

4. THE COMPRESSING PROXY PROBLEM

In this section we present the design of the Compressing Proxy system. Our description is derived from that given by Garlan, Kindred, and Wing [Garlan et al. 1995b].

To improve the performance of UNIX-based World Wide Web browsers over slow networks, one could create an HTTP (Hyper Text Transfer Protocol) server that compresses and uncompresses data that it sends across the network. This is the purpose of the Compressing Proxy, which weds the **gzip** compression/decompression program to the standard HTTP server available from CERN.

A CERN HTTP server consists of *filters* strung together in series. The filters communicate using a function-call-based stream interface. Functions are provided in the interface to allow an upstream filter to "push" data into a downstream filter. Thus, a filter F is said to *read* data whenever the

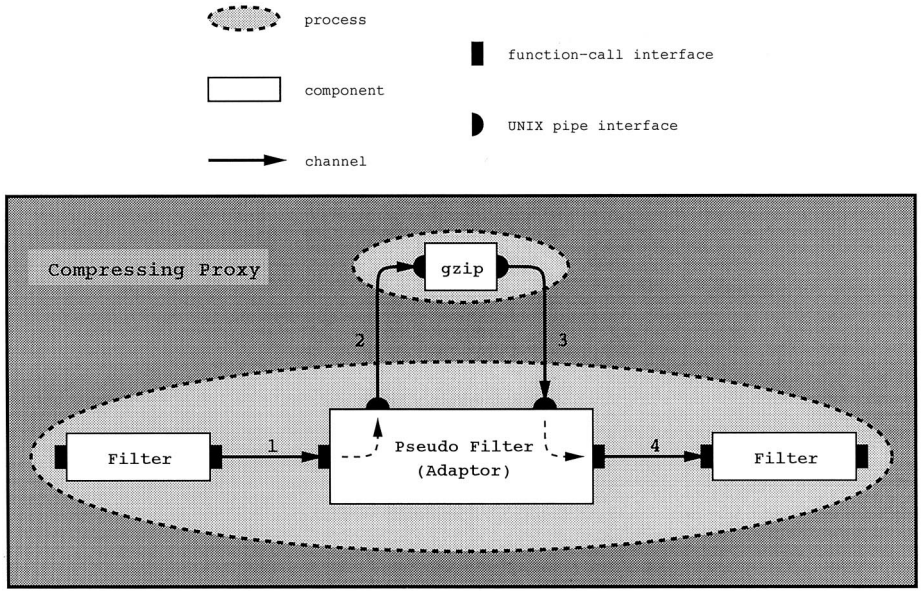

Fig. 1. The Compressing Proxy.

previous filter in the series invokes the proper interface function in F. The interface also provides a function to close the stream. Because the interface between filters is function-call based, all the filters must reside in a single UNIX process.

The **gzip** program is also a filter, but at the level of a UNIX process. Therefore, it uses the standard UNIX input/output interface, and communication with **gzip** occurs through UNIX pipes. An important difference between UNIX filters, such as **gzip,** and the CERN HTTP filters is that the UNIX filters explicitly choose when to read, whereas the CERN HTTP filters are forced to read when data are pushed at them.

To assemble the Compressing Proxy from the existing CERN HTTP server and **gzip** without modification, we must insert **gzip** into the HTTP filter stream at the appropriate point. But since **gzip** does not have the proper interface, we must create an adaptor, as shown in Figure 1. This adaptor acts as a pseudo-CERN HTTP filter, communicating normally with the upstream and downstream filters through a function-call interface, and with **gzip** using pipes connected to a separate **gzip** process that it creates.

Without a proper understanding of the assumptions made by each component, a mismatch in the interaction behavior of the components can occur when they become integrated into a single system. Consider the following straightforward method of structuring the adaptor. The adaptor simply passes data onto **gzip** whenever it receives data from the upstream filter. Once the stream is closed by the upstream filter (i.e., there are no more data to be compressed), the adaptor reads the compressed data from **gzip** and pushes the data toward the downstream filter.

From the perspective of the adaptor, this individual behavior makes sense. But it is making assumptions about its interactions with **gzip** that are incompatible with the actual behavior of **gzip**. In particular, **gzip** uses a one-pass compression algorithm and may attempt to write a portion of the compressed data (perhaps because an internal buffer is full) before the adaptor is ready, thus blocking. With **gzip** blocked, the adaptor also becomes blocked when it attempts to pass on more of the data to **gzip**, leaving the system in deadlock.

Obviously, the way to avoid deadlock in this situation is to have the adaptor handle the data incrementally and use nonblocking reads and writes. This would allow the adaptor to read some data from **gzip** when its attempt to write data to **gzip** is blocked.

The Compressing Proxy is a simple example with a well-understood solution. Nevertheless, one can see that it is representative of the all-too-common problem of architectural mismatch in software development.

5. SPECIFYING COMPONENT BEHAVIOR AND ASSUMPTIONS

In this section we show how to specify the behavior of a component at the architectural level and, from this, how it is then possible to derive a representation of its actual behavior as well as the assumptions that it makes on the external context. In essence, each component is modeled using a separate CHAM, which we refer to as a *component CHAM*. Conceptually, a complete system is specified by combining the separate CHAMs into a single, integrated *system CHAM*.

5.1 Component CHAMs

To specify a component CHAM, we give a syntax for the molecules representing the component, rules describing the behavior of the component, an initial molecule representing the initial state of the component, and a set of final molecules representing the possible final states of the component. For the Compressing Proxy we must specify four component CHAMs (Table I).

It is important to note that the justification for choosing these particular specifications of the Compressing Proxy component behaviors is not germane to the topic of this paper. In fact, a detailed understanding of the specifications is unnecessary to follow the discussion below. Therefore, we only give a partial, high-level explanation of the specifications.

Consider the upstream CERN filter $\mathbf{CF}_u$. The syntax for molecules M representing this component consists of four sets. P represents the name of the component's processing element $\mathbf{CF}_u$. It also represents a metavariable Φ_1 to refer to the unknown syntactic structure of other components with which $\mathbf{CF}_u$ is expected to interact. As discussed below in Section 5.2, metavariables are instantiated (and thereby eliminated) as a side effect of configuring component CHAMs into a system CHAM. Note that for presentation purposes we use Greek letters for metavariables (i.e., Φ and ρ) to clearly distinguish them from ordinary variables. As a further presentation aid, we use unique metavariables (e.g., ρ_1, ρ_2, ...) across the component CHAMs. This is not

Table I. Component CHAMs for the Compressing Proxy Example

Upstream CERN Filter ($\mathbf{CF}_u$)

Syntax	$M ::= P \mid C \mid M \diamond M$ $P ::= \mathbf{CF}_u \mid \Phi_1$ $C ::= i(N) \mid o(N)$ $N ::= \rho_1$
Transformation Rules	$T_1 \equiv i(\rho_1) \diamond m,\ o(\rho_1) \diamond \mathbf{CF}_u \longrightarrow m \diamond i(\rho_1),\ \mathbf{CF}_u \diamond o(\rho_1)$ $T_2 \equiv \mathbf{CF}_u \diamond o(\rho_1) \longrightarrow o(\rho_1) \diamond \mathbf{CF}_u$
Initial Molecule	$\mathbf{CF}_u \diamond o(\rho_1)$
Final Molecule	$\mathbf{CF}_u \diamond o(\rho_1)$

Downstream CERN Filter ($\mathbf{CF}_d$)

Syntax	$M ::= P \mid C \mid M \diamond M$ $P ::= \mathbf{CF}_d \mid \Phi_2$ $C ::= i(N) \mid o(N)$ $N ::= \rho_2$
Transformation Rules	$T_1 \equiv i(\rho_2) \diamond \mathbf{CF}_d,\ o(\rho_2) \diamond m \longrightarrow \mathbf{CF}_d \diamond i(\rho_2),\ m \diamond o(\rho_2)$ $T_2 \equiv \mathbf{CF}_d \diamond i(\rho_2) \longrightarrow i(\rho_2) \diamond \mathbf{CF}_d$
Initial Molecule	$\mathbf{CF}_d \diamond i(\rho_2)$
Final Molecule	$\mathbf{CF}_d \diamond i(\rho_2)$

