Checking Assumptions in Component Dynamics at the Architectural Level

Paola Inverardi
Dipartimento di Matematica
Universitá di L'Aquila
I-67010 L'Aquila, Italy
(inverard@univaq.it)

Alexander L. Wolf
Department of Computer Science
University of Colorado
Boulder, CO 80309 USA
(alw@cs.colorado.edu)

Daniel Yankelevich
Departmento de Computación
Universidad de Buenos Aires
Buenos Aires, Argentina
(dany@se.uba.ar)

University of Colorado
Department of Computer Science
Technical Report CU-CS-829-97 February 1997

ABSTRACT

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 begun to formulate a method that addresses this problem. The method operates at the architectural level so that behavioral integration errors, such as deadlock, can be revealed early in development. For each component, a specification is given both of its own interaction behavior and of the assumptions that it makes about the interaction behavior of the external context in which it expects to operate. We have defined an algorithm that, given such specifications for a set of components, performs "adequacy" checks between the component context assumptions and the component interaction behaviors. 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.

In this paper we give a first demonstration of the feasibility of our approach by showing its application to a system that contains incompatibilities between the assumptions and the interaction behaviors of two of its components. The algorithm successfully reveals the fact that the error can result in a deadlock of the system.

The work of A.L. Wolf was supported in part by the National Science Foundation under grant INT-95-14202.

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 [8, 12, 13], we developed a specification and analysis method for software architectures based on the CHAM (CHemical Abstract Machine) formalism [5]. 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 could be used instead to describe actual software systems. The method has proven to be useful for uncovering a variety of errors at the architectural level.

The method as we defined it, however, has a significant shortcoming. This shortcoming limits the method's usefulness when one is developing a system by assembling existing architectural components. In particular, the method depends on the specification and analysis of a system's *global* component interaction behavior. A more appropriate method would permit the specification of the *local* interaction behavior of an individual component. This would include both the actual behavior of the component and the assumptions it makes about the expected interaction behavior of other components. The method would then use the component specifications to discover mismatches among the components at system integration or configuration time.

We have begun to formulate a new method that takes this approach. Although currently based on the CHAM formalism, the method is likely to have wider applicability. In this method, rather than specifying whole systems and their global behavior, we specify individual components and their local behavior. We have defined an algorithm that, given such specifications for a set of components, performs "adequacy" checks between the component context assumptions and the component interaction behaviors. 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.

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 [10], and later by Compare, Inverardi, and Wolf [9]. The system contains incompatibilities between the assumptions and the interaction behaviors of two of its components. Our algorithm successfully reveals the known fact that the error can result in a deadlock of the system.

In the next section we briefly review related work. In Section 3 we provide some background on CHAM and our use of it for software architectures. In Section 4 we briefly describe the Compressing Proxy, which we use as an example target system for our method. The specification 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

Software architectures are structures of individual components that behave independently and interact. Moreover, the dynamics of these structures are of interest. In this line, it is not unexpected that many languages used to express concurrency semantics are borrowed to describe software architectures. Besides CSP and CHAM, other models have been used, such as the Pi Calculus [18] and Posets [14]. We believe that our approach is independent of the particular specification language used, but one advantage of the CHAM formalism is that it has not embedded within it any particular form of interaction. In most other languages, synchronous or asynchronous, broadcast or point-to-point communications are chosen.

From the perspective of Module Interconnection Languages, informal or semi-formal languages have been used to describe software architectures [20]. In those cases, it is more difficult to prove properties of the systems. Perry [16] presents a model in which the semantics of connections are taken into account to check when modules match. The semantic information in the modules, given as predicates, is used to verify some properties. However, as it was aimed to modules and assembly of modules, the dynamics of the system are not considered.

The use of sequences of actions associated locally to modules (components) to describe the behavior of the *allowed* interactions was introduced in Path Expressions [7]. In that work, a description of potential behavior is given by a regular expression in which atomic elements represented calls to the module.

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 [1, 2]. In their architectural description language Wright [19], each component has one or more *ports* that represent points of interaction with other components. Rather than interacting 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 [11]. A notion of consistency is introduced via a behavioral equivalence between the CSP agents describing the semantics of corresponding ports and roles. Although roles where introduced explicitly to support connector reuse, the idea is related to our notion of expected behavior. Roles, in a sense, describe the expected behavior for a particular port. However, consistency is checked only at the port level; it is not possible to verify properties that require several ports to interact among them. In other words, the internal behavior of the component is not taken into account, and complex evolutions are not captured by the equivalence. In the example introduced in our paper, we show how the behavior of the component is used in order to find such anomalies.

We can illustrate this point about roles and ports 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.

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 [5]. 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 [3], [4]. 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 CSP [11] and the CCS process calculus [15]. Boudol [6] 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* $m, m', \ldots$ defined as terms of a syntactic algebra that derive from a set of constants and a set of operations, and *solutions* $S, S', \ldots$ 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 CHAM specification contains *transformation rules* $T, T', \ldots$ that define a *transformation relation* $S \longrightarrow S'$ 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*.

