Behavioral Type Checking of Architectural Components Based on Assumptions

Paola Inverardi

Dipartimento di Matematica
Universitá dell' 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-861-98 April 1998

© 1998 Paola Inverardi, Alexander L. Wolf, and Daniel Yankelevich

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.

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, 14, 15], 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. One class of such errors involves mismatches in architectural components [11], where assumptions made by different components are in conflict. We showed how our method could be used to uncover architectural mismatch in component behavior [9].

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 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 "legal" 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 [12], 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. Using a corrected version of one of the components, the system is then shown to be free of deadlock.

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 [20] and Posets [16]. 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 Languages, informal or semi-formal languages have been used to describe software architectures [22]. In those cases, it is more difficult to prove properties of the systems. Perry [18] 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, since it was aimed at modules and assembly of modules, the dynamics of the system are not considered.

The use of sequences of actions associated locally with components to describe permissible 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 represent calls to 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 [2, 1]. In their architectural description language Wright [21], 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 [13]. A notion of consistency is introduced via a behavioral equivalence between the CSP agents describing the semantics of corresponding ports and roles. Although roles were 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. In other words, the internal behavior of the component is not taken into account, and complex dynamics are not captured by the equivalence. In the example introduced here, 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.

Zaremski and Wing [23] describe a technique called *specification matching* that is intended primarily 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" [10]. 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 [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 [13] and the CCS process calculus [17]. 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 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* $T, T', \ldots$ 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*. Transformation rules can be *conditional*, in that their application may depend 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 [14] 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 [19], 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. In process algebras like CCS [17], a final state is identified using a special process, denoted as "nil", that does not perform any action.

Notice that the preceding discussion of basic concepts is given mainly in terms of a monolithic specification of a whole system. One of the contributions of this work 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*.

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 [12].

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.

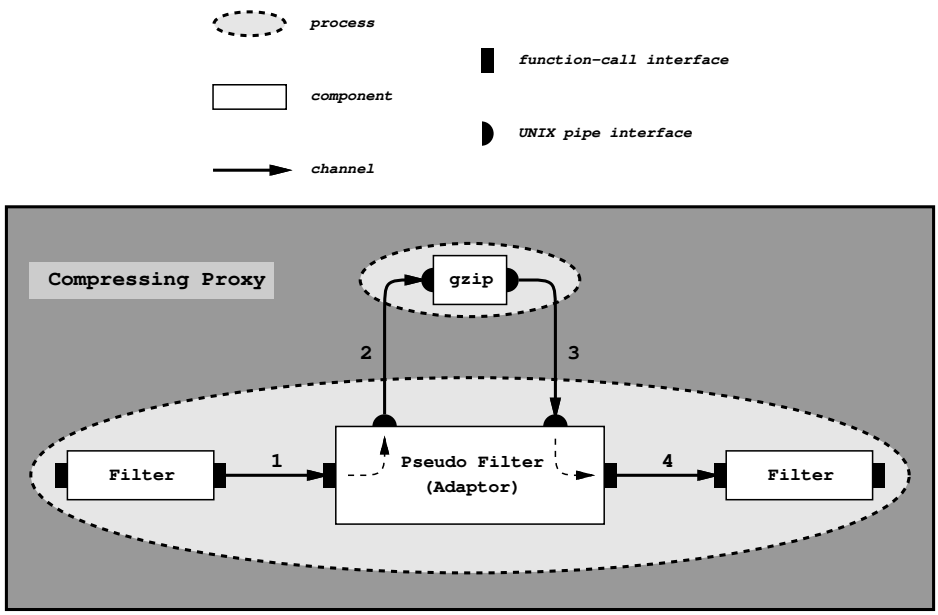

Figure 1: The Compressing Proxy.

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-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 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. 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*.