Adaptor (AD)

Syntax	$M ::= P \mid C \mid E \mid M \diamond M$ $P ::= \mathbf{AD} \mid \Phi_3$ $C ::= i(N) \mid o(N)$ $N ::= \rho_3 \mid \rho_4 \mid \rho_5 \mid \rho_6$ $E ::= \mathbf{end}_i \mid \mathbf{end}_o$
Transformation Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2 \longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n),\ \mathbf{end}_i \diamond m_2 \diamond i(n) \longrightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o,\ m_2 \diamond i(n) \diamond \mathbf{end}_i$ $T_4 \equiv \mathbf{AD} \diamond i(\rho_3) \diamond m \longrightarrow i(\rho_5) \diamond \mathbf{end}_i \diamond o(\rho_6) \diamond \mathbf{AD}$ $T_5 \equiv \mathbf{AD} \diamond i(\rho_5) \diamond m \longrightarrow i(\rho_3) \diamond o(\rho_4) \diamond \mathbf{end}_o \diamond \mathbf{AD}$
Initial Molecule	$i(\rho_3) \diamond o(\rho_4) \diamond \mathbf{end}_o \diamond \mathbf{AD}$
Final Molecule	$\mathbf{AD} \diamond i(\rho_5) \diamond m$

GZIP (GZ)

Syntax	$M ::= P \mid C \mid E \mid M \diamond M$ $P ::= \mathbf{GZ} \mid \Phi_4$ $C ::= i(N) \mid o(N)$ $N ::= \rho_7 \mid \rho_8$ $E ::= \mathbf{end}_i \mid \mathbf{end}_o$
Transformation Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2 \longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n),\ \mathbf{end}_i \diamond m_2 \diamond i(n) \longrightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o,\ m_2 \diamond i(n) \diamond \mathbf{end}_i$ $T_4 \equiv \mathbf{end}_i \diamond m_1 \diamond \mathbf{GZ} \diamond m_2 \longrightarrow m_1 \diamond \mathbf{GZ} \diamond m_2 \diamond \mathbf{end}_i$ $T_5 \equiv \mathbf{end}_o \diamond \mathbf{GZ} \diamond m \longrightarrow \mathbf{GZ} \diamond m \diamond \mathbf{end}_o$ $T_6 \equiv \mathbf{GZ} \diamond m \longrightarrow m \diamond \mathbf{GZ}$
Initial Molecule	$i(\rho_7) \diamond \mathbf{end}_i \diamond o(\rho_8) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$
Final Molecule	$\mathbf{GZ} \diamond m$

required by our method, since each component CHAM is understood to exist on its own, but simplifies the presentation of the configuration step.

The set C represents the connecting elements. The connecting elements for this component are two operations, i for input and o for output, that act on the elements of a third set N. In general, elements of N are used to refer to the channels through which a component communicates with other components. Therefore, elements of N also act as metavariables that are instantiated in a configured system CHAM. In the case of $\mathbf{CF}_u$ we only need to consider one channel, namely the output channel for this upstream filter. Notice that for $\mathbf{CF}_d$, the downstream filter, we also only consider one channel, in this case the one representing input to the filter.

The final syntactic element of $\mathbf{CF}_u$ is the infix operator "$\diamond$", which is used to express the status of the component with respect to its communication behavior. The status is understood by "reading" a molecule from left to right. The leftmost position (i.e., the left operand of the leftmost "$\diamond$" operator) in the molecule indicates the next action that the molecule is prepared to take. If this position is occupied by a communication operation, then the kind of communication represented by that operation can take place. Correspondingly, if this position is occupied by an element of P (i.e., the name of the processing element), then the molecule is interpreted to be in a "wait" state, unprepared to communicate with other components until, and unless, there is a transformation rule that reactivates the component (e.g., T_2 in the case of $\mathbf{CF}_u$).

The interaction behavior of the upstream filter component is captured using two transformation rules, where m ranges over M. Looking at the second rule first, we see that T_2 models the iterative communication behavior of $\mathbf{CF}_u$. T_1 is an instantiation of a general interelement communication rule that describes pairwise input/output communication between processing elements. A different instantiation of this same rule is found in the component CHAM for $\mathbf{CF}_d$. A generic version of the rule is found in the component CHAMs for **AD** and **GZ,** where simple variable n ranges over N. Because M involves metavariables, the molecule m appearing in rule T_1 will range over a set that is not precisely defined until after an instantiation of the metavariables through a suitable configuration substitution. In other words, once $\mathbf{CF}_u$ has been configured to interact with its environment, m will range over a set that depends on the syntax of components with which it interacts. In the Compressing Proxy example, $\mathbf{CF}_u$ interacts with only one other component, the adaptor, and therefore m will range over a set of molecules built by using **AD** as the other processing element in the component CHAM syntax.

The initial molecule for $\mathbf{CF}_u$ is quite simple. It indicates that the component starts out in a state in which it is waiting to output data. The second transformation rule would have to be applied to this solution before it could actually carry out a communication. The set of final molecules for $\mathbf{CF}_u$ is also simple, consisting of just one molecule equivalent to the initial molecule. This reflects the iterative nature of $\mathbf{CF}_u$.

The CHAMs for the other three components follow a similar structure and share similar rules. The critical issue for this example is the interaction behaviors of **gzip** and the adaptor, so we explain them a bit further.

In the specifications of **gzip** and the adaptor, the syntaxes include a set E. The elements of E are used by **AD** and **GZ** as markers to indicate that they are in a position to end their data transfer, if appropriate; $\mathbf{end}_i$ denotes "ending input," while $\mathbf{end}_o$ denotes "ending output." Both component CHAMs share transformation rule T_2, which governs the iteration of the input and output behaviors involving markers $\mathbf{end}_i$ and $\mathbf{end}_o$. Both also share rule T_3, which represents how the component synchronizes with its environment to end iterative input or output communication. In rule T_3, the variable n ranges over N.

GZ has associated with it two additional rules for ending communication, T_4 and T_5. These rules capture the fact that **gzip** can independently terminate its input and output, respectively, without synchronizing with its environment, as it would through T_3. Intuitively, the first situation can arise when an internal buffer is full, while the second can arise when an internal buffer becomes nonfull. Note that this is exactly the root of the problem between **gzip** and its adaptor; the adaptor assumes that termination of input and output with **gzip** is always synchronized.

Rule T_6 of the **gzip** component CHAM describes a simple iterative behavior. The iterative behavior of the adaptor, on the other hand, is more complex, actually changing structure with rule T_5 of **AD.** In particular, it is characterized by a phased behavior in which the component switches from a mode of accepting raw data and then passing the data along (presumably to **gzip,** but in fact to any other component for which it is acting as an adaptor), to a mode of receiving data (again, presumably from **gzip** but also from any adapted component) and then passing the data on down the stream.

5.2 Configuring a System CHAM

As mentioned above, when component CHAMs are integrated to form a system CHAM, a certain amount of configuration must occur. For instance, in the Compressing Proxy example, the metavariables for communication channels used in the component CHAMs are instantiated according to the channel numbers in the diagram of Figure 1, resulting in actual connections being established between the components. Thus, the substitution would cause the symbolic channel ρ_1 of the upstream filter and the symbolic channel ρ_3 of the adaptor to be identified, and correspond to the channel labeled "1" in Figure 1. The substitution for a metavariable Φ appearing in a component CHAM indicates the other components with which the modeled component interacts. For instance, Φ_1 of $\mathbf{CF}_u$ would be replaced by **AD.** The complete substitution for metavariables in the configured Compressing Proxy system is given by the following set of substitution pairs:

$$\{ \quad (\Phi_1 = \mathbf{AD}),\ (\Phi_2 = \mathbf{AD}),\ (\Phi_3 = \mathbf{CF}_d \mid \mathbf{CF}_u \mid \mathbf{GZ}),\ (\Phi_4 = \mathbf{AD}),$$

$$(\rho_1 = 1),\ (\rho_2 = 4),\ (\rho_3 = 1),\ (\rho_4 = 2),\ (\rho_5 = 3),\ (\rho_6 = 4),\ (\rho_7 = 2),\ (\rho_8 = 3) \quad \}$$

A CHAM specification that does not contain metavariables is said to be *ground*.