The transformation rules can be of two kinds: general *laws* that are valid for all CHAMs and specific *rules* that depend on the particular CHAM being specified. The specific rules must be elementary rewriting rules that do not involve any premises. In contrast, the general laws are permitted such premises.

Solutions can be built from other solutions by combining them through the multiset union operator. For example, given solutions $S = m_1, \ldots, m_n$ and $S' = m'_1, \ldots, m'_k$, we obtain $S \uplus S' = m_1, \ldots, m_n, m'_1, \ldots, m'_k$ that is another solution.

CHAMs obey four general laws. Two of those laws are of relevance here.

The Reaction Law. An instance of the right-hand side of a rule can replace the corresponding instance of its left-hand side. Thus, given the rule

$$M_1, M_2, \ldots, M_k \longrightarrow M'_1, M'_2, \ldots, M'_l$$

if $m_1, m_2, \ldots, m_k$, and $m'_1, m'_2, \ldots, m'_l$ are instances of the $M_{1\ldots k}$ and $M'_{1\ldots l}$ by a common substitution, then we can apply the rule and obtain the following solution transformation.

$$m_1, m_2, \ldots, m_k \longrightarrow m'_1, m'_2, \ldots, m'_l$$

The Chemical Law. Reactions can be performed freely within any solution, as follows.

$$\frac{S \longrightarrow S'}{S \uplus S'' \longrightarrow S' \uplus S''}$$

In words, when a subsolution evolves, the supersolution in which it is contained is also considered to have evolved.

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*.

When applying the formalism to software architecture, we structure specifications into three parts [12]:

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.

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 [17], 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.

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. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration.

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 [10].

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 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-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.

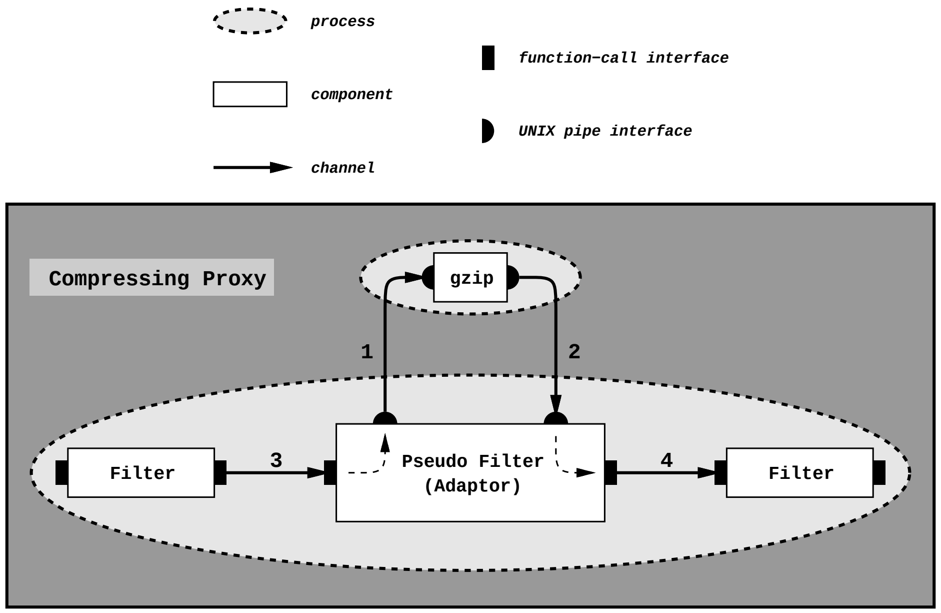

Figure 1: The Compressing Proxy.

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 local 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 have the adaptor handle the data incrementally and use non-blocking 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. (We note, however, that the system was indeed initially developed according to the first approach, and the architectural mismatch was not detected until the system was implemented and in use.) Nevertheless, one can see that it is representative of an all-too-common problem 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*. The details of the composition process are beyond the scope of this paper. Here, we are concerned only with how to specify component CHAMs and how they can be checked pairwise for compatibility.

5.1 Component CHAMs

To specify a component CHAM, we give a syntax for the molecules representing the component, transformation rules describing the behavior of the component, and an initial solution of molecules representing the initial state of the component. For the Compressing Proxy we must specify four component CHAMs, as shown in Table 1.

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 this paper. In fact, a detailed understanding of the specifications are unnecessary to follow the discussion below. Therefore, we only give a high-level and incomplete description of the specifications here.

Consider the upstream CERN filter. The syntax for molecules M representing this component consists of the set P representing the name of the component's processing element $\mathbf{CF}_u$ and a placeholder Φ to refer to other components with which $\mathbf{CF}_u$ interacts. M also includes the set C representing 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. 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. Finally, the syntax includes an infix operator "$\diamond$" 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 left-most position (i.e., the left operand of the left-most "$\diamond$" 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.

