Uncovering Architectural Mismatch in Component Behavior

Daniele Compare and Paola Inverardi

Dipartimento di Matematica
Universitá di L'Aquila
I-67010 L'Aquila, Italy

({compare,inverard}@univaq.it)

Alexander L. Wolf

Department of Computer Science
University of Colorado
Boulder, CO 80309 USA

(alw@cs.colorado.edu)

University of Colorado
Department of Computer Science
Technical Report CU-CS-828-97 February 1997

A version of this report to appear in
Science of Computer Programming

ABSTRACT

When constructing software systems from existing components, the engineer is faced with the problem of potential conflicts in the interactions among the components. Of particular difficulty is guaranteeing compatibility in the dynamic interaction behavior. Using an architectural description of the system and its intended components, the engineer can reason about the interactions early and at a high level of abstraction. In this paper we give a case study of the Compressing Proxy system, which was first investigated by Garlan, Kindred, and Wing. We present architectural specifications and analyses of two versions of the system. One version is a seemingly obvious melding of the components. The other is a solution to deadlock problems uncovered by formal analyses of the first version. We use the Chemical Abstract Machine as an example of an architectural description formalism that can help uncover architectural mismatches in the behavior of components.

1 Introduction

An architectural description makes the analysis, design, and construction of a complex system intellectually tractable by characterizing the system at a high level of abstraction. Using an architectural description, the engineer can reason about the satisfaction of system requirements in terms of the assignment of functionality to design elements and the interaction of those design elements at their interfaces. This is particularly useful for one emerging method of design, namely that of assembling a software system from existing components.

Components naturally embody assumptions about the structure and behavior of the larger contexts in which they operate. When constructing software systems from existing components, the engineer is therefore faced with the problem of uncovering and avoiding *architectural mismatch*. According to Garlan, Allen, and Ockerbloom [11]:

> *"Architectural mismatch stems from mismatched assumptions a reusable part makes about the structure of the system it is to be part of. These assumptions often conflict with the assumptions of other parts and are almost always implicit, making them extremely difficult to analyze before building the system."*

An important class of mismatches can be understood to arise from conflicts at two levels of interaction. One is the compatibility of the data exchanged among the components, and is usually captured quite well by the type information present in the interfaces and the static analyses based on the type information. The other, more difficult compatibility is the dynamic interaction and communication behavior of the components. Mismatches often arise at this level because engineers lack the understanding about individual component behaviors that contribute to the correct (or incorrect) global behavior of the system.

A number of researchers have been experimenting with a variety of techniques for describing and analyzing systems at the architectural level of design [1, 2, 14, 18]. Each of the techniques is based on a different underlying formalism. For example, Abowd et al. use Z [23] for specifying architectural styles, while the Wright architectural description language [2] is based on CSP [13]. The technique developed by Inverardi and Wolf [14] is based on the CHAM (CHemical Abstract Machine) formalism [5]. CHAM is an operational formalism that leads to a description of an architecture as a set of components (the "molecules") whose states and interactions are governed by transformation rules (the "reactions").

In this paper we demonstrate how designers can use formal architectural specifications and analyses to help uncover architectural mismatch in component behavior. To illustrate the benefits of this approach, we employ the techniques that we developed for the CHAM formalism. Our earlier work [14] exploited the algebraic and term-rewriting flavor of the CHAM formalism to introduce the basic algebraic analysis approach to the architectural level of design. Here we extend that work by giving improved structure to the transformation rules and by showing a second kind of analysis based on transition-system generation in the style of Milner [20].

We use as our example a case study of the Compressing Proxy system introduced by Garlan, Kindred, and Wing [12]. This example was later used by Inverardi, Wolf, and Yankelevich [16] to demonstrate an algorithm for checking assumptions in component behaviors. The Compressing Proxy is designed as a combination of two pre-existing component systems, each individually designed and separately useful. Due to an architectural mismatch problem, it took the designers of the Compressing Proxy two attempts to properly develop the system. In their first attempt, the designers used a specially built adaptor component to account for an obvious mismatch between the function-call-based stream interface of one component and the standard UNIX pipe interface of the other component. However, this first version of the system exhibited deadlock problems arising from a behavioral mismatch among the components. After analysis revealed the source of the deadlock, the adaptor was modified and the second attempt at a solution worked.

As shown below, the Compressing Proxy case study clearly illustrates the point that, when assembling existing components to form a system, there is a need for precise specifications of the behavior of the components at the architectural level. Analysis of the specifications can then provide early feedback about the feasibility of the assembly. Moreover, the analysis can indicate where adjustments to the components and their interconnections might be made.

Of course, in our study of the Compressing Proxy, we had *a priori* knowledge of where the mismatch arose. Nevertheless, it should be clear from the discussion below that the specifications one would create for the components, as well as the analyses that one would apply to the specifications to discover the mismatch, reasonably could be expected to follow those illustrated here.

In the next section we introduce the Compressing Proxy problem, giving an intuitive description of the challenge it presents. In Section 3 we review related work in software architecture specification and analysis. Following that, we review the essentials of the CHAM formalism that are required for this paper. We then present the CHAM specifications for the two versions of the Compressing Proxy architecture, demonstrating the deadlock that arises in the first. The two different styles of analysis supported by the CHAM formalism and applied to this problem, algebraic analysis and transition-system generation, are discussed and illustrated in Section 6. We conclude in Section 7 by considering how the two kinds of analysis techniques might be employed in concert.

2 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.

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.

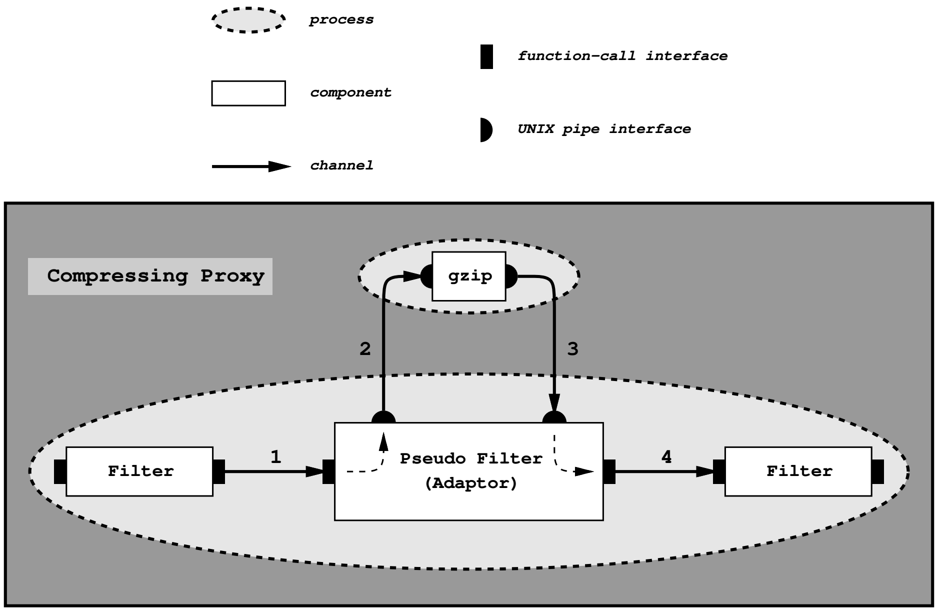

Figure 1: The Compressing Proxy.

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.

An adaptor constructed in this way clearly solves the first level of interface mismatch. However, a deeper level of mismatch can occur without a proper understanding of the behaviors of the components. Consider the following straightforward method of structuring the adaptor. The adaptor simply passes data on to **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. At a component level, this behavior makes sense. But at a global system level we can experience deadlock. 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. Below we show how analyses performed on CHAM descriptions of the component behaviors can reveal such problems.

3 Related Work

In this section we review several formal specification techniques appropriate to the architectural level of design, and describe some analyses associated with the specification techniques.