The syntax for a system CHAM, Σ_S, is given by

$$\sum_S = \sum_{C_1} \cup \sum_{C_2} \cup \ldots \cup \sum_{C_n}$$

where $\Sigma_{C_1} \ldots \Sigma_{C_n}$ are the syntaxes of the component CHAMs after a suitable configuration substitution is given for the metavariables. For instance, in the system CHAM for the Compressing Proxy example, the resulting set P is as follows:

$$P ::= \mathbf{CF}_u \mid \mathbf{CF}_d \mid \mathbf{AD} \mid \mathbf{GZ}$$

We refer to any molecule that contains as a syntactic subterm an element of the set P as a *component molecule*.

Once a suitable configuration substitution has been applied, the transformation rules of the system CHAM are obtained in the following way. If the right-hand side of a rule t in a component CHAM C only involves C component molecules, then t can be simply added to the system CHAM. Otherwise, let us assume that t involves i other components. Then t can be added to the system CHAM only if there exists in the other i component CHAMs a rule that subsumes or is subsumed by t. In this case, the molecule variables range only over the i component molecules. In our example, each component CHAM contains a rule, labeled T_1, that either subsumes or is subsumed by the rule T_1 of all other component CHAMs. This means that all four components agree on the interaction protocol, although with different degrees of specialization. In fact, a rule that involves other component molecules in its right-hand side dictates, through that rule, a state change in another component. This can only be accepted at a global system level if the other components exhibit the same intended behavior. For example, consider the rule T_2 shared between **AD** and **GZ.** The rule is added to the system CHAM with the constraint that the variable m can only range over {**AD,GZ**}.

A certain amount of simplification could be performed on the resulting set of rules. For example, the specialized rules labeled T_1 in the component CHAMs for $\mathbf{CF}_u$ and $\mathbf{CF}_d$ are subsumed by the rule T_1 of the other two component CHAMs. Such simplifications are not, however, formally required.

The initial solution of the system CHAM is simply formed as a union of the initial molecules of the component CHAMs once a suitable configuration substitution has been applied. Creating the set of final solutions is a bit more complicated. In particular, it is derived from the cross product of the set of final molecules of each component CHAM, and in general contains a subset of the cross product. The Compressing Proxy example is a degenerate case, since the component CHAMs each have only a single final molecule. Clearly, a solution in which all the molecules represent final

states of their corresponding components must be a final solution for the system as a whole.

Of course, the configuration activity described here is not guaranteed to result in a "correct" system, which is the purpose of the checking mechanism that we develop in Section 6. The mechanism works by comparing the assumptions made in a component CHAM to the actual specified behavior of other component CHAMs with which it has been configured to interact.

5.3 Deriving Actual and Assumed Behaviors

In order to check for compatibility between components, we need representations of the actual behavior, AC, of a component, and assumed behavior, AS, of the external context. For each component, we derive AC and AS from its component CHAM once a suitable configuration substitution has been applied. Following a common hypothesis in the automated checking of properties of complex systems [Inverardi and Priami 1996], our approach assumes that these representations of dynamic behavior can be finite. The model for both representations is a finite, directed, rooted graph, where both nodes and arcs are labeled. Formally,

$$G = (N, A, so{:}A \rightarrow N, ta{:}A \rightarrow N, m{:}N \rightarrow M \cup \mathcal{N}, l{:}A \rightarrow \Lambda)$$

where N is the set of nodes; A is the set of arcs; $\mathcal{N}$ is the set of natural numbers; M is the set of node labels taken from the CHAM molecule set; and Λ is the set of arc labels taken from a set that is obtained from the syntax of the components, plus two special labels τ and α. In the Compressing Proxy example, labels are in the set $\Lambda = \{\tau, \alpha\} \cup C \cup E$. The label τ can appear only in AC graphs, while the label α can appear only in AS graphs. In addition to these sets, so is the source node function; ta is the target node function; m is the node-labeling function; and l is the arc-labeling function. Finally, the graphs are enriched with a relation on arcs called and, where $and \subseteq \mathcal{P}(A)$, the powerset of A.

AC graphs model behaviors in the following intuitive manner. Nodes represent states of the component and, therefore, are molecules. The root node is the initial state of the component. Note that in the current formulation we do not allow dynamic creation of components. Each arc represents a possible transition into a new state by using a transformation rule of the component CHAM. The label on the arc is the part of the component molecule that is required in the rule in order to match. If no other molecule should occur in the transformation, then the label of the arc is τ—that is, the transition can occur without interaction with the external context. An example of such a transformation is rule T_4 of **GZ.**

Definition 1 (*AC Graph for a Component CHAM*). AC graphs are defined constructively as follows:

—The root node is associated with the initial molecule of the component CHAM.

—Let ν be a node, and let m_ν be the molecule associated with the node ν. Then ν has a child node ν_i if and only if there exists a rule r whose application to a solution s requires m_ν to be in s. The labels and the *and* relation are constructed as follows.
  —The molecule associated with ν_i is the molecule obtained by modifying m_ν with r.
  —The arc from ν to ν_i is labeled τ if r can be applied to a solution that contains only m_ν.
  —The arc is labeled λ if λ is the part of the molecule m_ν required to match the rule r.
  —If the application of r results in more than one component molecule, then all the arcs connecting ν to a node labeled with a component molecule are identified as *and* arcs. △

Informally, *and* arcs identify alternative subgraphs for the same component. As discussed below, this corresponds to a concurrent (i.e., multithreaded) behavior of a component. With respect to proving the absence of deadlock, it is sufficient to show that there is at least one "active" alternative subgraph in every derivation.

AS graphs are intuitively the counterpart of AC graphs. They model the assumed behavior of the external context. For each AC graph, therefore, there is a corresponding AS graph that models the behavior of the context required to perform all the derivations modeled by the AC graph. Since in general the context can be provided by several components, an AS graph refers to the behavior of more than one component. It is structured as a graph because, at each step of the actual behavior, a molecule should be present in the context such that the expected transformation in the AC graph can take place. Informally, if AC nodes represent states of a component, AS nodes represent states of the other components that permit a reaction to occur in a solution. Thus, the number of nodes in an AS graph must be the same as the number of nodes in an AC graph. Moreover, there must be a correspondence between a node in an AC graph and a node in an AS graph, since they together describe a subsolution reaction.

Given an AC graph for a component CHAM we can define the corresponding AS graph.

Definition 2 (*AS Graph for a Component CHAM*). Let G_{ac} be an AC graph for some component CHAM, then the corresponding AS graph, G_{as}, is constructed as follows:

—G_{as} has as many nodes as G_{ac}.
—The root node of G_{as} is associated with the root node of G_{ac}.
—Let μ be a node in G_{as}, and let ν be the associated node in G_{ac}. Then if ν has an outgoing arc to a node ν_1 labeled λ ($\lambda \neq \tau$) due to the application of a rule r, then μ has an outgoing arc to the node corresponding to ν_1 labeled with the conjunction of the labels required by r to be applied. Each such label corresponds to the part of a molecule required in the

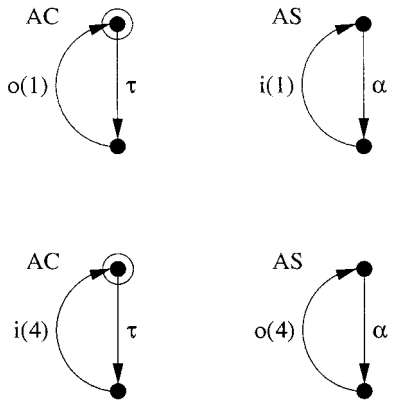

Fig. 2. AC and AS graphs for the upstream (top) and downstream (bottom) filters.

context to perform the reaction by r. If the outgoing arc in G_{ac} is labeled τ, then the outgoing arc from μ is labeled α.