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 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 $\mathbf{CF}_u$. 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 meta-variable Φ to refer to the unknown syntactic structure of other components with which $\mathbf{CF}_u$ is expected to interact. As discussed in Section 5.2, meta-variables are instantiated (and thereby eliminated) as a side-effect of configuring component CHAMs into a system CHAM. 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 meta-variables 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 left-most position (i.e., the left operand of the left-most "$\diamond$" operator) in the molecule

	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	$T_1 \equiv i(n_1) \diamond m,\ o(n_1) \diamond \mathbf{CF}_u \longrightarrow m \diamond i(n_1),\ \mathbf{CF}_u \diamond o(n_1)$ $T_2 \equiv \mathbf{CF}_u \diamond o(n_1) \longrightarrow o(n_1) \diamond \mathbf{CF}_u$	$T_1 \equiv i(n_1) \diamond \mathbf{CF}_d,\ o(n_1) \diamond m \longrightarrow \mathbf{CF}_d \diamond i(n_1),\ m \diamond o(n_1)$ $T_2 \equiv \mathbf{CF}_d \diamond i(n_1) \longrightarrow i(n_1) \diamond \mathbf{CF}_d$
Init. Mol.	$\mathbf{CF}_u \diamond o(n_1)$	$\mathbf{CF}_d \diamond i(n_1)$
Final Mol.	$\mathbf{CF}_u \diamond o(n_1)$	$\mathbf{CF}_d \diamond i(n_1)$

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

Table 1: Component CHAMs for the Compressing Proxy Example.

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. 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 inter-element 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 $\mathbf{CF}_d$, and a generic version of the rule is found in the component CHAMs for **AD** and **GZ**. The molecule m appearing in the rule is understood to range over the set M that corresponds to a suitable instantiation of Φ, the meta-variable serving as a placeholder for other components. In other words, once $\mathbf{CF}_u$ has been configured to interact with a particular other component, m will derive from the syntax for that other component.

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.

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 non-full. 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 meta-variables 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.

More formally, the syntax for a system CHAM, Σ_S, is given by:

$$\Sigma_S = \Sigma_{C_1} \cup \Sigma_{C_2} \cup \ldots \cup \Sigma_{C_n}$$

where $\Sigma_{C_1} \ldots \Sigma_{C_n}$ are the syntaxes of the component CHAMs, and a suitable substitution is given for the meta-variables that relate across the component CHAMs. For the Compressing Proxy, the substitution would cause the symbolic channel n_1 of the upstream filter and the symbolic channel n_1 of the adaptor to be identified, and thus correspond to the channel labeled "1" in Figure 1. The substitution for the meta-variable Φ appearing in the component CHAMs would result in a set

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

in the system CHAM. Notice that the meta-variable Φ is dropped from the system CHAM. A CHAM specification that does not contain meta-variables is said to be *ground*.

The transition rules of the system CHAM is simply the union of the transition rules of the component CHAMs. A certain amount of simplification could be performed on the resulting set of rules.

For example, the specialized communication 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, again, simply formed as a union of the initial molecules of the component CHAMs. 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 is 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 behavioral type checking mechanism that we develop in Section 6. The checking 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 suitable representations of the actual behavior, AC, of a component, and assumed behavior, AS, of the external context. For each component, we derive these two representations from its component CHAM specification.

The model for both representations is a finite, directed, rooted graph, where both nodes and arcs are labeled. Formally,

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

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 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* 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 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* 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.

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 context to perform the reaction by r. If the outgoing arc in G_{ac} is labeled with τ, then the outgoing arc from μ is labeled with α.*
- *if ν has* or *arcs, then μ also has corresponding* or *arcs.*

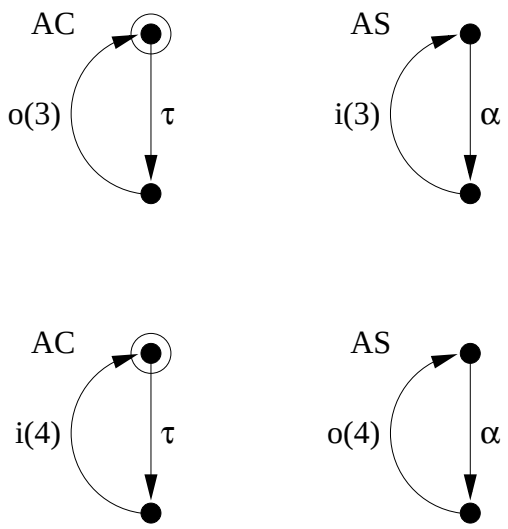

Figure 2: AC and AS Graphs for the Upstream (top) and Downstream (bottom) Filters.