The specification of an architecture involves a delineation of the components and the ways in which those components are connected. Perry and Wolf [21] give a model for architectural specification that distinguishes three different 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.

The Wright specification language [2] provides a means to formalize connecting elements. The idea is that connecting elements should be treated as "first class", such that they have their own specification independent components. This should allow components to be more easily connected and reconnected in a variety of ways, as long as those connections satisfy the specifications. Moreover, it should be possible to demonstrate the correctness of components and connectors somewhat more independently so that the verification task is reduced in complexity and cost.

Wright employs a subset of CSP [13] to define communication protocols among components. In particular, CSP is used for the specification of component interface behavior, and for the specification of roles and glue in connectors.

Formal analyses of Wright specifications concentrate on two properties. The first is the standard property of deadlock freedom, which has been extensively studied in the context of CSP. The second, and more interesting, is the port-role compatibility problem. The simple form of this check is to guarantee that ports and roles realize identical protocols. But we could allow for more flexible combinations of components if we could guarantee the weaker condition that the "promised" behavior of a role is "respected" by the corresponding port. This can be cast in CSP terms, where the problem is interpreted as *refinement* of protocols. Once cast this way, there are commercial tools available for performing the analyses. An example is FDR [9], which is an application of model-checking techniques.

A somewhat different perspective on architectural specification and analysis is provided by event languages. Here the key property is the identification and ordering of events, which are discrete markings of computational progress. An event is a very flexible, abstract notion that allows the architect to describe the system at an arbitrary level of detail, depending on the particular definition of events of interest.

Rapide [17, 18] is an executable, event-based specification language targeted for the architectures of distributed systems. The idea behind Rapide is that simulation is a key capability for checking the consistency of interfaces and connections, for understanding the behavior of the system as a whole, and for verifying that the system's communication adheres to the desired interaction structure of the architecture. A Rapide specification can be thought of as a very high-level prototype.

A Rapide architecture specification consists of a description of a set of *interfaces*, *connections*, and *constraints*. Interfaces specify the components of the system in terms of the resources that they provide and require, and specify the behaviors of the components in terms of the actions that they exhibit and to which they respond. Connections define the communications among the components in terms of the interfaces of those components. Finally, constraints restrict the behaviors of the interfaces and connections.

Behavior specifications for components and interfaces are given by *event patterns* that describe the partially ordered set of events, called a *poset*, that can be generated by the actions of the components and interfaces. An event can be considered an instantiation of an action at some particular point in time. An execution of a Rapide specification (i.e., a simulation of the architecture) results in a poset that represents some particular interaction among the components. The poset indicates the dependencies and independencies among events exhibited by the system for that particular execution.

Architecture analysis using Rapide amounts to checking for proper orderings of events within the context of constraints on those orderings. It also involves checking for causality among events. Particular event orderings are generated through simulation and examined for these properties. A somewhat different kind of analysis also possible with Rapide is to guarantee that the communication structure of the architecture is strictly preserved as specified. This is particularly useful when comparing an implementation of the system to its architectural specification and leads to a form of acceptance testing based on architecture-level concerns.

Le Métayer [19] has developed an approach to software architecture specification in which software architectures are modeled as graphs and software architecture styles are modeled as graph grammars. The approach is based on drawing a clear distinction between the specification of a single component's behavior and the specification of the overall structure of the system. A graph represents an architecture by interpreting nodes as components, whose behavior is separately specified in a conventional specification language, and interpreting arcs as the communication links between components. Architectures that exhibit the same graph structure are considered to be elements of the same architectural style. Thus, an architectural style can be naturally expressed as a (context-free) graph grammar.

The evolution of a system, in terms of its topological structure, is governed by a "coordinator" component whose behavior is specified as conditional graph rewriting rules. The main contribution with respect to the analysis of architectures is that it is possible to statically check if a given coordinator changes the structure of the graph specification according to the given style—that is, according to the given graph grammar specifying the style. Of course, the use of a dedicated component, the coordinator, to manage the dynamic structure of a system imposes a specific view of how a software architecture must be constructed, which limits the generality of the approach.

4 Background on the CHAM Formalism

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 older, more familiar computational models, such as CSP [13] and the CCS process calculus [20]. Boudol [6] points out that the CHAM formalism has also been demonstrated as a modeling tool for concurrent-language definition and implementation.

Inverardi and Wolf [14] developed a framework for architectural specification and analysis based on the CHAM formalism. Their goal is to apply the power of the CHAM, not to its original purpose of capturing computational models and defining programming languages, but rather to the design phase of specific software systems. Below, we briefly summarize the concepts in the CHAM formalism relevant to this paper. We also report on the use of those concepts in the domain of software architecture.

4.1 The Chemical Abstract Machine

The CHAM formalism is operational in nature. It is has a notion of state and a way to specify the possible evolutions from one state to another. The set of all possible evolutions that a CHAM can perform, starting from a given state, completely describe its behavior.

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*.

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.

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. Informally, the general rules define the computational mechanisms any CHAM is based on, while specific rules define the particular behavior of a given CHAM specification.

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 relevant 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.

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

We use an upper case M to represent a generic pattern, while a lower case m represents a suitable instance of the pattern.

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*.

As discussed in Section 3, several formalisms have been proposed to model software architectures. The CHAM formalism represents a minimalist and flexible approach, allowing for specifications that immediately reflect the dynamic behavior of the specified system. The algebraic structure of the molecules allows one to also model the static structure of the system, thus obtaining a comprehensive framework in which both static and dynamic features of the software architecture can be expressed. Of course, the minimalism of a CHAM can be a drawback when the system specification becomes too detailed.

4.2 Specifying Software Architectures

The CHAM specification of a software architecture consists of three parts [14]:

1. a description of the syntax by which components of the architecture (i.e., the molecules) can be represented;
2. a solution representing the initial state of the architecture; 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 [21], we distinguish three classes of components: data elements, processing elements, and 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 configuration of the system. Transformation rules applied to the initial solution define how the system dynamically evolves from its initial configuration.

In our use of the CHAM, 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.

With this necessarily brief introduction to the CHAM formalism and its use in the domain of software architecture, we can now illustrate the utility of our approach to uncovering architectural mismatch in dynamic behavior.

5 Specifications of the Compressing Proxy

As described in Section 2, the Compressing Proxy architecture was developed in two versions. We refer to them as the Blocking and the Non-blocking Compressing Proxy, respectively. In this section we give their CHAM specifications to serve as a basis for the analyses discussed in Section 6. The specifications are purposefully kept simple and focused to highlight the important aspects of our approach.

Note that in keeping the example simple, we are specifying the system at a rather high level. Nevertheless, it is already possible at this level to shed light on potential architectural problems of behavioral mismatch. If required, however, it is appropriate within the CHAM model to incorporate additional detail into those descriptions.

In our formulation of the Compressing Proxy architecture we refer to the depiction given in Figure 1. The filter to the left of the adaptor is referred to as the "upstream" filter, while the filter to the right is referred to as the "downstream" filter. Communication along channels 1 and 2 represents the passing of data from the upstream filter through the adaptor to **gzip** for compressing. The reverse communication along channels 3 and 4 represents the passing of compressed data back through the adaptor and onto the downstream filter. Notice that the data themselves are not represented, only the communication channels and the protocols governing them.

Below, we first present the specification of the Blocking Compressing Proxy and then show a series of applications of its transformation rules to illustrate the system's dynamic behavior. As a demonstration of the mismatch problem, this particular behavior results in deadlock. We then present the specification of the Non-blocking Compressing Proxy in terms of its differences from the Blocking Compressing Proxy. These differences precisely embody the enhancements to the adaptor module that eliminate the deadlock problem. The proof of this is supported by the analyses presented in Section 6.