—if ν has *and* arcs, then μ also has corresponding *and* arcs. △

The intuitive meaning of the α label in AS graphs is that of abstracting away from requirements on actual behaviors. That is, an α transition means a *do not care* requirement that can be matched by any sequence of transformations in the actual behavior graph AC. Actually, by construction, one of the purposes of α arcs is to model τ cycles—that is, the fact that a certain molecule can be "spontaneously" offered infinitely many times in the context. The other use of α arcs is to label *and* arcs when the transformation in the actual behavior graph has not required any context.

AC and AS graphs for the component CHAMs of the Compressing Proxy example appear in Figures 2, 3, and 4.

6. CHECKING ASSUMPTIONS

Our primary goal is to provide a way for an architect to check that a given configuration of components results in a correct system. In essence this means comparing the assumptions on the external context made by one component to the actual behavior exhibited by the components with which it interacts. To date we have concentrated on deadlock freedom as the correctness criterion and have developed an algorithm that performs the check.

The checking algorithm makes use of an equivalence relation between AC graphs and AS graphs. Informally, the goal of the check is to find a way to *match* components. This means that all the component's assumptions have to be fulfilled by some other component's actual behavior. In general, of course, multiple actual behaviors can contribute to fulfilling the assumptions of a single component. In our example, this is true for the adaptor component.

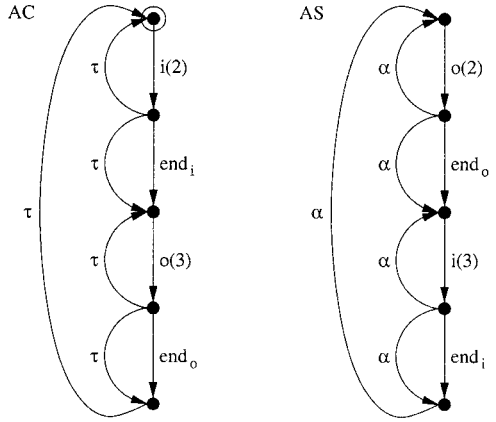

Fig. 3. AC and AS graphs for **gzip.**

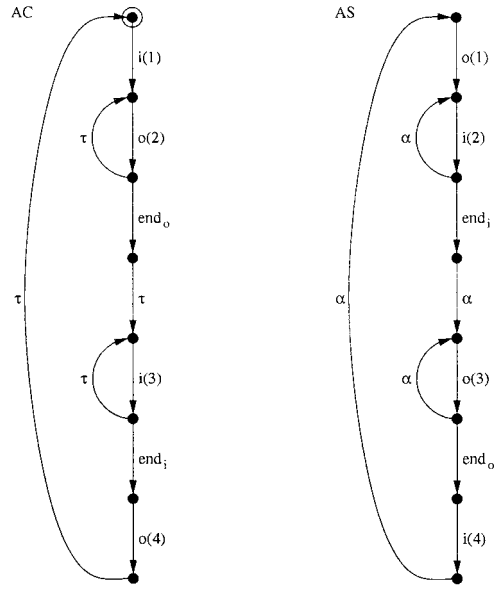

Fig. 4. AC and AS graphs for the adaptor.

If the check *succeeds*, then the system is deadlock free. If the check *fails*, then it means that there is no way to satisfy the assumptions of a component—that is, some component will block along some derivation in any possible match of components. Of course, this is not enough to conclude that the whole system blocks, but in a conservative framework it indicates a potentially erroneous situation.

6.1 Equivalence

The checking algorithm is built upon a notion of equivalence that allows us to compare AC graphs with AS graphs. It works by attempting to put

corresponding nodes and arcs into relation. Since an AS graph can be fulfilled by more than one AC graph, we try to put one AS graph in relation with more than one AC graph. The idea is that all the nodes and arcs of the AS graph must be "covered" by the relation, and, moreover, it should not be possible for an actual behavior to avoid providing the required interaction. Therefore, our equivalence is inspired by the familiar Milner bisimulation [Milner 1989].

Why do we need a form of bisimulation? An AS graph should be completely satisfied by one or more AC graphs, which means that everything the AS graph requires must be provided by the context. So, there should exist AC graphs that behave according to the needs of the AS graph—that is, they should simulate the assumed behavior. If an AC graph exists that simulates a portion of the AS graph behavior, then it should also be the case that the AC graph cannot do *less* than required by the AS graph. Otherwise there is the risk that the AC graph can take execution paths that are not guaranteed to provide the required context. Thus, we need bisimulation, wherein the AS graph must also simulate the AC graph. For example, consider the case in which the AC graph has two arcs leaving a given node, one labeled β and the other γ, while the AS graph only needs γ. Then if we only require that the AC graph simulates the AS graph, it is possible that the AC graph can take the β branch and never satisfy the assumption.

Now, why do we need something different from the usual bisimulation? Because we have to take into account two problems unique to our setting. The first is that, in general, more than one AC graph is needed to fulfill the assumptions recorded in an AS graph. For example, the assumptions made by the adaptor component of the Compressing Proxy must be satisfied by the actual behaviors of three components in the external context: the upstream filter, **gzip,** and the downstream filter. Thus, we have to understand when there is only a portion of the AS graph that is satisfied by, and therefore simulated by, some AC graph. The other problem we encounter is when we check that the AS graph is simulating the AC graph behavior. In this case, we have to take into account the possibility that the AC graph is an *and* graph, and, therefore, there can be a different thread providing the required context.

Notice that we need a notion of equivalence for verification purposes only. Our aim is to be able to compare two behavioral descriptions in a meaningful way with respect to the property we want to check. Therefore, the chosen equivalence does not imply an observational semantics at the level of the architectural language. Actually, one could go a bit further in this reasoning, where the choice of equivalence depends on the property of interest. Given a particular property, one can think of our method as parametric with respect to the equivalence. The way in which AS graphs and AC graphs are compared can be changed without modifying the overall technique. For deadlock freedom, bisimulation is adequate. For a different property, another equivalence could better fit the analysis.

In our formulation of the equivalence relation, we make use of a predicate R on nodes that is true when a node can be considered a *root*. A root is a node that is an origin of a computation—that is, whose reachability does not depend on the context, but is always guaranteed. The initial molecule for a component is, in general, a root. In Figures 2 and 3, the roots are indicated by circled nodes. Nodes that are reachable through τ arcs are also considered roots, since a τ arc means that there is no constraint imposed by the external context. To simplify the figures, we do not circle the nodes that are roots due to τ arcs.

We call an arc *recursive* when from the root node there exist infinite paths that contain that arc. A recursive arc in an AS graph indicates an assumption that the behavior should be offered an indeterminate number of times, while a recursive arc in an AC graph indicates that the behavior occurs an indeterminate number of times. Clearly, a recursive arc in an AS graph requires a corresponding recursive arc in an AC graph.

We introduce the following definitions.

Definition 3 (*Unavoidability*). Let G_{ac} be an actual behavior graph and $\nu_1 \ldots \nu_n$ be nodes in G_{ac}. We say that a node ν_k is unavoidable from a node ν_i, denoted as $\nu_i \Rightarrow \nu_k$, if there exists a path from ν_i to ν_k, and for all such paths $\nu_i \stackrel{\alpha_1}{\rightarrow} \nu_{i+1} \stackrel{\alpha_2}{\rightarrow} \nu_{i+2} \stackrel{\alpha_3}{\rightarrow} \ldots \nu_{k-1} \stackrel{\alpha_h}{\rightarrow} \nu_k$ each step $\nu_r \stackrel{\alpha_r}{\rightarrow} \nu_{r+1}$ has either $\alpha_r = \tau$ or $R(\nu_{r+1})$. $\triangle$

By definition, any root node $R(\nu)$ is reachable from anywhere—that is, it is unavoidable.

In the following we use the notation $\nu_k \in \simeq$ to mean that there exists a node μ_h such that $\nu_k \simeq \mu_h$. That is, the node ν_k already appears in a pair of the relation $\simeq$. Furthermore, we assume that in the graphs there are no nodes with outgoing arcs having the same label. This is a hypothesis that allows us to simplify the presentation of the equivalence and can be easily removed.

Given this background, we can now define what it means for an AS graph to be in relation with an AC graph.