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.

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.

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. This is not enough to conclude that the whole system blocks, but we can iterate the checking phase, reducing the actual behavior of

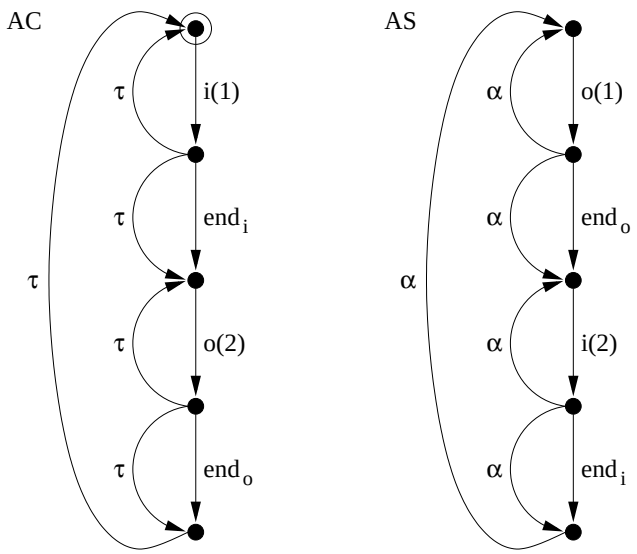

Figure 3: AC and AS Graphs for gzip.

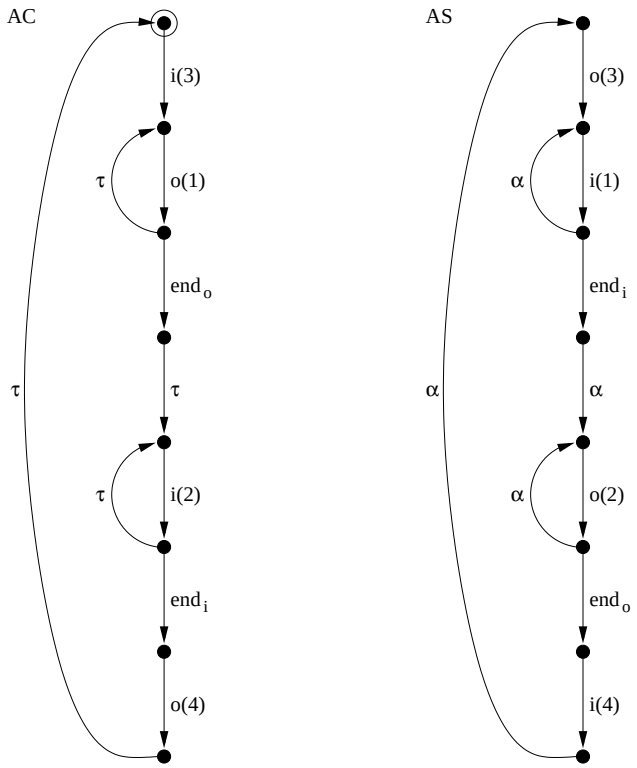

Figure 4: AC and AS Graphs for the Adaptor.

the blocking components. In other words, we can eliminate the part that can block and see how this affects other components.

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 [17].

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 *more* 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 simulate 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 *or* graph and, therefore, there can be a different thread providing the required context.

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 there exist infinite paths from the root node that contain the arc. A recursive arc in an AS graph indicates an assumption that the behavior will 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.

Given this background, we can now define what it means for AS and AC graphs to be in relation.