The interaction behavior of the upstream filter component is captured using two transformation rules. The first rule is a general inter-element communication rule that generically describes pairwise input/output communication between processing elements. Notice that this same rule is found in all four component CHAMs. The second transformation rule allows $\mathbf{CF}_u$ to iterate its communication behavior.

The initial solution 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.

	Upstream CERN Filter ($\mathbf{CF}_u$)	**Downstream CERN Filter ($\mathbf{CF}_d$)**
Syntax	$M ::= P \mid C \mid M \diamond M$ $P ::= \mathbf{CF}_u \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= n_1$	$M ::= P \mid C \mid M \diamond M$ $P ::= \mathbf{CF}_d \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= n_1$
Trans. Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2$ $\longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv \mathbf{CF}_u \diamond c \longrightarrow c \diamond \mathbf{CF}_u$	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2$ $\longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv \mathbf{CF}_d \diamond c \longrightarrow c \diamond \mathbf{CF}_d$
Init. Sol.	$\mathbf{CF}_u \diamond o(n_1)$	$\mathbf{CF}_d \diamond i(n_1)$

	GZIP (GZ)	**Adaptor (AD)**
Syntax	$M ::= P \mid C \mid E \mid M \diamond M$ $P ::= \mathbf{GZ} \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= n_1 \mid n_2$ $E ::= \mathbf{eof}_i \mid \mathbf{eof}_o$	$M ::= P \mid C \mid E \mid M \diamond M$ $P ::= \mathbf{AD} \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= n_1 \mid n_2 \mid n_3 \mid n_4$ $E ::= \mathbf{eof}_i \mid \mathbf{eof}_o$
Trans. Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2$ $\longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{eof}_o \diamond m_1 \diamond o(n),\ \mathbf{eof}_i \diamond m_2 \diamond i(n)$ $\longrightarrow m_1 \diamond o(n) \diamond \mathbf{eof}_o,\ m_2 \diamond i(n) \diamond \mathbf{eof}_i$ $T_4 \equiv \mathbf{eof}_i \diamond m \longrightarrow m \diamond \mathbf{eof}_i$ $T_5 \equiv \mathbf{GZ} \diamond m \longrightarrow m \diamond \mathbf{GZ}$	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2$ $\longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{eof}_o \diamond m_1 \diamond o(n),\ \mathbf{eof}_i \diamond m_2 \diamond i(n)$ $\longrightarrow m_1 \diamond o(n) \diamond \mathbf{eof}_o,\ m_2 \diamond i(n) \diamond \mathbf{eof}_i$ $T_4 \equiv \mathbf{eof}_i \diamond m \longrightarrow m \diamond \mathbf{eof}_i$ $T_5 \equiv \mathbf{AD} \diamond i(n_3) \diamond m$ $\longrightarrow i(n_2) \diamond \mathbf{eof}_i \diamond o(n_4) \diamond \mathbf{AD}$ $T_6 \equiv \mathbf{AD} \diamond i(n_2) \diamond m$ $\longrightarrow i(n_3) \diamond o(n_1) \diamond \mathbf{eof}_o \diamond \mathbf{AD}$
Init. Sol.	$i(n_1) \diamond \mathbf{eof}_i \diamond o(n_2) \diamond \mathbf{eof}_o \diamond \mathbf{GZ}$	$i(n_3) \diamond o(n_1) \diamond \mathbf{eof}_o \diamond \mathbf{AD}$

Table 1: Component CHAMs for the Compressing Proxy Example.

The CHAMs for the other three components follow a similar structure and share similar transformation rules. The critical issue for this example is the interaction behaviors of **gzip** and the adaptor, so let us 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 when communications through **AD** and **GZ** take place; $\mathbf{eof}_i$ denotes "input end of file", while $\mathbf{eof}_o$ denotes "output end of file". Both component CHAMs share transformation rules T_2 through T_4, which govern the iteration of the input and output behaviors involving data markers $\mathbf{eof}_i$ and $\mathbf{eof}_o$. Rule T_5 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_6. 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.

As mentioned above, when component CHAMs are integrated to form a system, a certain amount of configuration must occur. For instance, in the Compressing Proxy example, the symbolic communication channels referred to by the individual CHAMs as elements of N are instantiated according to the channel numbers in the diagram of Figure 1, resulting in actual connections being established between the components. Thus, although not obvious from the preceding discussion, the configuration operation would cause the symbolic channel n_1 of the upstream filter and the symbolic channel n_3 of the adaptor to be identified with the actual channel labeled "3" in the diagram.

5.2 Deriving Actual and Assumed Behaviors

In order to check for compatibility between components, we need suitable representations of the actual behavior of a component, AC, and assumed behavior of the external context, AS. For each component, we derive these two representations from its component CHAM specification.

The model for both representations is a 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 or where $or \subseteq \mathcal{P}(A)$.