Definition 4 (*Relation between AS and AC Graphs*). Let G_{ac} be an actual behavior graph, $\nu_1 \ldots \nu_n$ be nodes in G_{ac}, G_{as} be an assumption graph, $\mu_1 \ldots \mu_m$ be nodes in G_{as}, and $\gamma \in \Lambda \setminus \{\tau, \alpha\}$. Two nodes are related, $\nu_i \simeq \mu_j$

—if $\mu_j \stackrel{\gamma}{\rightarrow} \mu_{j+1}$, then either ($\nu_i \stackrel{\gamma}{\rightarrow} \nu_{i+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$), or ($\nu_i \Rightarrow \nu_k$ and $\nu_k \in \simeq$), or ν_i has no outgoing arcs; if $\mu_j \stackrel{\alpha}{\rightarrow} \mu_{j+1}$, then $\mu_{j+1} \simeq \nu_i$.

—if $\nu_i \stackrel{\gamma}{\rightarrow} \nu_{i+1}$, then either ($\mu_j \stackrel{\gamma}{\rightarrow} \mu_{j+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$), or there exists a ν_k such that ($R(\nu_k)$ and $\nu_k \simeq \mu_j$); if $\nu_i \stackrel{\tau}{\rightarrow} \nu_{i+1}$, then either ($\nu_{i+1} \simeq \mu_j$), or ($\mu_j \stackrel{\alpha}{\rightarrow} \mu_{j+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$), or ($\nu_i \Rightarrow \nu_k$ and $\nu_k \in \simeq$).

The nodes in G_{as} that are in relation are called *covered* nodes. Nodes with no outgoing arcs are covered by definition. If all the nodes of G_{as} are covered, then G_{as} is *completely covered;* otherwise it is *partially covered.* We extend this notion of coverage to arcs by saying that an arc is covered

when both its source and target nodes are covered. Analogously we say that the corresponding arcs in the symmetric G_{ac} are covered.

$G_{ac} \simeq G_{as}$ if and only if there exists a node ν_k such that $R(\nu_k)$, and ν_k is in relation with some G_{as} node and, for each covered recursive arc in G_{as}, the source and target nodes of that arc are in relation with source and target nodes of a correspondingly covered recursive arc in G_{ac}. △

The definition above allows us to compare AC and AS graphs. The two differences with respect to the standard notion of bisimulation are reflected in the definition, as follows.

The first difference appears in the first part of the definition, where we are checking that an AC graph properly simulates a given AS graph. In fact, we have to detect that an AC graph covers only a portion of the AS graph. When an AC graph performs a τ move, then this means that the component can change state autonomously, without interacting with the environment. If this happens, we only require that the node reached by the AC graph is in relation with some reachable node of the AS graph. This allows us to use the AC graph to partially fulfill the needs of an AS graph. As we discuss below, the AC and AS label structures are modified during the checking process so that it is possible to take into account partial successful matching. Note that this problem does not arise when we try to simulate an AC graph with an AS graph, since in fact we only have to check that the AC graph does not perform more actions than what is required by the AS graph.

The second difference with standard bisimulation is captured in the second part of Definition 4, which deals with the simulation of an AC graph by an AS graph. In our view, this corresponds to checking that the AC graph will always provide the portion of assumptions that it matches. This means that there should not exist any AC behavior that does not provide the matched context. In other words, all the possible AC behaviors must provide the given context.

6.2 Checking Algorithm

We can now define the checking algorithm. To do so, let us first define the notion of substitution.

Definition 5 (*Substitution*). A *substitution* is a set of pairs (AC, AS) such that the first element of any pair only occurs once in the set. We denote the empty substitution as ϵ and denote a generic substitution as $\sigma = [AC_1/AS_1, \ldots, AC_n/AS_n]$. △

Given a configuration Γ— that is, a set of ground components—$\sigma(\Gamma)$ is the system built out of the components in Γ and checked according to the associations in the substitution σ.

Definition 6 (*Checking Algorithm*). Let $\Gamma = \{C_1, C_2, \ldots, C_n\}$ be a configuration and σ be an empty substitution:

(1) If there are no AS graphs in Γ, then Check(Γ) = (true,σ) and terminate.

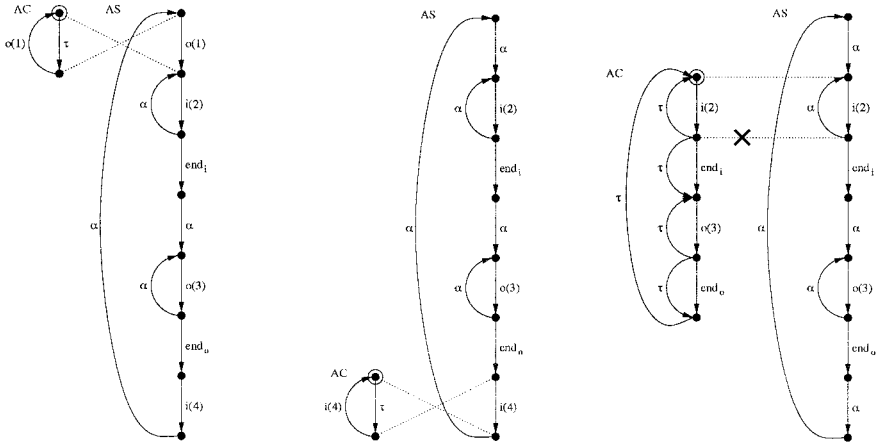

Fig. 5. Mismatch in actual and assumed behavior leading to deadlock.

(2) Try to find a pair $AC_{C_i} \simeq AS_{C_j}$. If no pair is found, then Check(Γ) = (false,σ) and terminate.
(3) Let $\sigma = \sigma \cup (AC_{C_i}/AS_{C_j})$.
(4) Obtain a new graph AC'_{C_j} by labeling as root nodes all the nodes that are reachable from the root through covered arcs (i.e., the predicate R is true on those nodes).
(5) If AS_{C_j} is completely covered, then remove AS_{C_j} from Γ and go to step 1. Otherwise, obtain a new graph AS'_{C_j} that reflects the partial match by labeling all covered arcs with α and go to step 2. △

Note that the method for selecting a pair of candidate AC and AS graphs in step 2 is not important for the purposes of this discussion. It is sufficient that, for a given configuration, the algorithm either terminates successfully on any one series of selections or terminates unsuccessfully on all series of selections. We address this point further in Section 6.3 as part of our discussion of complexity.

Let us see how we can apply these definitions to the Compressing Proxy example. We start by creating a configuration $\Gamma = \{\mathbf{GZ}, \mathbf{AD}, \mathbf{CF}_u, \mathbf{CF}_d\}$. This configuration follows the diagram of Figure 1 and, of course, contains several assumption graphs. We then select a pair (AC,AS) to put in relation. Assume it is the pair ($AC_{\mathbf{CF}_u}$, $AS_{\mathbf{AD}}$). This pair successfully puts in relation the two nodes of $AC_{\mathbf{CF}_u}$ with two of the nodes in $AS_{\mathbf{AD}}$, as shown on the left side of Figure 5. We obtain as a result a partially covered assumption graph for the adaptor, $AS'_{\mathbf{AD}}$. If we next select the pair ($AC_{\mathbf{CF}_d}$,$AS'_{\mathbf{AD}}$), then both the nodes of $AC_{\mathbf{CF}_d}$ match two nodes of $AS'_{\mathbf{AD}}$, as shown in the middle of Figure 5, resulting in a further matched assumption graph for the adaptor, $AS''_{\mathbf{AD}}$. Next, we select the pair ($AC_{\mathbf{GZ}}$,$AS''_{\mathbf{AD}}$), attempting to match the actual behavior of **gzip** with the remaining, uncovered part of the assumption graph of the adaptor. In this case, we are not able to relate all nodes in $AC_{\mathbf{GZ}}$ to the nodes in $AS''_{\mathbf{AD}}$. This is indicated

in Figure 5 by the large cross. The algorithm tries to find other pairs, but it is easy to see that it will not be able to match $AS''_{\mathbf{AD}}$, since its assumption could only be provided by **gzip.** Hence the algorithm, after all the possible attempts, will terminate at step 2.