5.1 The Blocking Compressing Proxy

The first step in specifying the Blocking Compressing Proxy architecture is to define the syntax Σ_b of its molecules M.

$$
\begin{aligned}
M &::= P \mid C \mid E \mid M \diamond M \\
P &::= F \mid \mathbf{AD} \mid \mathbf{GZ} \\
C &::= i(N) \mid o(N) \\
N &::= 1 \mid 2 \mid 3 \mid 4 \\
E &::= \mathbf{end}_i \mid \mathbf{end}_o \\
F &::= \mathbf{CF}_u \mid \mathbf{CF}_d
\end{aligned}
$$

The syntax consists of the set P representing the three kinds of processing elements and of an infix operator "$\diamond$" used to express the status of a processing element. The connecting elements for the architecture are given by a second set C consisting of two operations, i (for input) and o (for output), that act on the elements of a third set N. This third set is used to define the topology of the system in terms of the communication channels connecting the components, and correspond to the numbers given in Figure 1. A fourth set E introduces the control signals used in the communication between **gzip** and the adaptor. The set F contains the representation of the "upstream" and the "downstream" CERN filters between which is placed the adaptor for **gzip**. Notice that at this level of abstraction we are not concerned with the actual data transferred between the components, simply the protocol by which they communicate. We take as the set of syntactic elements the initial algebra in the class of all the Σ_b algebras.

Let us provide some intuition behind this syntax. We use the two operations i and o to represent primitive communications over the channels between components, where i is for input and o is for output. 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". Finally, the infix operator "$\diamond$" is used to express the status of a processing element with respect to its input/output behavior. In particular, the status is understood by "reading" the molecule from left to right. Consider, for example, the **AD** molecule $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$. This is interpreted to mean that **AD** offers output on channel 2 and is then prepared to end output. It is further interpreted that **AD** has previously accepted input on channel 1. 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 next step in specifying the Blocking Compressing Proxy architecture is to define an initial solution S_0. This solution is a subset of all possible molecules that can be constructed under Σ_b and corresponds to the initial configuration of a system conforming to the architecture.

$$
\begin{aligned}
S_0 = \; & \mathbf{CF}_u \diamond o(1), \\
& \mathbf{CF}_d \diamond i(4), \\
& i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}, \\
& i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}
\end{aligned}
$$

The solution establishes the basic connectivity of the components, which corresponds to the channel numbers shown in Figure 1. It establishes that the upstream filter will offer data along channel 1 (to **AD**) and the downstream filter will accept data along channel 4 (from **AD**), although both are initially in a quiescent state, since the left-most position of each molecule is not a communication operation. **GZ** and **AD** are somewhat more complicated. **GZ** is initially in the state of accepting data along channel 2 (from **AD**). It can then end its input and enter a state of offering data along channel 3 (to **AD**), after which it can end that output. **AD** is initially in the state of accepting data along channel 1 (from an upstream filter) and must wait until it has stopped accepting the data before it can offer data on channel 2 (to **GZ**). It can then end its output. The full meaning of the initial state becomes apparent when combined with the transformation rules.

There are eight transformation rules that define the complete behavior of the Blocking Compressing Proxy at this level of architectural modeling.

$$T_1 \equiv i(x) \diamond m_1,\ o(x) \diamond m_2 \longrightarrow m_1 \diamond i(x),\ m_2 \diamond o(x)$$

$$T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$$
$$T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(x),\ \mathbf{end}_i \diamond m_2 \diamond i(x) \longrightarrow m_1 \diamond o(x) \diamond \mathbf{end}_o,\ m_2 \diamond i(x) \diamond \mathbf{end}_i$$
$$T_4 \equiv \mathbf{end}_i \diamond m_1 \diamond \mathbf{GZ} \diamond m_2 \longrightarrow m_1 \diamond \mathbf{GZ} \diamond m_2 \diamond \mathbf{end}_i$$
$$T_5 \equiv \mathbf{end}_o \diamond \mathbf{GZ} \diamond m \longrightarrow \mathbf{GZ} \diamond m \diamond \mathbf{end}_o$$

$$T_6 \equiv \mathbf{GZ} \diamond m \longrightarrow m \diamond \mathbf{GZ}$$

$$T_7 \equiv f \diamond c \longrightarrow c \diamond f$$

$$T_8 \equiv \mathbf{AD} \diamond i(1) \diamond m \longrightarrow i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$$
$$T_9 \equiv \mathbf{AD} \diamond i(3) \diamond m \longrightarrow i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$$

where $m, m_1, m_2 \in M$, $x \in N$, $c \in C$, $e \in E$, and $f \in F$. Rule T_1 is a general inter-element communication rule, rules T_2 through T_5 capture the communication protocol between **gzip** and its adaptor, rule T_6 enables the iteration of **GZ**, and rule T_7 describes the activation of the upstream and downstream filters. Rules T_8 and T_9 are the critical rules for the mismatch problem; these rules describe the behavior of the adaptor and are replaced with two other rules in the corrected version of the architecture.

Let us provide more explanation for each rule.

- T_1 generically describes pairwise input/output communication between processing elements. In particular, communication occurs if there is a processing element m_1 that accepts input from a channel offered as output by some other processing element m_2 on the same channel. Recall that the ability of a processing element to communicate is syntactically indicated by the appearance of a communication operation in the left-most position of the molecule. Completion of the communication—that is, the result of the transformation—is indicated by a rewriting of the molecule such that the communication operation is moved to the right-most position of the molecule.

- T_2 allows either **AD** or **GZ** to iterate its communication behavior.

- T_3 terminates communication through **AD** and **GZ** in the successful case—that is, the components "agree" to terminate the data transfer between them.

- T_4 allows **GZ** to independently terminate its input. This situation can arise, for example, when its internal buffer is full.

- T_5 allows **GZ** to independently terminate its output. This situation can arise, for example, when the internal buffer has emptied.

- T_6 reactivates **GZ** to allow new compressions.

- T_7 also reactivates components, in this case those representing upstream and downstream CERN filters.

- T_8 changes the structure of **AD** with respect to the initial solution to indicate its readiness to receive compressed data. As such, **AD** can no longer receive data on channel 1 from an upstream filter and pass them along to **GZ** on channel 2 for compression.

- Finally, T_9 permits **AD** to receive new data for compression, restoring the molecule to its original status.

Notice that most of the rules apply to individual components, and thus are independent of the global context. For example, the "decision" by **gzip** to end its input or output through rule T_4 or T_5 is local to the component. Only rules T_1 and T_3 involve coordination among multiple components.

To summarize the formulation, let us take the perspective of each component in the system. A filter is modeled as a component that iteratively offers data for output (if it is an upstream filter) or accepts data for input (if it is a downstream filter). Rule T_7 models this iterative behavior. The adaptor **AD** is modeled as a bi-modal element. In its initial mode, **AD** iteratively accepts data from an upstream filter and passes data on to **GZ**. Rule T_2 models this iterative behavior. Rule T_8 models the change from **AD**'s initial mode to one of iteratively accepting data for input from **GZ** and offering data for output to a downstream filter. Rule T_9, on the other hand, models the reverse change in mode. **GZ** begins its behavior by iteratively accepting data from the adaptor. Rule T_2 models this iterative behavior. Rule T_4 models the decision by **GZ** to end this iterative input behavior and begin to iteratively offer data for output to **AD**. The iterative output behavior of **GZ** is modeled by rule T_2, while the decision by **GZ** to end its iterative output behavior is modeled by rule T_5. **GZ** and **AD** can coordinate the ending of their transfer of data. This is modeled by rule T_3, which applies generically to both the case of **AD**→**GZ** data transfer as well as the case of **GZ**→**AD** data transfer. The overall iterative behavior of **GZ** is modeled by rule T_6. Finally, the actual input/output behavior of all components is modeled using rule T_1.

We have thus defined an abstract machine that evolves dynamically from one admissible state to another, starting from the initial solution S_0 and using the transformation rules T_1 through T_9 to model the possible behaviors of the system. Naturally, these behaviors involve only a subset of all possible molecules that can be constructed under Σ_b.