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 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 molecule that is deleted or transformed by the rule. 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_2 of $\mathbf{CF}_u$.

Definition 1 *(AC graph for a component CHAM)*
AC graphs are defined constructively as follows.

- *The root node is labeled by 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* or *relation are constructed as follows.*

  - *The molecule associated with ν_i is the molecule obtained by modifying m_ν with r.*
  - *The arc connecting ν to ν_i is labeled with τ if r can be applied to a solution that contains only m_ν.*
  - *The arc is labeled λ if λ is the label consumed by r when applied to m_ν.*
  - *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* or *arcs.*

Informally, *or* arcs identify alternative subgraphs for the same component. As discussed below, this corresponds to a concurrent (i.e., multi-threaded) 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, AS graphs refer 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.

In assumption graphs, nodes are labeled differently in order to distinguish the molecules that can participate in each transformation. As mentioned above, the graph structure allows us to recover the ordering in which a component asks for molecules from the context. In general, more molecules can participate in producing the required total context. Thus, we identify nodes with numbers, but we can replicate a node whenever a potentially different molecule can provide the required context. In other words, nodes with the same number refer to the same solution, and therefore are associated with the same node in the corresponding AC graph, but they are considered different if the required context can come from different molecules. This helps during the matching phase. If a single molecule produces a subset of the context, then its actual behavior must have the structure of an AS subgraph.

Given an AC graph for a component CHAM we can define the corresponding AS graph. AS graphs have nodes labeled with natural numbers.

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}. G_{as} can have replicated nodes.*
- *The root node of G_{as} has the label* 0 *and is associated with the root node of G_{ac}.*
- *Let ν be a node in G_{as}, let k be its label, and let μ be the associated node in G_{ac}. Then if μ has an outgoing arc to a node μ_1 labeled λ ($\lambda \neq \tau$) due to the application of a rule r, then ν has an outgoing arc to the node corresponding to μ_1 labeled with the conjunction of the labels*

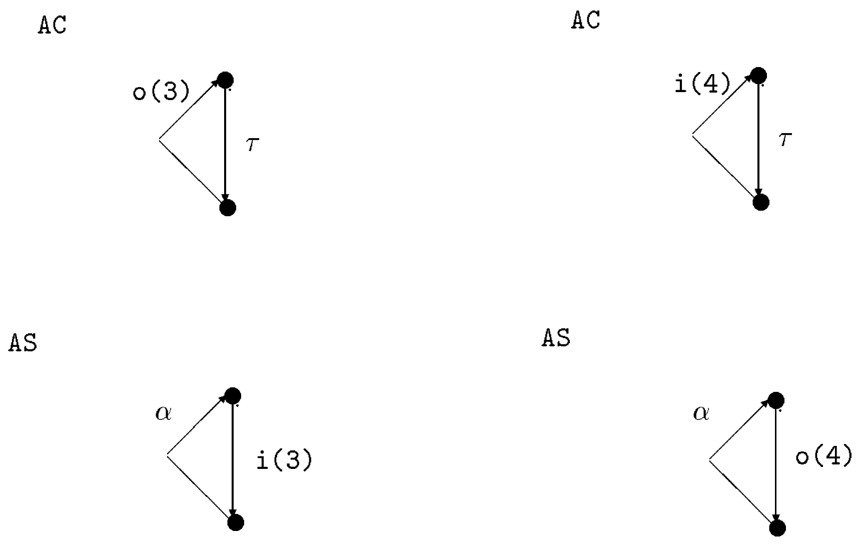

Figure 2: AC and AS Graphs for the Upstream (left) and Downstream (right) Filters.

consumed by r. Each such label corresponds to the consumed label of a molecule required in the context to perform the reaction by r. If the reached node does not already exist, then its label is $k + 1_j$ if there are already $j - 1$ children of ν. If the outgoing arc in G_{ac} is labeled with τ and the label of the reached node is less than k, then the outgoing arc from ν is labeled with α. In general, the node is replicated unless it is possible to show that the external molecule participating in the reaction is a transformation of the preceding one.

- *if μ has* or *arcs, then ν also has corresponding* or *arcs. If the AC arcs are labeled with τ then the corresponding AS arcs are labeled with α.*

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 *or* arcs when the transformation in the actual behavior graph has not required any context. Note that this is the only case in which an α arc can label a forward arc.