It is worth noticing that the mismatch occurs exactly where the deadlock in the system appears. In particular, we cannot satisfy the assumption made by the adaptor in which it requires an $\mathbf{end}_i$ from the context. Thus, the adaptor will be blocked, not producing an $\mathbf{end}_o$, which in turn will cause **gzip** to block, thus achieving a state of deadlock.

Let us more precisely define what we mean by *deadlock*.

Definition 7 (*Deadlock*). Let S be a system with reaction rules T and final solution set F. We say that S is in deadlock if there exists a terminating computation $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$, $S_n \notin F$ and there exists an $m_j \in S_n$ such that $(m_j, m_{i+1}, \ldots, m_{i+k}) \rightarrow S_r$. $\triangle$

Notice that this definition characterizes a global deadlock; the last part of the definition avoids the possibility that the deadlock is caused by all components either reaching a final state or reaching an internal deadlock state (i.e., a local deadlock). We do this because it is global deadlock that is of interest to us here.

An important point is that it can never be the case that an AC arc becomes covered because the starting and ending nodes are covered when there exists no other AC graph performing the complementary transition. In particular, if the situation arises where an arc has starting and ending nodes that are roots, then this means that a partial match should have been performed such that a node became a root because the corresponding node in the AS graph has been covered—that is, it has been put in relation with a node in another AC graph. Due to the equivalence definition, which preserves the branching structure of the graphs, and due to the simplification hypothesis, which prevents graphs from having nodes with outgoing arcs labeled the same, the arc between the two covered nodes must have been followed.

Let us now state the correctness of our algorithm.

PROPOSITION 1. *Let* Γ *be a configuration. If* Check(Γ) = (bool, σ) *succeeds, i.e.,* bool = true and $\sigma \neq \epsilon$, *then* Γ *is deadlock free.* $\triangle$

PROOF. The proof is by contradiction.

Let us assume Γ is not deadlock free. Therefore, there should be a terminating computation $S_0 \rightarrow \ldots \rightarrow S_n$, and $S_n \notin F$. This means that there exists a molecule $m_i \in S_n$, describing the state of the component C_i, such that m_i does not represent a final state and is not inert. This means that there exists a rule T_j that can be applied to a suitable solution $S = m_i, m_{i+1}, \ldots, m_k$. By construction, in the AS graph associated with C_i, there should exist a node μ_i corresponding to m_i, since m_i represents a reachable state from the initial solution of C_i. Furthermore, from μ_i there is an arc due to the application of rule T_j labeled with $\lambda = m_{i+1}, \ldots, m_k$ and leading to the node μ_{i+p}.

By hypothesis we are considering a given configuration $\sigma(\Gamma)$. Therefore, there should exist an iteration in the checking algorithm that led to $\sigma(\Gamma)$ and covers the portion of the AS graph containing the arc from μ_i to μ_{i+p}. Now, by definition of equivalence, there should exist an AC graph of a component C_r whose root ν_j is in relation to the root μ_{i-h} of an outer tree that includes μ_i. The AC graph is then supposed to provide the required context λ when needed by the AS graph—that is, when the component C_i reaches the state m_i. In fact, by definition of equivalence, it cannot be the case that the arc becomes covered because the starting and ending nodes are covered when there exists no AC graph performing the corresponding transition. Thus, the only possibility is that the component C_r does not provide the required context because either it has that behavior but it does not provide it an infinite number of times, or it blocks before reaching it.

Now we proceed by induction on the distance of the μ_i node from the node μ_{i-h}, and show that actually the context has to be available.

Base Case: $h = 0$. This means that $\mu_i \simeq \nu_j$. By definition there exists an arc in the AC graph from which an arc labeled λ exits, and the reached nodes are in the equivalence. Therefore, due to the fact that ν_j is a root, the only possibility for a solution to block containing the molecule m_i is that the AS graph is recursive on the arc labeled λ, while the corresponding arc in the AC graph is not. This is clearly not possible, since the equivalence would have failed.

Inductive Case: $h = n + 1$. This means that there exists a path from μ_{i-h} to μ_i such that all the nodes lying on the path are in relation with some AC graph node. Let us consider the predecessor of μ_i in the path, μ_{i-1}. Then μ_{i-1} has to be in relation with some ν_x. Now, we have only to guarantee that from ν_x a node ν_i equivalent to μ_i is eventually reached. By the inductive hypothesis, we know that the AC graph actually provides the context to AS until the node μ_{i-1}. This means that the two components are progressing together until those nodes. Now, since μ_{i-1} and ν_x are in relation, and from μ_{i-1} the node μ_i is reachable, the only possibility is that from ν_x a node ν_i equivalent to μ_i cannot be reached. Due to the definition of equivalence and to the fact that we are considering the AC graph covering the portion of the AS graph containing the arc from μ_i to μ_{i+p}, this is possible only if ν_i is a node satisfying the root condition, and, therefore, it could be directly placed in relation to μ_i. The proof then proceeds as in the base case, resulting in a contradiction. □

The adaptor can be modified to eliminate the deadlock by introducing parallelism into its behavior, as discussed in Section 4. The modified component CHAM for the adaptor is shown in Table II. It replaces the phased behavior of the adaptor with nonblocking reads and writes. Figure 6 shows the AC and AS graphs obtained from the modified specification of the adaptor.

Let us now show how the matching process succeeds with the new specification for the adaptor. The process is shown in four parts. In Figure 7, the algorithm matches the two AC graphs of the filters with part of the AS graph

Table II. Modified Component CHAM for the Adaptor

	Adaptor (AD)
Syntax	$M' ::= P \mid C \mid E \mid M' \diamond M' \mid M' \parallel M'$ $P ::= \mathbf{AD} \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= \rho_3 \mid \rho_4 \mid \rho_5 \mid \rho_6$ $E ::= \mathbf{end}_i \mid \mathbf{end}_o$
Transformation Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2 \longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n),\ \mathbf{end}_i \diamond m_2 \diamond i(n) \longrightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o,\ m_2 \diamond i(n) \diamond \mathbf{end}_i$ $T'_4 \equiv m_1 \parallel m_2 \parallel \cdots \parallel m_k \longrightarrow m_1, m_2, \ldots, m_k$ $T'_5 \equiv \mathbf{AD} \diamond m \longrightarrow m \diamond \mathbf{AD}$
Initial Molecule	$i(\rho_3) \diamond o(\rho_4) \diamond \mathbf{end}_o \diamond \mathbf{AD} \parallel i(\rho_5) \diamond \mathbf{end}_i \diamond o(\rho_6) \diamond \mathbf{AD}$
Final Molecule	$\{m_1 \diamond \mathbf{AD}, m_2 \diamond \mathbf{AD}\}$

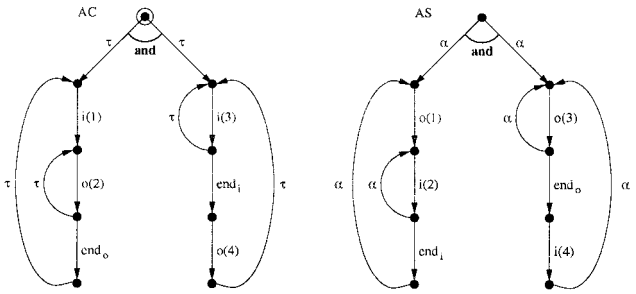

Fig. 6. AC and AS graphs for the modified adaptor.