One thing to observe about our formulation is that there is no way for **gzip** to operate on an empty stream, although it is possible for the actual tool to do so. This is an example of the fuzzy boundary between architectural abstraction and what might be considered an implementation detail. To model the ability of **gzip** to operate on an empty stream requires the addition of a simple rule to account for this case. We did not include such a rule here, however, because it does not materially affect the analyses we are demonstrating.

Another thing to observe about our formulation is that the data themselves are not modeled. We simply indicate the behavior that leads to data transfer, without specifying either the form or granularity of the data.

Finally, a possible source of confusion in this formulation arises from the generic manner in which rules T_1 through T_3, as well as rule T_7, are defined. For example, rule T_1 applies to any pair of communicants, while rules T_2 and T_3 apply to both **AD** and **GZ**, but in different situations and in different roles. This is a stylistic issue that is not in any way dictated by our approach. We chose in this example to develop a compact set of transformation rules, perhaps at the cost of some degree of readability. An alternative would have been to instantiate the generic rules for each of their possible specific uses. While it would then have been clear as to which rule applied to which component, the number of rules would have increased. We regard this ability to flexibly tradeoff succinctness against specificity as one of the strengths of our approach.

We now trace through just a few applications of the transformation rules to illustrate how our formulation captures the essence of the architecture. This particular trace happens to be one that leads to the deadlock resulting from the architectural mismatch.

First, data to be compressed must be available, and therefore the solution must be "heated" by rule T_7 acting on the molecule $\mathbf{CF}_u \diamond o(1)$.

$$\begin{aligned} & S_0 \xrightarrow{T_7} S_1, \text{ where} \\ & S_1 = o(1) \diamond \mathbf{CF}_u, \\ & \quad\;\; \mathbf{CF}_d \diamond i(4), \\ & \quad\;\; i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}, \\ & \quad\;\; i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \end{aligned}$$

Now a reaction can occur within the subsolution consisting of molecules $o(1) \diamond \mathbf{CF}_u$ and $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$. This reaction is governed by T_1 and represents the initial transfer of data from $\mathbf{CF}_u$ to $\mathbf{AD}$.

$$\begin{aligned} & S_1 \xrightarrow{T_1} S_2, \text{ where} \\ & S_2 = \mathbf{CF}_u \diamond o(1), \\ & \quad\;\; \mathbf{CF}_d \diamond i(4), \\ & \quad\;\; o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1), \\ & \quad\;\; i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \end{aligned}$$

The data transfer has occurred through a single reaction, and $\mathbf{CF}_u$ is now in a state in which T_7 is required to activate it once again for a further data transfer. Although T_7 can be applied, for brevity we do not consider this possibility further in the discussion, since our intention here is only to illustrate the behavior of the system.

At this point, reaction T_1 can occur again, modeling the passing of data from $\mathbf{AD}$ to $\mathbf{GZ}$ for compression. T_1, in this case, acts upon the subsolution consisting of molecules $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$.

$$\begin{aligned} & S_2 \xrightarrow{T_1} S_3, \text{ where} \\ & S_3 = \mathbf{CF}_u \diamond o(1), \\ & \quad\;\; \mathbf{CF}_d \diamond i(4), \\ & \quad\;\; \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2), \\ & \quad\;\; \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \end{aligned}$$

From this state, any one of the three reactions T_2, T_3, or T_4 can occur nondeterministically. (Recall that we are not further considering applications of T_7, although it too can be applied in this situation.)

A reaction involving T_2 would model the availability of new data. In fact, this reaction models the possible iteration of communication from $\mathbf{AD}$ to $\mathbf{GZ}$—that is, the cycle $T_2, T_2, T_1, \ldots, T_2, T_2$, in which every cycle results in a new amount of data to be compressed. It is evident that a possibly infinite amount of effort could be spent unproductively looping between the rules T_2 and T_1. This behavior amounts to modeling that the internal buffers of the adaptor and **gzip** have an infinite capacity. Although this is clearly unrealistic, it has a minimal impact on the modeling of the architecture. However, this can be easily solved by modifying the specification such that the $T_2, T_2, T_1, \ldots T_2, T_2$ cycle is suitably constrained to, for example, simulate some bounded buffer [7, 8]. In fact, it is important to introduce a constraint such as this only if it is necessary in the description of the system, which for our purposes here it is not. Therefore, we do not consider this situation further. Instead, we only consider situations in which the buffer is implicitly treated as finite.

We postpone the application of T_4 and now consider the application of T_3, which represents the situation in which **AD** terminates its output before **GZ** has terminated its input.

$$\mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2),\ \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \xrightarrow{T_3}$$
$$\mathbf{AD} \diamond i(1) \diamond o(2) \diamond \mathbf{end}_o,\ o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$$

Notice that the molecule $\mathbf{AD} \diamond i(1) \diamond o(2) \diamond \mathbf{end}_o$ represents the fact that the adaptor has completed the first part of its processing. **GZ**, on the other hand, is in the state of offering output on channel 3. To this end, the molecule $\mathbf{AD} \diamond i(1) \diamond o(2) \diamond \mathbf{end}_o$ has to react by using T_8. We now have a solution S_4.

$$\begin{aligned} S_3 &\xrightarrow{T_3,T_8} S_4, \text{ where} \\ S_4 &= \mathbf{CF}_u \diamond o(1), \\ &\quad \mathbf{CF}_d \diamond i(4), \\ &\quad i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}, \\ &\quad o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \end{aligned}$$

This reflects the changed state of **AD**, which now is ready to receive compressed data back from **GZ**, since **AD** has terminated its (blocking) writes.

At this point we can have a reaction between **AD** and **GZ**.

$$\begin{aligned} S_4 &\xrightarrow{T_1} S_5, \text{ where} \\ S_5 &= \mathbf{CF}_u \diamond o(1), \\ &\quad \mathbf{CF}_d \diamond i(4), \\ &\quad \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD} \diamond i(3), \\ &\quad \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3) \end{aligned}$$

Again, T_2, T_3, or T_4 can occur nondeterministically, and we consider the reaction performed by T_3.

$$\mathbf{end}_i \diamond o(4) \diamond \mathbf{AD} \diamond i(3),\ \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3) \xrightarrow{T_3}$$
$$o(4) \diamond \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i,\ \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o$$

This reflects the fact that **GZ** has terminated its output and becomes idle, while **AD** is ready to output the compressed data to $\mathbf{CF}_d$. Molecule $\mathbf{CF}_d$ can be activated through T_7, and then T_1 can occur.

$$i(4) \diamond \mathbf{CF}_d,\ o(4) \diamond \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i \xrightarrow{T_1}$$
$$\mathbf{CF}_d \diamond i(4),\ \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i \diamond o(4)$$

With this reaction, $\mathbf{CF}_d$ terminates its wait and in fact accepts all the compressed data from **AD**. It is then ready to take on new data.

By allowing T_6 and T_9 to react, we reach a state equal to the initial solution S_0, from which other reactions can start. In practice, this corresponds to the iterative behavior of the Compressing Proxy.

Let us now return to the state of the system represented by solution S_3 and consider the application of T_4 instead of T_3.

$$\begin{aligned}
S_3 &\xrightarrow{T_4} S_4', \text{ where} \\
S_4' &= \mathbf{CF}_u \diamond o(1), \\
&\quad \mathbf{CF}_d \diamond i(4), \\
&\quad \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2), \\
&\quad o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i
\end{aligned}$$

Notice that this models the situation in which **gzip** must terminate its input because its buffer is full. It needs to write the compressed data, but the adaptor has not yet terminated its output of non-compressed data. For this reason **gzip** blocks waiting for the adaptor to read. However, the adaptor can only try to send more output to **gzip**. In fact, now the only reaction that can occur for **AD** is within the subsolution consisting of the molecule $\mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2)$, which is governed by T_2.

$$\begin{aligned}
S_4' &\xrightarrow{T_2} S_5', \text{ where} \\
S_5' &= \mathbf{CF}_u \diamond o(1), \\
&\quad \mathbf{CF}_d \diamond i(4), \\
&\quad o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1), \\
&\quad o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i
\end{aligned}$$

It is easy to see that no further useful transformation rules can be applied to this solution. Thus, the system has deadlocked because there is no possible correct evolution from S_5'.