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

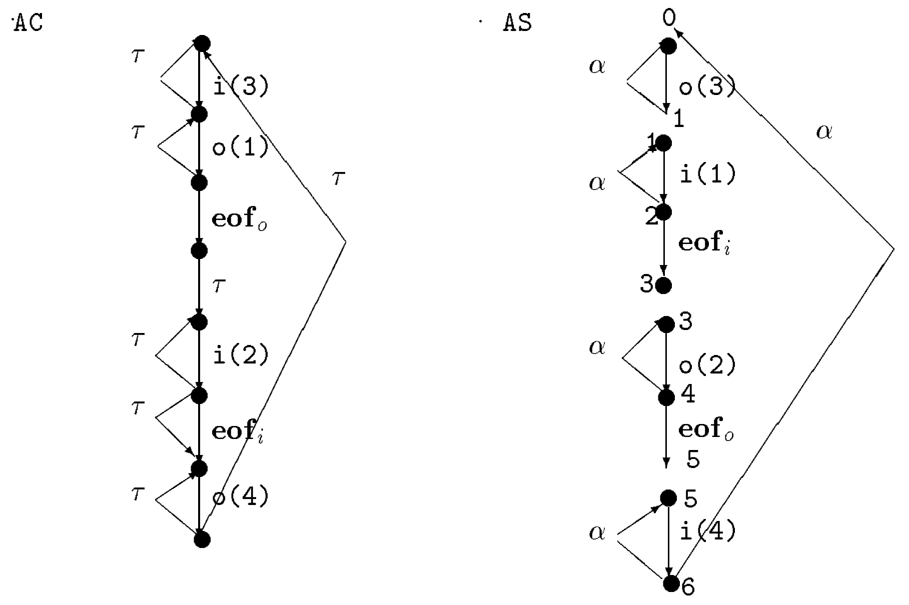

Figure 3: AC and AS Graphs for the Adaptor.

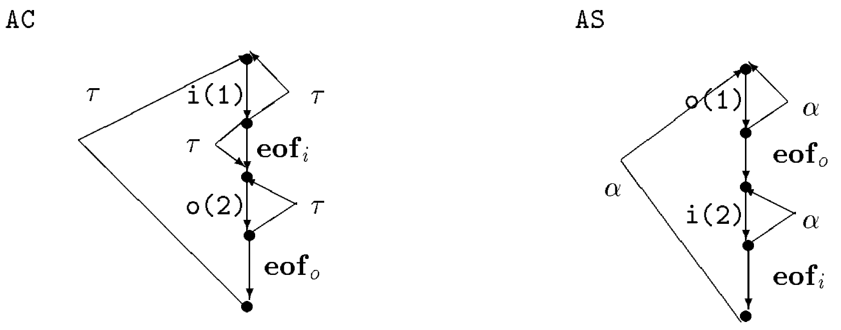

Figure 4: AC and AS Graphs for gzip.

6 Checking Assumptions

The primary goal of this work 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. In this work 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 configuration phase 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.

If a configuration phase *succeeds*, then the system is deadlock free. If the configuration phase *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. This is not enough to conclude that the whole system blocks, but we can iterate the checking phase, reducing the actual behavior of the blocking components. In other words, we can eliminate the part that can block and see how this affects other components.

6.1 Checking Algorithm

The checking algorithm is built upon a notion of equivalence that allows us to compare AC graphs with AS graphs. The equivalence relation allows nodes and arcs to be put in relation. Actually, since an AS graph can be fulfilled by more than one AC graph, we try to put in relation one AS graph with more than one AC graph. The idea is that all the arcs and nodes of the AS graph have to be covered and there should not exist the possibility that an actual behavior cannot provide the required information.

In the following, we denote by $\xrightarrow{\gamma^*}$ any sequence of transformations, suitably labeled, including the empty transformation. We also consider the following equivalence on transformations

$$\nu_i \xrightarrow{\alpha} \nu_{i+1} \xrightarrow{\gamma} \nu_{i+2} = \nu_{i+1} \xrightarrow{\gamma} \nu_{i+2}$$

Analogously, the following also holds.

$$\nu_i \xrightarrow{\gamma} \nu_{i+1} \xrightarrow{\alpha} \nu_{i+2} = \nu_i \xrightarrow{\gamma} \nu_{i+2}$$

Definition 3 *Let G_{ac} be an actual behavior graph, G_{as} be an assumption graph, and $\gamma \in \Lambda \setminus \alpha$, then two nodes are related, $\nu_i \simeq \mu_j$:*

- *if $\mu_j \xrightarrow{\gamma} \mu_{j+1}$, then also $\nu_i \xrightarrow{\gamma} \nu_{i+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$;*
- *if $\mu_j \xrightarrow{\alpha} \mu_{j+1}$ then $\nu_i \xrightarrow{\gamma^*} \nu_{i+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$ or ν_{i+1} is already a covered node.*
- *if $\nu_i \xrightarrow{\gamma} \nu_{i+1}$ and $\mu_j \not\xrightarrow{\gamma} \mu_{j+1}$ then either there exists a node ν_k such that $\nu_k \simeq \mu_j$ and $\nu_{i+1} \xrightarrow{\gamma^*} \nu_k$ or there exists a ν_k such that if $to(A) = \nu_k$ then $so(A) = \nu_r$, $l(A) = \alpha$, A is a or arc and ν_i is a descendant of ν_r but not of ν_k.*

The two graphs are related if and only if all the nodes in G_{ac} are in relation with G_{as} nodes. The G_{as} nodes in relation are called covered *nodes. If the G_{as} nodes are all covered, then the G_{as} graph is* completely covered*, otherwise it is* partially covered.