Definition 3 *(Relation between AS and AC graphs)*
Let G_{ac} be an actual behavior graph, $\nu_1 \dots \nu_n$ be nodes in G_{ac}, G_{as} be an assumption graph, $\mu_1 \dots \mu_m$ be nodes in G_{as}, $\gamma \in \Lambda \setminus \{\tau, \alpha\}$, then two nodes are related, $\nu_i \simeq \mu_j$:

- *if $\mu_j \xrightarrow{\gamma} \mu_{j+1}$, then either ($\nu_i \xrightarrow{\gamma} \nu_{i+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$), or ($\nu_i \xrightarrow{\tau} \nu_{i+1}$ and $\mu_j \simeq \nu_{i+1}$), or ($\nu_i \xrightarrow{\tau} \nu_{i+1}$ and $\nu_{i+1} \in \simeq$), or ν_i has no outgoing arcs;*
  if $\mu_j \xrightarrow{\alpha} \mu_{j+1}$ then $\mu_{j+1} \simeq \nu_i$.
- *if $\nu_i \xrightarrow{\gamma} \nu_{i+1}$, then either ($\mu_j \xrightarrow{\gamma} \mu_{j+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$) or there exists a ν_k such that $R(\nu_k)$ $\nu_k \simeq \mu_j$;*
  if $\nu_i \xrightarrow{\tau} \nu_{i+1}$, then either $\nu_{i+1} \simeq \mu_j$, or ($\mu_j \xrightarrow{\alpha} \mu_{j+1}$ and $\nu_{i+1} \simeq \mu_{j+1}$), or there exists a ν_k such that $R(\nu_k)$ and $\nu_k \simeq \mu_j$;

$G_{ac} \simeq G_{as}$ 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 recursive arc in G_{as}, the source and target nodes of that arc are in relation with source and target nodes of a correspondingly recursive arc in G_{ac} such that the paths in G_{ac} are obtained only from nodes in the relation.

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.*

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 3, 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 4 *(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 association in the substitution σ.

Definition 5 *(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.*
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* $\sigma = \sigma \cup (AS_{C_j}, AC_{C_i})$.
4. *Obtain a new graph* AC'_{C_i} *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 do not address this issue here.

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 puts in relation all the nodes of $AC_{\mathbf{CF}_u}$ with some of the nodes in $AS_{\mathbf{AD}}$. 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 we similarly obtain 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 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. 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 $\mathbf{end}_o$ from the context. Thus, the adaptor will be blocked, not producing an $\mathbf{end}_i$, which in turn will cause **gzip** to block, thus achieving a state of deadlock.