A symmetrical deadlock occurs if we consider the application of T_5 to solution S_5. Here it is reasonable to imagine that **gzip**'s internal buffer has emptied. For brevity, we do not show this situation.

5.2 The Non-blocking Compressing Proxy

Let us turn to the second version of the architecture and see what changes are necessary in the CHAM model we developed for the first. The primary difference that we must account for in the second version is that the adaptor can use non-blocking writes when sending data to **gzip**. Therefore, any time one of the writes would have blocked, the adaptor should now be able to read any available data from **gzip** using non-blocking reads. In addition to avoiding deadlock, this approach introduces a certain degree of incremental processing by allowing the Compressing Proxy to start sending out compressed data before all the incoming data have arrived.

We give the specification of the Non-blocking Compressing Proxy in terms of its differences with the Blocking Compressions Proxy. First, we must introduce new elements to represent the new behavior of the adaptor. In particular, we enrich the structure of the molecules by introducing an infix operator "$\|$" to syntactically represent a molecule that can be broken down into parallel subcomponents, thus allowing multiple reactions to occur simultaneously. In more familiar terms, "$\|$" can be interpreted as a parallel operator.

This change requires a new syntax of molecules for the architecture. We can formulate this syntax by a simple modification to Σ_b. Let Σ_n be the syntax that we obtain by replacing the highest-level molecule syntax generator M by M', which is defined as follows.

$$M' ::= P \mid C \mid E \mid M' \diamond M' \mid M' \parallel M'$$

Next, we need to alter the solution that represents the initial configuration. S'_0 is obtained from S_0 by simply replacing its **AD** molecule with the following **AD** molecule.

$$i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \parallel i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$$

This new molecule in the initial solution represents the parallel communications that **AD** can perform. In this way, **AD** will never block while performing input or output.

To complete the specification, we need to replace rules T_8 and T_9 with rules appropriate for the Non-blocking Compressing Proxy.

$$\begin{aligned} T'_8 &\equiv m_1 \parallel m_2 \parallel \cdots \parallel m_n \longrightarrow m_1, m_2, \ldots, m_n \\ T'_9 &\equiv \mathbf{AD} \diamond m \longrightarrow m \diamond \mathbf{AD} \end{aligned}$$

where $m, m_1, \ldots, m_n \in M$. Note that rules T_1 through T_7 defined for the Blocking Compressing Proxy also hold for the Non-blocking Compressing Proxy.

T'_8 is a rule that breaks apart a complex molecule into its (parallel) components, which can then participate in (parallel) reactions. T'_9 reactivates the idle adaptor, encompassing rules T_8 and T_9 from the Blocking Compressing Proxy.

The introduction of the parallel operator in Σ_n is for notational convenience and readability. As evident from rule T'_8, its semantics is simply that of reaching a solution in which the elements of the parallel molecule are placed into the solution by themselves. The same effect could thus be achieved by directly placing the elements into the initial solution, avoiding the need for rule T'_8. On the other hand, use of the parallel operator allows us to express the close relationship of the concurrent threads of the component. The fact that the CHAM itself is an inherently parallel model leads to the interpretation of that operator in more basic CHAM terms. The parallel operator also allows for a more uniform treatment of the adaptor reactivation rule, which now can be simply expressed by rule T'_9.

There is an important thing to notice about this specification, particularly in regard to the interaction of the processing. We assume that an element of F always performs a single input, followed by a single output, followed by another single input, and so on. In essence, we are modeling the input/output behavior of these processing elements as $[IO]^*$. The adaptor performs a single input or a single output during the communication with an element of F, so that we can also model its input/output behavior as $[IO]^*$. But it performs a sequence of outputs followed by a sequence of inputs when it communicates with **GZ**, so that its behavior in this case is modeled by $[O^+I^+]^*$. Finally, the input/output behavior of **GZ** is also modeled by $[I^+O^+]^*$, because it can make a sequence of inputs followed by a sequence of outputs. (Recall the role of $\mathbf{end}_i$ and $\mathbf{end}_o$ in the communication.)

Let us simply trace how the new architecture solves the deadlocks possible in the previous version. Consider the following intermediate solution.

$$\begin{aligned} S_i = {} & \mathbf{CF}_u \diamond o(1), \\ & \mathbf{CF}_d \diamond i(4), \\ & \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2), \\ & i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}, \\ & \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \end{aligned}$$

This solution models the situation in which output from **AD** to **GZ** has already begun. If T_4 is applied to S_i, we have the following reaction.

$$\begin{aligned} & S_i \xrightarrow{T_4} S_{i+1}, \text{ where} \\ & S_{i+1} = \mathbf{CF}_u \diamond o(1), \\ & \qquad \mathbf{CF}_d \diamond i(4), \\ & \qquad \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2), \\ & \qquad i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}, \\ & \qquad o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \end{aligned}$$

Notice that **GZ**, because it has terminated its input of non-compressed data, can immediately begin its output of compressed data to **AD**, which in turn can terminate its writing in order to read from **GZ**.

S_{i+1} admits two possible reactions and, since they do not conflict, they can occur in parallel. One reaction is T_1, the communication of compressed data from **GZ** to **AD**. The other is T_2, which allows **AD** to wait for its output of non-compressed data.

$$\begin{aligned} & S_{i+1} \xrightarrow{T_1, T_2} S_{i+2}, \text{ where} \\ & S_{i+2} = \mathbf{CF}_u \diamond o(1), \\ & \qquad \mathbf{CF}_d \diamond i(4), \\ & \qquad o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1), \\ & \qquad \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD} \diamond i(3), \\ & \qquad \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3) \end{aligned}$$

We can observe that **AD** holds the non-compressed data that it could not yet pass to **GZ**, but in this version of the architecture will not block. From this state, rules T_2, T_3, and T_4 can once again be applied in a nondeterministic—and deadlock free—manner. This reflects the fact that **AD** uses non-blocking writes when sending data to **GZ**.

5.3 Discussion

Clearly, the key role in the system is played by the adaptor component. In particular, the role is that of a "matchmaker", since it must interconnect a CERN HTTP Filter with the **gzip** UNIX filter, each having a different communication modality. In the first architecture, the adaptor initially has a structure that follows the communication style of function-call-based streams. After the application of T_8, the molecule is radically changed to allow communication through UNIX pipes. It then returns to its initial structure by means of the application of T_9. Therefore, we can see that the two communication behaviors are mutually exclusive in the first architecture, leading to the mismatch between **gzip** and the upstream and downstream filters. Conversely, the second architecture permits these communication behaviors to coexist through a concurrent behavior (i.e., multi-threading) of the adaptor. This choice avoids the potential deadlocks exhibited by the first architecture.

The two specifications make use of the same set of processing elements and the same communication channels. Therefore, the two architectures have the same topology. They differ, however, in the internal behavior of the adaptor component, which has a significant effect on the global behavior of the system. This difference is clearly reflected in the augmentation of Σ_b with a parallel operator, the alteration to the adaptor molecule in the initial solution S_0, and the replacement of the two adaptor-specific rules in the set of transformation rules.

In the next section, we use formal techniques to analyze the two architectures for the critical properties that reveal both the mismatch and its resolution.

6 Analysis of the Architectures

A primary reason for why we are exploring the use of the CHAM formalism at the architectural level is that it allows for two, quite different analysis techniques. On the one hand, we exploit the algebraic and equational nature of CHAM to allow us to prove a variety of important properties about an architecture. This was the technique illustrated in general in our earlier work [14]. On the other hand, we take advantage of the CHAM formalism's operational flavor by generating transition systems from specifications and then reasoning at the transition-system level. In fact, we have developed a tool to automate the process of generating a transition system from the CHAM specification of a software architecture; the definitions underlying this process are presented here. The flexibility in analysis techniques provided by the CHAM formalism can be very convenient for architectural engineers since, depending on the kind of property of interest, they can choose the most appropriate technique to apply.