The above definition allows us to compare AC and AS graphs. Note that in this way we require that an actual behavior must completely match (part of) an assumption.

This definition derives to some extent from the well known Milner bisimulation. Here we do not require complete matching between components and we also have to take into account the potential concurrent behavior of a component. In this respect, the last condition in the above definition says that if an actual behavior performs something that is not required from the assumptions, then this is not harmful if either along that derivation it is possible to reach a solution that allows the required context or if there exists a concurrent component behavior that can actually provide the context.

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

Definition 4 *(Substitution) A* substitution *is a set of pairs (AC, AS). We denote with ϵ the empty substitution and a generic substitution $\sigma = [AC_1/AS_1, \ldots, AC_n/AS_n]$.*

Given a configuration Γ—that is, a set of components—we identify with the notation $\sigma(\Gamma)$ the system built out of the component in Γ according to the association in the substitution σ.

Definition 5 *(Matching Algorithm)*

Let $\Gamma = \{C1, C2, \ldots, Cn\}$ be a configuration and σ be an empty substitution.

1. *If in Γ there are no more AS graphs then Match(Γ) = (true, σ).*
2. *Try to find a pair $AC_{Ci} \simeq AS_{Cj}$.*
3. *If AS_{Cj} is partially covered, obtain a new graph AS'_{Cj} that reflects this partial match by labeling all covered arcs with α; let $\sigma = \sigma \cup \{AS_{Cj}, AC_{Ci}\}$; go to step 2.*
4. *Remove from Γ the assumptions AS_{Cj}; go to step 1.*

The following propositions, which we give here without proof, hold.

Proposition 1 *Let Γ be a configuration if Match(Γ) = (bool, σ) succeeds—that is, bool = true and $\sigma \neq \epsilon$, then the system $\sigma(C)$ is* deadlock free.

Proposition 2 *Let Γ be a configuration, C be any given component, AS be the assumption graph of C. If there are no* or *arcs and there is no actual behavior AC in Γ that satisfies the assumptions AS (that is, $AC \not\simeq AS$) then for any possible substitution AC/AS there exists a computation such that C will block.*

Proposition 2 might be generalized to the case with *or* arcs, showing that the algorithm is *complete*. However, the case with *or* arcs is much more complex and requires further, detailed analysis.

Let us now see how we can apply these definitions to our example. The aim is to define a single system out of the four components specified in Table 1. We start with a configuration $\Gamma = \{\mathbf{GZ}, \mathbf{AD}, \mathbf{CF}_u, \mathbf{CF}_d\}$, and we try to apply the algorithm. The first thing to do is try to find a possible pair (AC,AS). We try with the pair ($AC_{\mathbf{CF}_u}$, $AS_{\mathbf{AD}}$). This pair succeeds, since we can put in relation all nodes of $AC_{\mathbf{CF}_u}$ with nodes in $AS_{\mathbf{AD}}$. We obtain as a result a partially covered assumption graph for the adaptor, $AS'_{\mathbf{AD}}$. Analogously, it happens for the pair ($AC_{\mathbf{CF}_d}$, $AS'_{\mathbf{AD}}$), thus resulting in the assumption graph $AS''_{\mathbf{AD}}$. Now we can attempt to match the actual behavior of **gzip** with the remaining 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 $AS''_{\mathbf{AD}}$. Figure 5 illustrates this mismatch problem.