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

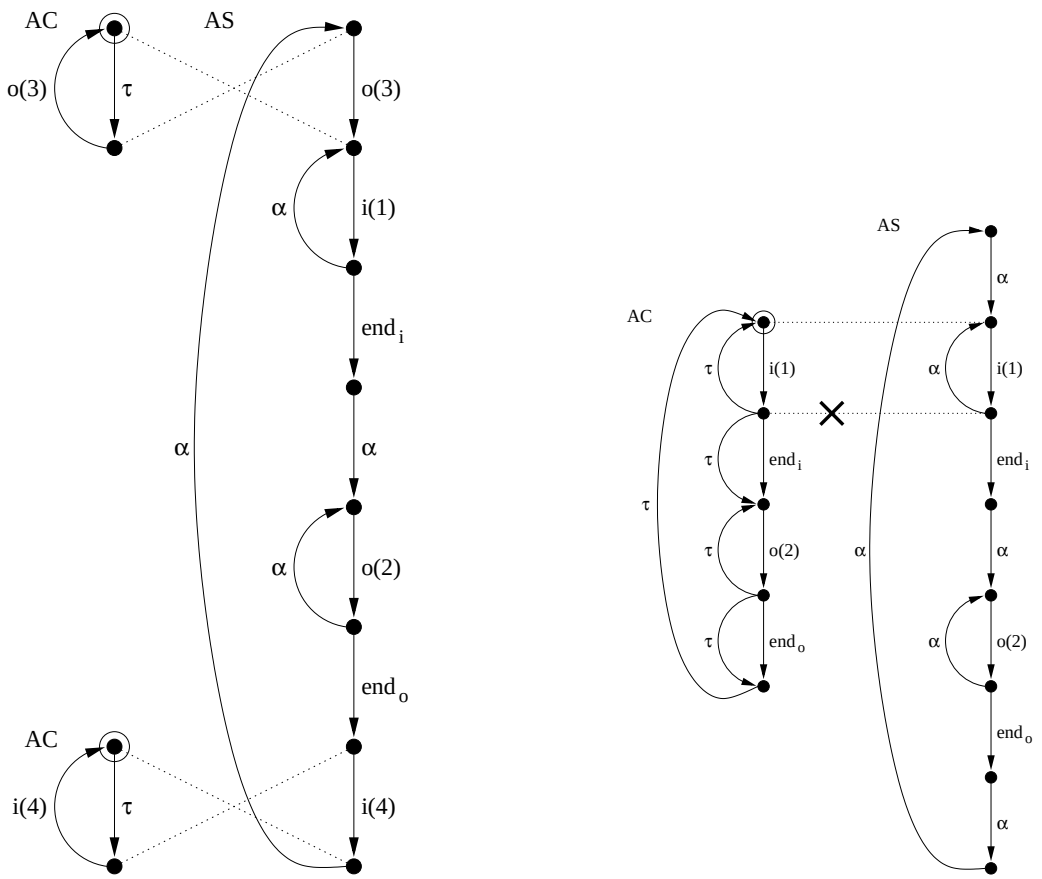

Figure 5: Mismatch in Actual and Assumed Behavior Leading to Deadlock.

Definition 6 *(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 $S_0 \rightarrow S_1 \rightarrow \ldots \rightarrow S_n$ and $S_n \notin F$.

Let us now state the correctness of our algorithm.

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

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. Furthermore, this means that there exists a rule T_j that can be applied to the solution $S = M_i, M_{i+1}, \ldots, M_{k-1}$. 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 $\lambda = M_{i+1}, \ldots, M_{k-1}$ 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, i.e., 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 because it blocks before reaching it.

Now we proceed by induction on the distance of the M_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 the node ν_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, the only possibility is that from ν_x the node ν_j cannot be reached. This is possible only if ν_j 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 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.

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 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** 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.

	Adaptor (AD)
Syntax	$M' ::= P \mid C \mid E \mid M' \diamond M' \mid M' \parallel M'$ $P ::= \mathbf{AD} \mid \Phi$ $C ::= i(N) \mid o(N)$ $N ::= n_1 \mid n_2 \mid n_3 \mid n_4$ $E ::= \mathbf{end}_i \mid \mathbf{end}_o$
Trans. Rules	$T_1 \equiv i(n) \diamond m_1,\ o(n) \diamond m_2 \longrightarrow m_1 \diamond i(n),\ m_2 \diamond o(n)$ $T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$ $T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(n),\ \mathbf{end}_i \diamond m_2 \diamond i(n) \longrightarrow m_1 \diamond o(n) \diamond \mathbf{end}_o,\ m_2 \diamond i(n) \diamond \mathbf{end}_i$ $T'_4 \equiv m_1 \parallel m_2 \parallel \cdots \parallel m_k \longrightarrow m_1, m_2, \ldots, m_k$ $T'_5 \equiv \mathbf{AD} \diamond m \longrightarrow m \diamond \mathbf{AD}$
Init. Mol.	$i(n_1) \diamond o(n_2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \parallel i(n_3) \diamond \mathbf{end}_i \diamond o(n_4) \diamond \mathbf{AD}$
Final Mol.	$\{m_1 \diamond \mathbf{AD}, m_2 \diamond \mathbf{AD}\}$

Table 2: Modified Component CHAM for the Adaptor.