In this section we employ both the algebraic and transition-system techniques in order to uncover the architecture-level mismatch in component behaviors of the Compressing Proxy system. For convenience, Table 1 reproduces the initial solutions and transformation rules for the two architectures presented in Section 5.

As usual when analyzing concurrent systems, we are interested in safety and liveness properties. In this section, we restrict our attention to the analysis of safety properties, which are intended to state that nothing "bad" can ever happen. In the case of the Compressing Proxy, we are interested in analyzing our specifications with respect to deadlock, since that is the erroneous behavior resulting from the use of the first version of the adaptor component. Liveness properties state that something "good" will eventually happen. In Appendix A we use the algebraic technique to show the absence of livelocks—that is, our specifications indicate that the Compressing Proxy cannot run forever unproductively.

It is worth noticing that although the algebraic techniques used here are applied to a specific system, the proof approach is common to a wide range of problems and can be easily adapted to prove similar results on other systems. In particular, as shown in this section, proving the presence of deadlock in iterative systems like the one we are modeling amounts to proving the existence of terminating derivations. In our approach, this reduces to a case analysis on the set of possible derivations. The algebraic structure of the solutions, and the fact that we always only

Blocking Compressing Proxy	Non-blocking Compressing Proxy
Initial Solutions	
$S_0 = \mathbf{CF}_u \diamond o(1),$ $\mathbf{CF}_d \diamond i(4),$ $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ},$ $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$	$S_0' = \mathbf{CF}_u \diamond o(1),$ $\mathbf{CF}_d \diamond i(4),$ $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ},$ $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$ $\parallel i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$
Transformation Rules	
$T_1 \equiv i(x) \diamond m_1,\, o(x) \diamond m_2 \longrightarrow m_1 \diamond i(x),\, m_2 \diamond o(x)$	
$T_2 \equiv e \diamond m \diamond c \longrightarrow c \diamond e \diamond m$	
$T_3 \equiv \mathbf{end}_o \diamond m_1 \diamond o(x),\, \mathbf{end}_i \diamond m_2 \diamond i(x) \longrightarrow m_1 \diamond o(x) \diamond \mathbf{end}_o,\, m_2 \diamond i(x) \diamond \mathbf{end}_i$	
$T_4 \equiv \mathbf{end}_i \diamond m_1 \diamond \mathbf{GZ} \diamond m_2 \longrightarrow m_1 \diamond \mathbf{GZ} \diamond m_2 \diamond \mathbf{end}_i$	
$T_5 \equiv \mathbf{end}_o \diamond \mathbf{GZ} \diamond m \longrightarrow \mathbf{GZ} \diamond m \diamond \mathbf{end}_o$	
$T_6 \equiv \mathbf{GZ} \diamond m \longrightarrow m \diamond \mathbf{GZ}$	
$T_7 \equiv f \diamond c \longrightarrow c \diamond f$	
$T_8 \equiv \mathbf{AD} \diamond i(1) \diamond m \longrightarrow i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$	$T_8' \equiv m_1 \parallel m_2 \parallel \cdots \parallel m_n \longrightarrow m_1,\, m_2, \ldots, m_n$
$T_9 \equiv \mathbf{AD} \diamond i(3) \diamond m \longrightarrow i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$	$T_9' \equiv \mathbf{AD} \diamond m \longrightarrow m \diamond \mathbf{AD}$

Table 1: Initial Solutions and Transformation Rules for the Compressing Proxy Architecture Specifications.

consider derivations starting from the initial solution, greatly simplifies the proof structure. The same reasoning can be applied to the proof of liveness properties.

6.1 Algebraic Analysis

The property we wish to prove about the Blocking Compressing Proxy is that it allows two possible kinds of deadlock, one during the communication of non-compressed data from **AD** to **GZ** and the other in the symmetric case.

We make a first observation, which we employ in the following proofs, about the structure of solutions as controlled by the reaction rules.

Fact 1 *The application of any rule does not change the number of molecules or kind of processing elements in a solution but only transforms the state of the processing elements mentioned in the left-hand side of the applied rule. The only exception is rule T_8' of the Non-blocking Compressing Proxy, which breaks a complex parallel molecule into its constituent parts.*

The significance of this fact is that, given the initial solution S_0, every derived solution in the Blocking Compressing Proxy CHAM will have exactly the same number of molecules as the initial solution, namely four, one for each of the four processing elements of the specification. Given the initial solution S_0' of the Non-blocking Compressing Proxy CHAM and after a finite number of steps in every derivation, all the solutions will have exactly the same number of molecules, namely five, of which two correspond to the parallel threads of the adaptor molecule **AD** and three to the other three processing elements of the specification.

Because the CHAM formalism is inherently concurrent and nondeterministic, we need to restrict our analysis to fair policies in applying reaction rules.

Definition 1 *Let $\mathcal{R}$ be the set of reaction rules for a CHAM C. Then a derivation $D : S_1, S_2, \ldots, S_n, \ldots$ is* fair *if and only if there is no reaction rule in $\mathcal{R}$ whose application can be indefinitely delayed.*

This definition means that if a rule is enabled infinitely often, because its application pattern appears infinitely many times in the derivation, then it is impossible in a fair derivation to avoid applying the rule. Of course, any finite derivation is fair. Fair derivations only guarantee that if something has to be done it will eventually be done. Below, we restrict consideration to fair derivations only. It is worth noting that this restriction is reasonable, since it amounts to an assumption that any implementation of the system will adopt a fair scheduling policy.

We next recall the following definition relevant to all CHAM specifications of software architectures [14].

Definition 2 *A reaction derivation $S_0 \longrightarrow S_1 \longrightarrow \cdots \longrightarrow S_n$ is* normalizing *if S_n is inert.*

This definition means that a given derivation terminates, since a solution is inert when there are no other reaction rules that can be applied to it. Thus, a normalizing derivation is a derivation that *terminates*.

We can now prove the following property.

Proposition 1 *Let S_i be any solution derived from the initial solution by applying the rules of the Blocking Compressing Proxy CHAM. Then S_i contains the pair of molecules $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$ or the pair $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$ and $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$ if and only if S_i is inert or any derivation starting from S_i reaches an inert solution in a finite number of steps.*

Proof: For the "if" part, the proof is by case analysis. By Fact 1, the number and processing nature of the molecules that appear in any derived solution is fixed. If S_i is inert, this means that no reaction rule is applicable and, further, that appearing in S_i are the two molecules $o(1) \diamond \mathbf{CF}_u$ and $i(4) \diamond \mathbf{CF}_d$. For **GZ** and **AD**, then it can only be true that they are in a state in which they are going to perform either an input or an output, otherwise rule T_2 could be applied. This leads to the conclusion that they can only be of the form $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond eof_i$ or of the form $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$ and $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$.

For the "only if" part, we must show that if there is the pair of molecules $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$ in S_i, then there are only a fixed number of reaction steps that can be performed. By Fact 1 this is trivial, since the only possibility is that the filters are in a state that allows the application of T_7 and these are the only possible further reactions. The reasoning for the other pair is analogous. ■

The proposition above permits us to characterize the solutions that lead to normalizing derivations. It shows that any normalizing derivation implies that the adaptor and **gzip** will eventually reach a state in which one is willing to output on channel 2 while the other is willing to output on channel 3. This exactly models the deadlock situation, in which both processing elements are trying to output. The other pair of molecules models the symmetric situation in which deadlock can occur—that is, when both the adaptor and **gzip** are trying to input, one on channel 3 and the other on channel 2.

The next proposition shows that in order to reach an inert solution, it is necessary to apply rule T_4 or rule T_5. These rules model the situations in which **gzip** autonomously decides to end its input or output, thus eventually leading to a deadlock.