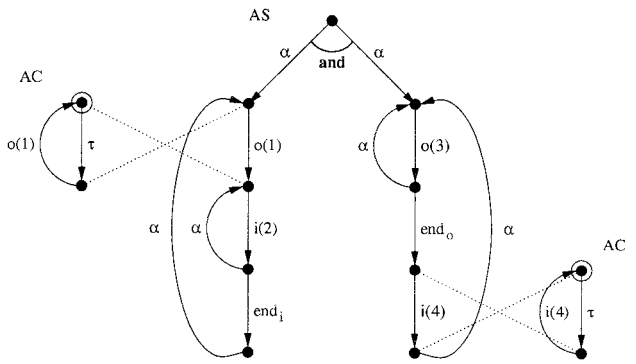

Fig. 7. Successful match of filter AC graphs against adaptor AS graph.

for the adaptor. The result is a partial match for the adaptor, where the partial match will be reflected both in the AS graph, by changing the matched label to α, and in the AC graph, by showing the root condition on the covered nodes. In Figure 8, the algorithm matches the AC graph of the adaptor with the AS graph for **gzip.** Note that this step would not have been possible without the earlier match involving the filters, which had the effect of moving the root condition in the AC graph of the adaptor to a point where it could match **gzip**

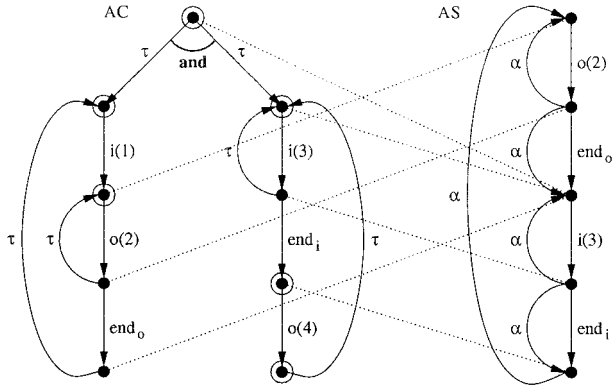

Fig. 8. Successful match of adaptor AC graph against **gzip** AS graph.

assumptions. In Figure 9, the algorithm matches the AC graph of **gzip** with the remaining assumptions of the adaptor. Finally, in Figure 10, the filter assumptions are matched with the AC graph of the adaptor. At this point, the assumptions of all four components have been satisfied.

6.3 Complexity

The algorithm presented above is designed to prove deadlock freedom of a system without building a complete finite-state model of the system. Thus, it attacks one of the main problems found in the practical use of known verification techniques, namely explosion in the size of the state space. In our approach, we only build finite representations of the individual components, which for each such component consists of equivalent-sized representations for its actual behavior and the assumed behavior of its context. Furthermore, the assumptions are automatically derived from the actual behavior, and used in the algorithm to check behavioral compatibility.

One of the main problems found in the practical use of known verification techniques is the size of the state space. Hence, we comment both on the algorithmic complexity and on the state space size of our proposal.

To check for deadlock freedom using standard approaches based on nondeterministic finite-state machines and reachability analysis, a global state space of the system is built, and then a full search for deadlock is performed. The size of the state space of a system composed of N concurrent components each of size $O(K)$ is $O(K^N)$. Here we are assuming that all components are of comparable size; the reasoning for components of different sizes would be similar. To avoid the problem of building the whole state space at once, techniques such as "on-the-fly" algorithms have been developed. However, in the worst case (i.e., if no deadlock is found) the whole state space must still be analyzed, which for reachability analysis has a time complexity of $O(K^N)$.

Our approach has the advantage that it dramatically reduces the state space to $O(KN)$, since the algorithm simply uses the AC and AS graphs for each of the N components. The state space constructed is not a cross

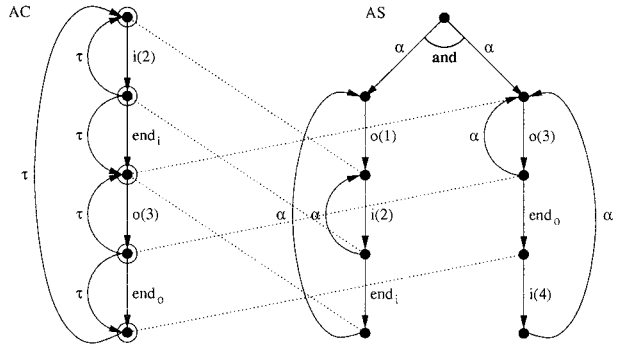

Fig. 9. Successful match of **gzip** AC graph against adaptor AS graph.

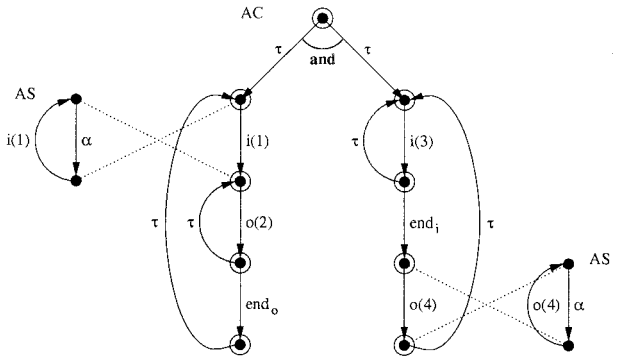

Fig. 10. Successful match of adaptor AC graph against filter AS graphs.

product of the graphs for each component. Instead what is constructed is just the single AC and AS graphs, which are much smaller in size. The algorithm needs $2N$ graphs of size K each, which leads to an $O(KN)$ state space.

The worst-case time complexity of our algorithm is $O(N!(K^2\log(K)))$, which is comparable to the worst case of standard reachability. Let us explain the reasoning that leads to this time complexity result.

The algorithm works by selecting a pair of AC and AS graphs and then checking them for compatibility. The check is similar to bisimulation, and in fact a standard bisimulation-checking algorithm can be used (see Definition 4). For purposes of this discussion, we assume that we can use a variant of the well-known Paige-Tarjan algorithm [Paige and Tarjan 1987], which has complexity $O(E\log(K))$, where K is the number of states, and E the number of arcs in the graph. Actually, E is bounded by K^2, and so a bound for our case is $O(K^2\log(K))$.

As we mention in Section 6.2, the order in which the graphs are compared is not important in explaining the functionality of the algorithm. However, that selection process has a significant impact on the complexity of the algorithm. In particular, we have the possibility of backtracking; if a particular choice of order results in an unsuccessful termination, then the

algorithm must try a different order. Given that there are N AS graphs and that we do not check an AC graph against its own AS graph, the algorithm will backtrack at most $(N - 1)!$ times. This worst-case behavior happens only if every possible order results in an unsuccessful termination—that is, the case in which the algorithm returns "false."

Finally, each AC graph is put in relation to an AS graph exactly once. Since we have N AC graphs, this occurs N times during the analysis of a system.

Thus, in the worst case, the overall complexity of the algorithm is $O(N((N - 1)!)(K^2\log(K)))$; N to put in relation all AC graphs, $(N - 1)!$ for all possible backtrackings, and $(K^2\log(K))$ to perform bisimulation on each candidate AC/AS pair. Simplifying the expression we get $O(N!(K^2\log(K)))$.

We would expect the typical time efficiency of the algorithm to be much better than the worst-case estimate of its complexity. In fact, in practice, the case of all possible backtrackings would not be very common, even using a random selection of candidate pairs. Moreover, although not explored in this paper, we foresee several heuristics that could improve the running time of the algorithm. These heuristics would be used to select "best" candidate AC and AS pairs, defining the order in which the graphs are checked, and hence dramatically increasing the efficiency of the algorithm. One simple example of such a heuristic is to use the signature of the graphs (i.e., the set of labels used in the graphs) to filter the set of candidates, since for an AC and an AS graph to be in relation it must be the case that the set of labels of the AC graph is included in the set of labels of the AS graph; AC graphs whose signatures are not included would not be considered. This reasoning is more complicated in the presence of *and* subgraphs, but the general intuition holds. When the components of a system behave differently from one another—by far the typical situation in practice—these and other such heuristics might prove to be extremely effective, while not being very hard to compute from an algorithmic point of view.

7. CONCLUSIONS AND FUTURE WORK

We have presented a method to statically check behavioral properties of a system at the architectural level. At this level, the properties of interest are mainly dynamic properties related to the *coordination* of components; a component has a potential behavior, but in order to be successfully integrated into an architecture, it expects the context to behave in some particular way. We introduced the notion of *assumptions* to formalize what a component expects from other components. In other words, in order to work together, components must agree not only on the actual behaviors (e.g., agree on communication protocol, port naming, and the like) but also on the assumptions they make about each other.

The checking algorithm uses the assumptions and actual behavior to verify that any differences cannot produce a deadlock situation. Clearly, this work needs to be generalized. We have introduced the basic concepts,

and we have presented an algorithm to check a particular problem in a particular situation. The case study of the Compressing Proxy shows that the algorithm can be useful in a real context. However, other properties of interest should be analyzed and algorithms developed to perform verification of those properties.