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

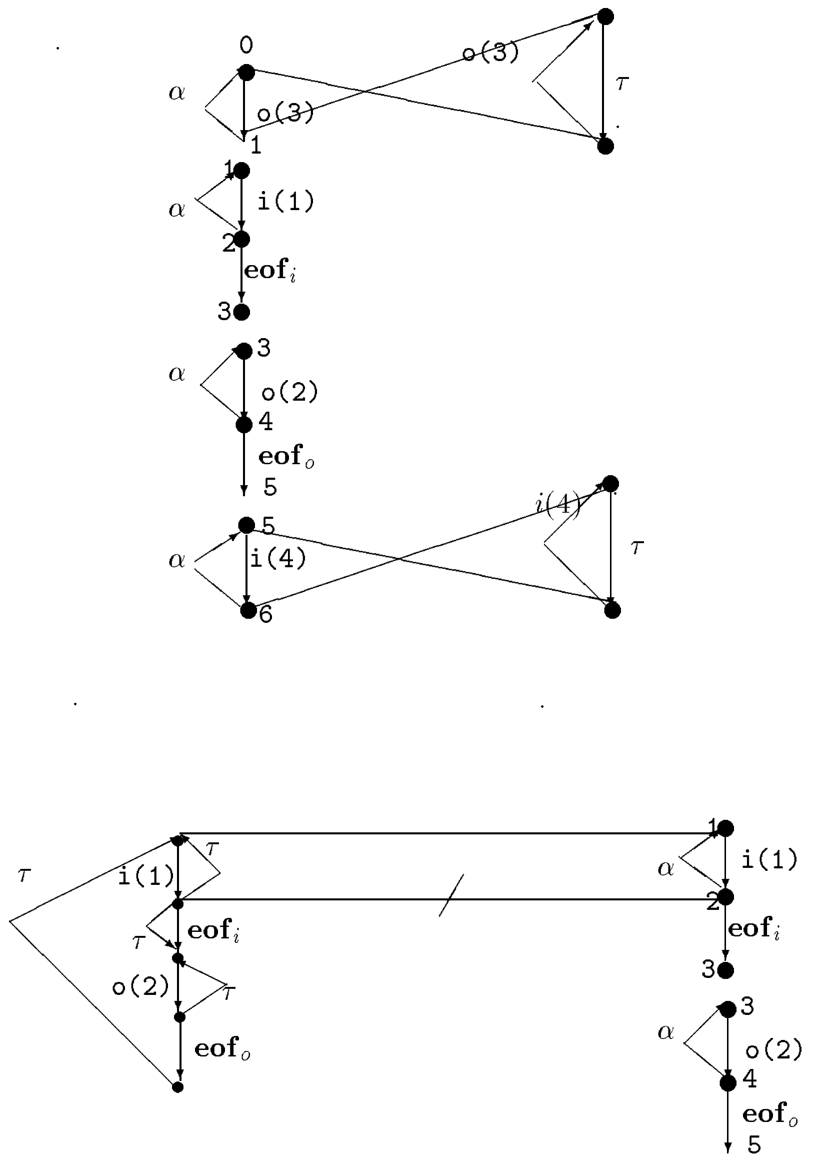

Figure 5: Mismatch in Actual and Assumed Behavior Leading to Deadlock.

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 2. It replaces the phased behavior of the adaptor with non-blocking reads and writes. Figure 6 shows the AC and AS graphs obtained from the modified specification of the adaptor. The successful match in behavior derived from the checking algorithm is illustrated in Figure 7.

Adaptor (AD)

	$M' ::= P \mid C \mid E \mid M' \diamond M' \mid M' \,\|\, M'$
	$P ::= \mathbf{AD} \mid \Phi$
Syntax	$C ::= i(N) \mid o(N)$
	$N ::= n_1 \mid n_2 \mid n_3 \mid n_4$
	$E ::= \mathbf{eof}_i \mid \mathbf{eof}_o$
	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2 \longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$
	$T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$
Trans. Rules	$T_3 \equiv \mathbf{eof}_o \diamond m_1 \diamond o(n),\ \mathbf{eof}_i \diamond m_2 \diamond i(n) \longrightarrow m_1 \diamond o(n) \diamond \mathbf{eof}_o,\ m_2 \diamond i(n) \diamond \mathbf{eof}_i$
	$T_4 \equiv \mathbf{eof}_i \diamond m \longrightarrow m \diamond \mathbf{eof}_i$
	$T_5' \equiv m_1 \,\|\, m_2 \,\|\, \cdots \,\|\, m_k \longrightarrow m_1,\ m_2, \ldots, m_k$
	$T_6' \equiv \mathbf{AD} \diamond m \longrightarrow m \diamond \mathbf{AD}$
Init. Sol.	$i(3) \diamond o(n_1) \diamond \mathbf{eof}_o \diamond \mathbf{AD} \,\|\, i(n_2) \diamond \mathbf{eof}_i \diamond o(n_4) \diamond \mathbf{AD}$

Table 2: Modified Component CHAM for the Adaptor.

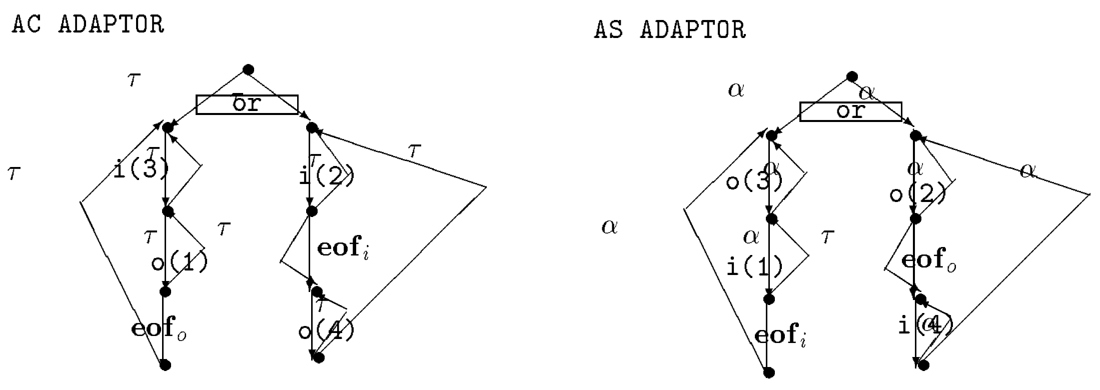

Figure 6: AC and AS Graphs for Modified Adaptor Component.