Proposition 2 *Let S_0 be the initial solution of the Blocking Compressing Proxy CHAM and let $\delta : S_0 \longrightarrow S_1 \longrightarrow \cdots \longrightarrow S_n$ be a derivation from S_0. Then any derivation δ' from S_n is normalizing if and only if there exists in δ a solution S_i such that $S_i \xrightarrow{T_4} S_{i+1}$ or $S_i \xrightarrow{T_5} S_{i+1}$.*

Proof: Let us first consider whether any derivation δ' from S_n implies the existence of $S_i \xrightarrow{T_4} S_{i+1}$ or $S_i \xrightarrow{T_5} S_{i+1}$ in δ. By the hypothesis δ' is terminating. Since δ' is normalizing this means that the last solution must contain either the pair of molecules $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$ or the pair of molecules $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$ and $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$ by Proposition 1, because otherwise a non-terminating derivation can be easily built. Looking at the rules, this is obtained only through the application of T_4 or T_5. On the other hand, starting from the initial solution, it is certainly possible to apply rule T_4 or T_5. In fact, from the initial solution, a solution is reached deterministically that allows for the application of T_4. If this rule is not applied, then eventually rule T_3 can be applied, thus leading to solutions that allow the application of rules T_8 or T_6. This in turn allows, in a few reaction steps, the production of a solution containing a redex for T_5 or again T_4. This situation can be repeated an infinite number of times if the derivation is non-terminating.

We must now consider the reverse condition. We only consider the application of rule T_4, since the reasoning for rule T_5 is analogous. Let us assume that $S_i \xrightarrow{T_4} S_{i+1}$ exists in δ and let us prove that any derivation δ' from S_n is normalizing. By examining the derivations from the initial solution, $S_i \xrightarrow{T_4} S_{i+1}$ means that either the pair of molecules $\mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \diamond o(2)$ and $o(3) \diamond \mathbf{end}_i \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$ is present in S_{i+1}, or the pair of molecules $o(4) \diamond \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i$ and $\mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3)$ appears in S_{i+1}. In either case, there exists a maximum number of reaction steps that can be further performed on S_n before the pair $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ and $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i$ is created and, by Proposition 1, we have a situation from which only inert solutions are reachable. ■

In our framework, a normalizing derivation models a terminating computation. What we want to do is understand whether this means a *successful* termination or an *incorrect* behavior. For the system under consideration, any terminating derivation signals an incorrect behavior if the data to be processed are still not exhausted. In fact, in the presence of data, the system must work forever. From the above proposition, we know that any normalizing derivation has to apply either rule T_4 or rule T_5. Then, if we look at the solutions we get after their application, it is easy to see that the data are not completely processed.

Let us now define a state of deadlock for the Blocking Compressing Proxy.

Definition 3 *Let S_i be a generic solution of the Blocking Compressing Proxy CHAM and let S_{i+1} be a solution such that $S_i \xrightarrow{T_4} S_{i+1}$ or $S_i \xrightarrow{T_5} S_{i+1}$. Then S_{i+1} defines a* state of deadlock.

This definition allows us to ignore the remaining part of the derivations that start from a solution obtained by the application of T_4 or T_5, since they (incorrectly) terminate, as we have proven in the previous proposition.

We can now prove that we have only two kinds of derivations that terminate.

Proposition 3 *The Blocking Compressing Proxy CHAM allows only two kinds of normalizing derivations,* $S_0 \cdots \longrightarrow S_i \longrightarrow \cdots \longrightarrow S_n$ *or* $S_0 \cdots \longrightarrow S_j \longrightarrow \cdots \longrightarrow S_m$*, such that* $\mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \in S_i$ *and* $\mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3) \in S_j$*.*

Proof: Starting from the initial solution S_0, we can easily see that T_4 or T_5 can be applied only if either $\mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2)$ or $\mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3)$ is in a given solution. In the first case, the application of T_4 corresponds to the situation in which **GZ** interrupts its input before **AD** has terminated its output. The second case corresponds to the symmetric situation. Then, looking at the rules, we can see that there are only two kinds of solutions in which T_4 or T_5 can be applied.

$$S_i = \ldots, \quad \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \qquad \text{and} \qquad S_j = \ldots, \quad \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3)$$

Moreover, these two molecules can obviously never appear in the same solution. This is by Fact 1, since they are both related to **gzip**. ■

At this point we have the following corollary.

Corollary 3.1 *In every normalizing derivation,* T_4 *is applied to solutions that contain the molecule* $\mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2)$ *and* T_5 *is applied to solutions that contain the molecule* $\mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \diamond o(3)$*.*

We use this corollary in the proof of the following proposition.

Proposition 4 *There are exactly two sets of deadlock states in the Blocking Compressing Proxy CHAM.*

Proof: The thesis follows immediately from Definition 3 and Corollary 3.1. ■

What still remains to be proven is that the second architecture removes the potential deadlocks that can occur in the first.

Proposition 5 *Let* S'_0 *be the initial solution of the Non-blocking Compressing Proxy CHAM and let* $\delta : S'_0 \longrightarrow S'_1 \longrightarrow S'_2 \longrightarrow \cdots \longrightarrow S'_n$ *be a derivation from* S'_0*. Then there exist no normalizing derivations starting from* S'_n*.*

Proof: We simply need to show that in the Non-blocking Compressing Proxy CHAM it is impossible to perform a normalizing derivation. The presence of the complex molecule $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \parallel i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$ together with the heating rule T'_9 guarantees that in a generic solution S'_i $(i > 0)$ there are always two molecules for the adaptor, $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$ and $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$ or the molecules evolved from them.

Let us assume that the thesis is not true. Then it should be possible to reach a solution that is inert. This means that no reaction can be performed on that solution. This implies that, as far as the upstream and downstream filters are concerned, their molecules are respectively waiting to output and to input, so that rule T_7 cannot be applied. Further, **GZ** has to be in a state that does not allow the application of T_2, T_4, T_5, or T_6. Then it can only be either waiting for input from **AD** or waiting to output to **AD**. As far as the two molecules of **AD** are concerned, similar to the reasoning for **GZ**, they can only be waiting for input from **GZ** and waiting to output to **GZ**. In both cases we can have a reaction between **GZ** and one of the **AD** molecules, thus contradicting the hypothesis that the solution is inert. ■

We can now state the following result.

Corollary 5.1 *The Non-blocking Compressing Proxy CHAM does not allow deadlock.*

Having formally proven the absence of deadlocks in the second architecture, we can prove that it allows infinite derivations if and only if data for compressing are infinitely available. This fact ensures that there exist no livelocks—that is, situations in which the system makes no progress, although it is not blocked. Livelocks are generated by infinite derivations in which it is always possible to apply at least one reaction rule, although no constructive progress is made. In our case, this would mean that along the derivation, no data processing is achieved. We present these results in Appendix A.

6.2 Transition-system Analysis

Let us now turn to the other kind of analysis made possible by a CHAM specification of an architecture, transition-system analysis. First we show how it is possible to generate a transition system from a CHAM description. Basically, we use the usual approach of deriving the transition system from the operational semantics [10, 20, 24] by considering that our reaction rules are indeed the operational semantic rules. Note, however, that due to the concurrent operational nature of the CHAM, we must also consider all the transitions in which sets of disjoint redexes can be applied. In terms of the generated transition system, this does not imply any increase in the number of states but only in the number of arcs to be considered.

The following definitions provide the generation mechanism.