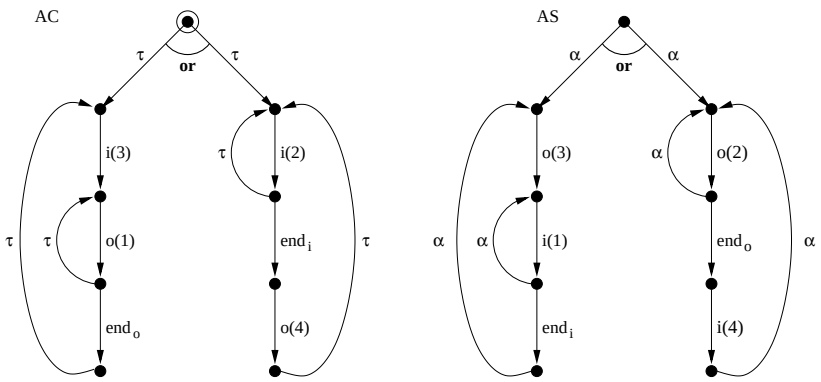

Figure 6: AC and AS Graphs for the Modified Adaptor.

7 Conclusions and Future 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; 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 of 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

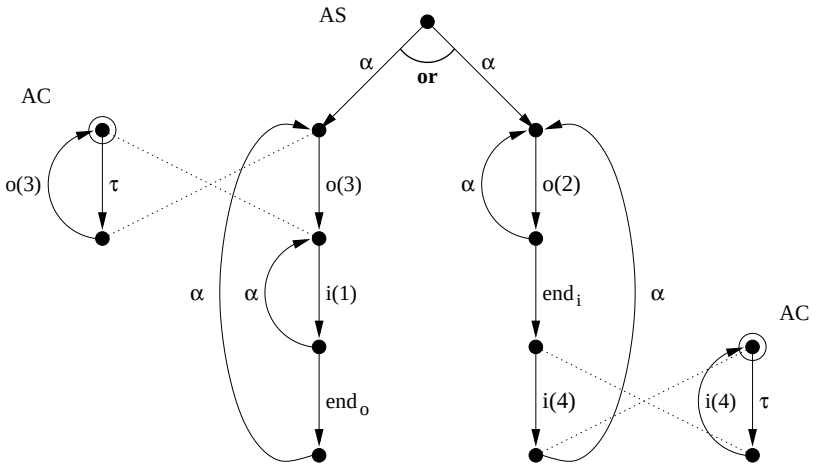

Figure 7: Successful Match of Filter AC Graphs against Adaptor AS Graph.

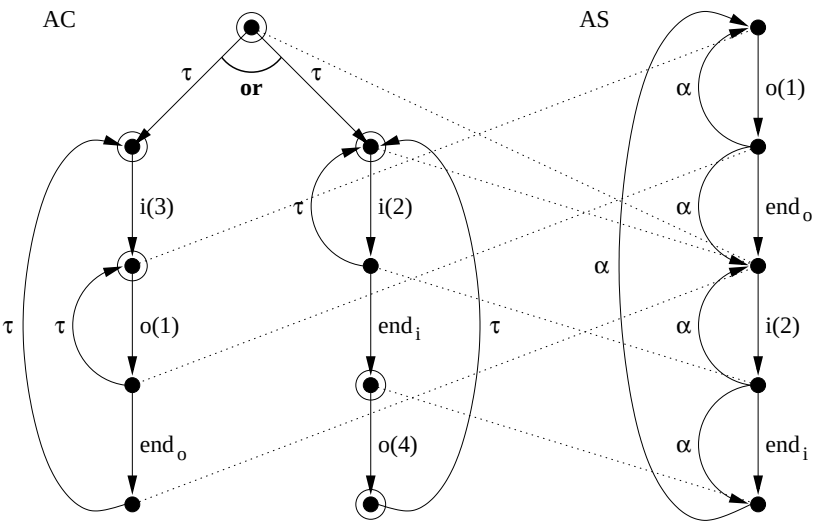

Figure 8: Successful Match of Adaptor AC Graph against gzip AS Graph.

situation. The case study of the Compressing Proxy 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.