7 Conclusions and Future Work

In this work, we have presented an algorithm to check 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; one component has a potential behavior, but in order to be successfully integrated in 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, an the like) but also on the assumptions they make of each other.

We used the CHAM formalism to specify component behavior. The CHAM has been previously

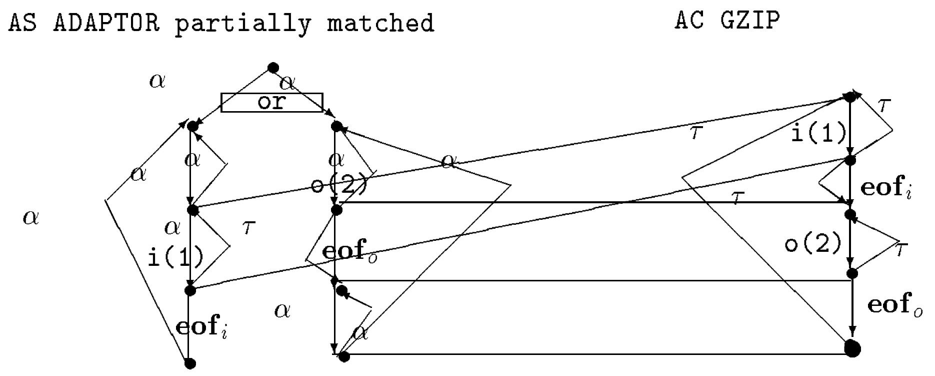

Figure 7: Successful Match of Components.

used to define software architectures and to analyze properties of architectures.

The checking algorithm introduced uses the assumptions and actual behavior to verify that the differences between the actual behavior of a component and the assumptions that other components make of what its behavior would be cannot produce a deadlock situation. We have shown how the algorithm works in a case study.

Clearly, this work needs to be generalized. We have introduced the notions needed, and we have presented an algorithm to check a particular problem in a particular situation. The example shows that the algorithm is useful in a real context. However, other properties of interest should be analyzed and algorithms developed to perform verification of those properties.

Moreover, the idea of associating assumptions to 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 the introduction, 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 component makes on how the context behaves. These considerations give additional motivation to generalize the results given in this work.

REFERENCES

[1] R. Allen and D. Garlan. Formalizing Architectural Connection. In *Proceedings of the 16th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, May 1994.

[2] R. Allen and D. Garlan. A Case Study in Architectural Modeling: The AEGIS System. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 6–15. IEEE Computer Society, March 1996.

[3] J.-P. Banâtre and D. Le Métayer. The Gamma Model and its Discipline of Programming. *Science of Computer Programming*, 15:55–77, 1990.

[4] J.-P. Banâtre and D. Le Métayer. Programming by Multiset Transformation. *Communications of the ACM*, 36(1):98–111, January 1993.

[5] G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.

[6] G. Boudol. Some Chemical Abstract Machines. In *A Decade of Concurrency*, number 803 in Lecture Notes in Computer Science, pages 92–123. Springer-Verlag, May 1994.

[7] R.H. Campbell and A.N. Habermann. 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, pages 89–102. Springer-Verlag, April 1974.

[8] D. Compare and P. Inverardi. Modelling Interoperability by CHAM: A Case Study. In *Proceedings of the First International Conference on Coordination Models and Languages*, number 1061 in Lecture Notes in Computer Science, pages 428–431. Springer-Verlag, April 1996.

[9] D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. Technical Report CU-CS-828-97, Department of Computer Science, University of Colorado, Boulder, Colorado, February 1997.

[10] D. Garlan, D. Kindred, and J.M. Wing. Interoperability: Sample Problems and Solutions. Technical report, Carnegie Mellon University, Pittsburgh, Pennsylvania, In preparation.

[11] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[12] P. Inverardi and A.L. Wolf. Formal Specification and Analysis of Software Architectures using the Chemical Abstract Machine Model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[13] P. Inverardi and D. Yankelevich. Relating CHAM Descriptions of Software Architectures. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 66–74. IEEE Computer Society, March 1996.

[14] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[15] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[16] D.E. Perry. The Inscape Environment. In *Proceedings of the 11th International Conference on Software Engineering*, pages 2–11. IEEE Computer Society, May 1989.

[17] D.E. Perry and A.L. Wolf. Foundations for the Study of Software Architecture. *SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.

[18] M. Radestock and S. Eisenbach. 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, pages 635–647. Springer-Verlag, 1994.

[19] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline.* Prentice-Hall, Englewood Cliffs, New Jersey, 1996.

[20] A.L. Wolf, L.A. Clarke, and J.C. Wileden. The AdaPIC Tool Set: Supporting Interface Control and Analysis Throughout the Software Development Process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.