The idea of associating assumptions with components may have interesting consequences besides deadlock checking. In general, when components are assembled together to form a system, the verification performed is based on type checking of the interfaces. As mentioned in Section 2, some work has been done in checking the dynamics of components. But the notion of checking assumptions against actual behavior may lead to a general way of verifying that the assembly of a system, at the architectural level, is correctly done. The information in the interfaces, besides operations (or ports), types of the operations, and even potential behavior, might be enriched by the assumptions that the components make on how the context behaves. These considerations give additional motivation for generalizing the results presented here.

Our method is not limited to deadlock detection; it could be used as the basis for checking other safety and liveness properties as well. For example, a safety property that could be verified might be one related to proper access to shared resources. Suppose that there is a shared information resource that must be accessed in a particular way, such as by first initializing it, and then by querying and modifying it in some restricted order. This particular access protocol would be captured by the assumptions of the shared resource. The other components of the system would be required to respect this protocol. In general, one might have many shared resources with different protocols, and want to check for compatibility of all the accesses at once, since one access may depend on the others.

Using our method it is also possible to check for liveness properties. For instance, it might be possible to check whether each sender of a message in a message-based system always receives the corresponding acknowledgment for that message. The alternating bit protocol would be a small example of the application of this property.

These are just two of the possible properties that could be examined using the work described here, but further investigation is required in order to demonstrate how this would be realized. Notice that our method is more useful when global properties of the system are checked, because in those cases there is a clear advantage to building partial graphs as opposed to building the whole state space.

Driven by our case study we have defined the graphs and the equivalence based on a one-to-one communication between components. This can be seen in the transformation rules, whose left-hand-side arity is at most two. In general, more than two components can be involved in a communication and/or synchronization. Thus, both the structure of the AC and AS graphs and the definition of the equivalence have to be generalized. In terms of the graphs, this can be done by simply extending the definition of the labeling to be “set of labels.” Similarly, the matching algorithm can be extended by

modifying the portion of the equivalence that chooses a label from the set and by extending the notion of covered arc. A similar need for generalization applies to situations in which a nondeterministic point in an AS graph should match more than one AC graph. The algorithm as defined here is not able to perform the match, but we are currently working on extensions to the algorithm that address this weakness.

Another point worth mentioning is that we are able to prove that a certain configuration is "legal" only if it is possible to find a suitable way to match AS graphs with AC graphs. The matching algorithm as we have defined it is incremental. In fact, in our example, the successful matching can be found only by following a particular order in the processing. This is not surprising, since we want to be able to prove system-level dynamic properties by checking component-level adequacy. The strength of our approach is that we record partial successful matches in the graph structure, thus being able to easily accommodate the eventual checking of multiway communications and synchronizations.

As a last comment we would like to stress again that the technique we propose in this paper should not be seen as an alternative technique to more traditional model-checking techniques. Our intention is to propose it as a suitable complementary technique that, although incomplete, can be very effective in terms of its state space properties.

ACKNOWLEDGMENTS

The authors thank the associate editor, David Garlan, and the anonymous referees for their extensive and thoughtful comments, which have helped to greatly improve the paper. We also thank Sebastian Uchitel for his careful reading of the paper and his insightful suggestions for improving the checking algorithm.

REFERENCES

ALLEN, R. AND GARLAN, D. 1996. A Case Study in Architectural Modeling: The AEGIS System. In *Proceedings of the 8th International Workshop on Software Specification and Design* (March 1996), pp. 6–15. IEEE Computer Society.

ALLEN, R. AND GARLAN, D. 1997. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology 6,* 3 (July), 213–249.

BANÂTRE, J.-P. AND MÉTAYER, D. L. 1990. The Gamma Model and its Discipline of Programming. *Science of Computer Programming 15,* 55–77.

BANÂTRE, J.-P. AND MÉTAYER, D. L. 1993. Programming by Multiset Transformation. *Communications of the ACM 36,* 1 (Jan.), 98–111.

BERRY, G. AND BOUDOL, G. 1992. The Chemical Abstract Machine. *Theoretical Computer Science 96,* 217–248.

BOUDOL, G. 1994. Some Chemical Abstract Machines. In *A Decade of Concurrency,* Number 803 in Lecture Notes in Computer Science (May 1994), pp. 92–123. Springer-Verlag.

CAMPBELL, R. AND HABERMANN, A. 1974. The Specification of Process Synchronization by Path Expressions. In *Proceedings of an International Symposium on Operating Systems,* Number 16 in Lecture Notes in Computer Science (April 1974), pp. 89–102. Springer-Verlag.

COMPARE, D., INVERARDI, P., AND WOLF, A. 1999. Uncovering Architectural Mismatch in Component Behavior. *Science of Computer Programming 33,* 2 (Feb.), 101–131.

EDWARDS, S. AND WEIDE, B. 1997. WISR8: 8th Annual Workshop on Software Reuse Summary and Working Group Reports. *SIGSOFT Software Engineering Notes 22,* 5 (Sept.), 17–32.

GARLAN, D., ALLEN, R., AND OCKERBLOOM, J. 1995a. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software 12,* 6 (Nov.), 17–26.

GARLAN, D., KINDRED, D., AND WING, J. 1995b. Interoperability: Sample Problems and Solutions. Available from the authors.

HOARE, C. 1985. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey.

INVERARDI, P. AND PRIAMI, C. 1996. Automatic Verification of Distributed Systems: The Process Algebra Approach. In *Formal Methods in System Design,* pp. 7–38. Boston, Massachusetts: Kluwer Academic.

INVERARDI, P. AND WOLF, A. 1995. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering 21,* 4 (April), 373–386.

INVERARDI, P. AND YANKELEVICH, D. 1996. Relating CHAM Descriptions of Software Architectures. In *Proceedings of the 8th International Workshop on Software Specification and Design* (March 1996), pp. 66–74. IEEE Computer Society.

KOBAYASHI, N. 1998. A Partially Deadlock-Free Typed Process Calculus. *ACM Transactions on Programming Languages and Systems 20,* 2 (March), 436–482.

LISKOV, B. AND WING, J. 1994. A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems 16,* 6 (Nov.), 1811–1841.

LUCKHAM, D., KENNEY, J., AUGUSTIN, L., VERA, J., BRYAN, D., AND MANN, W. 1995. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering 21,* 4 (April), 336–355.

MAGEE, J., KRAMER, J., AND GIANNAKOPOULOU, D. 1997. Analysing the Behaviour of Distributed Software Architectures: A Case Study. In *Fifth IEEE Workshop on Future Trends of Distributed Computing Systems* (Oct. 1997), pp. 240–247.

MILNER, R. 1989. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey.

PAIGE, R. AND TARJAN, R. 1987. Three Partition Refinement Algorithms. *SIAM Journal on Computing 16,* 6, 973–989.

PERRY, D. 1989. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering* (May 1989), pp. 2–11. IEEE Computer Society.

PERRY, D. AND WOLF, A. 1992. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes 17,* 4 (Oct.), 40–52.

RADESTOCK, M. AND EISENBACH, S. 1994. What Do You Get From a Pi-calculus Semantics? In *Proceedings of PARLE'94 Parallel Architectures and Languages Europe,* Number 817 in Lecture Notes in Computer Science (1994), pp. 635–647. Springer-Verlag.

SHAW, M. AND GARLAN, D. 1996. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Englewood Cliffs, New Jersey.

SUMII, E. AND KOBAYASHI, N. 1998. A Generalized Deadlock-Free Process Calculus. In *3rd International Workshop on High-Level Concurrent Languages,* Volume 16 of *Electronic Notes in Theoretical Computer Science* (Sept. 1998). Elsevier.

WOLF, A., CLARKE, L., AND WILEDEN, J. 1989. The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE Transactions on Software Engineering 15,* 3 (March), 250–263.

ZAREMSKI, A. AND WING, J. 1997. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology 6,* 4 (Oct.), 333–369.

Received April 1998; revised December 1998 and September 1999; accepted February 2000