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 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

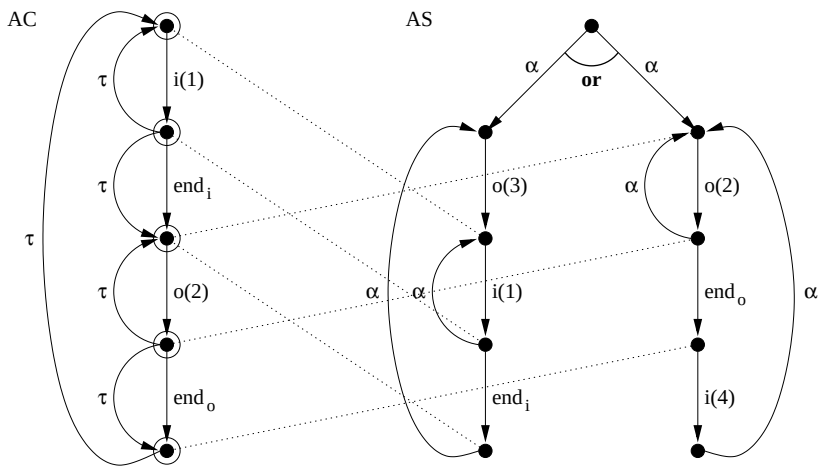

Figure 9: Successful Match of gzip AC Graph against Adaptor AS Graph.

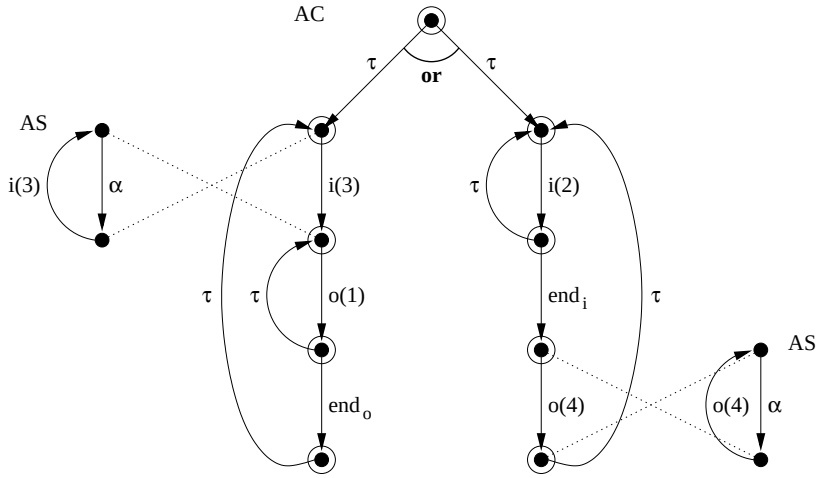

Figure 10: Successful Match of Adaptor AC Graph against Filter AS Graphs.

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 for generalizing the results presented here.

Let us now discuss current limitations of the proposed approach and possible extensions. 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 transition 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 label to be "set of labels". Similarly, the matching algorithm can be extended by

modifying the part of the equivalence that chooses a label from the set and by extending the notion of covered arc.

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 dynamic properties by locally checking component 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 multi-way communications and synchronizations.

Acknowledgments The work of A.L. Wolf was supported in part by the National Science Foundation under grant INT-95-14202 and by the Air Force Material Command, Rome Laboratory, and the Defense Advanced Research Projects Agency under Contract Number F30602-94-C-0253. 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 P. Inverardi was supported in part by a CNR/CONICET bilateral project and by the LOMAPS ESPRIT project.

REFERENCES

[1] 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.

[2] R. Allen and D. Garlan. A Formal Basis for Architectural Connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, July 1997.

[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. *Science of Computer Programming*, 1998. To appear.

[10] S.H. Edwards and B.W. Weide. WISR8: 8th Annual Workshop on Software Reuse Summary and Working Group Reports. *SIGSOFT Software Engineering Notes*, 22(5):17–32, September 1997.

[11] D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is So Hard. *IEEE Software*, 12(6):17–26, November 1995.

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

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

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] 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.

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

[22] 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.

[23] A.M. Zaremski and J.M. Wing. Specification Matching of Software Components. *ACM Transactions on Software Engineering and Methodology*, 6(4):333–369, October 1997.