Definition 4 (Operational semantics induced by $\mathcal{R}$.) *Let $\mathcal{R}$ be the set of reaction rules of a CHAM C. Then $\mathcal{R}$ defines a relation $D \subseteq Molecules \times Molecules$. The relation is the least relation satisfying the rules.*

Definition 5 (Derivative.) *Given a set of reaction rules $\mathcal{R}$, an $\mathcal{R}$-derivation from a solution S_0 to a solution S_n is a sequence $\{S_i, 0 \leq i \leq n\}$, $n > 0$, such that for any $0 \leq i \leq n-1$, $S_i \longrightarrow_{\mathcal{R}} S_{i+1}$. A solution S_j is called an $\mathcal{R}$-*derivative *of S_i if an $\mathcal{R}$-derivation exists from S_i to S_j. The set of derivatives of S_0 is called $D_{\mathcal{R}}(S_0)$.*

Definition 6 (Transition System.) *A transition system T is a triple (S, D, s_0), where S is a set of solutions, $s_0 \in S$ is the initial solution, and $D \subseteq S \times S$.*

Definition 7 (Transition System Corresponding to a Solution.) *Given a solution S and a set of reaction rules $\mathcal{R}$, $\mathcal{R}(\mathcal{S})$ is the transition system $(D_{\mathcal{R}}(S) \cup \{S\}, D, S)$, where D is the relation defined by $\mathcal{R}$.*

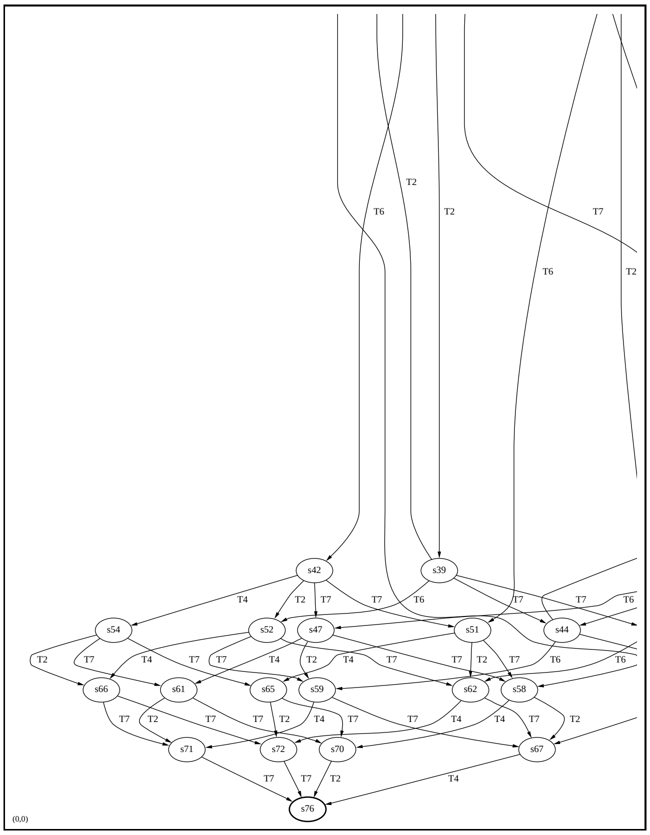

Figure 2: Excerpt from the Blocking Compressing Proxy Transition System.

In this way, given a CHAM and a solution, we can generate a transition system that represents the complete set of possible derivations. If the number of derivable solutions is finite, then the transition system is also finite.

A portion of the transition system for the Blocking Compressing Proxy is depicted in Figure 2. Each node represents a unique solution and each directed arc represents a transition applied to a solution to form another solution. The arcs are labeled with the transformation rules from Table 1. The graph was produced by a tool that we developed to generate transition systems from CHAM architecture specifications [22].

In the figure we see a solution with no outgoing arcs, namely S_{76}. The full transition system contains a second solution, S_{35}, that also has no outgoing arcs. These two states represent deadlocks in the system. It is clear from the full graph that all paths leading from S_0, the initial solution, to S_{35} or S_{76}, the deadlock solutions, involve an arc labeled T_4 or labeled T_5, respectively. This confirms our earlier result that the application of T_4 or of T_5 leads to deadlock.

The existence of exactly two deadlock states, S_{35} and S_{76}, in the transition system is in accord with Proposition 4, which identifies two sets of deadlock states. It is useful to recall that in the algebraic proposition we are referring to the solutions to which T_4 or T_5 can be applied, while in the transition system we are referring to the deadlock solutions themselves. The sets of solutions of Proposition 4 can be found by tracing back through the various paths that terminate at S_{35} or at S_{76}. If we look at the solutions corresponding to the two deadlock states we find, as expected, the two identified pairs of molecules of Proposition 1 that characterize the deadlock configurations of the system.

7 Conclusion

In this paper we have presented and discussed the specifications of two architectures for the Compressing Proxy case study. Our specifications are kept at a rather high level of abstraction, with many details of the system behavior left unaddressed. For example, we have purposely ignored the question of granularity of the data manipulated by the components that constitute the Compressing Proxy. This is not a weakness of the formalism, but rather an explicit degree of freedom in interpreting the architectural descriptions (e.g., purposely leaving certain implementation choices up to developers). If more constraints on the admissible behaviors are desired in the descriptions of the architectures, then the CHAM model allows us to provide more details in the specifications. Our goal is to see if the CHAM descriptions can usefully reveal the architectural mismatch inherent in the Compressing Proxy architecture. From the given specifications, this can be simply derived and formally proved.

It is interesting to note that the uses of the two kinds of analysis techniques, algebraic and transition, are actually complementary. In fact, while the proof of certain properties can be easier at the transition-system level, such as the absence/presence of deadlock/livelock, it can be more complex to use this level of analysis to understand what has to be done in order to prevent or correct these situations. The transition model is, from this point of view, too abstract, since it can be difficult to relate problems to the structure of the solutions. On the other hand, the analysis at the algebraic level can be tedious and complicated for certain properties, but it is highly informative, since it maintains all the information about the structure of the system. Of course, when the system under specification has an infinite number of states, then the use of algebraic techniques is the only practical choice.

In general, we advocate a mixed analysis strategy. Our goal is to be able to reason about a system at the level of software architecture in order to prove non-trivial properties of the system. In case we can discover a problem at this level of specification, such as the deadlock in the Blocking Compressing Proxy architecture, we would like to be assisted in the analysis that leads to a correction in the architecture. An environment that allows the automatic derivation of a transition system from a CHAM description plus an inference engine that allows one to simulate derivations at the CHAM level, can serve the purpose. In this way, the analysis strategy can proceed in two steps. First a transition system is generated and analyzed in order to identify the critical states and derivations. Then the inference engine can be used to execute the identified derivation and solutions. Thus, we can obtain a more informative view of the critical behaviors of the system with which it is possible to reason and understand the ways in which mismatches in behavior can be corrected.

The ability to generate a transition system allows the application of model-checking techniques, once the properties to be proved are expressed in a suitable logic. This can be very useful when analyzing alternative architectures of the same system that can be characterized by means of invariant properties. In fact, we have already begun to exploit this approach [15].

REFERENCES

[1] G.D. Abowd, R. Allen, and D. Garlan. Formalizing Style to Understand Descriptions of Software Architecture. *ACM Transactions on Software Engineering and Methodology*, 4(4):319–364, October 1995.

[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] D. Compare. Specifica ed Analisi del CERN Compressing Proxy con la CHAM. Technical Report Tesi di Laurea, Dipartimento di Matematica Pura ed Applicata, L'Aquila, Italy, December 1995.

[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] Formal Systems, Ltd. *Failures Divergence Refinement: User Manual and Tutorial*. Formal Systems, Ltd., Oxford, England, October 1992.

[10] N. De Francesco and P. Inverardi. Proving Finiteness of CCS Processes by Non-standard Semantics. *Acta Informatica*, 31(1):55–80, 1994.

[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 A.L. Wolf. On the Choice of a Software Architecture: A Guided Tour in the Analysis of Architectural Design. Available from the authors, October 1996.

[16] P. Inverardi, A.L. Wolf, and D. Yankelevich. Checking Assumptions in Component Dynamics at the Architectural Level. In *Proceedings of the Second International Conference on Coordination Models and Languages*, number 1282 in Lecture Notes in Computer Science, pages 46–63. Springer-Verlag, September 1997.

[17] 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.

[18] D.C. Luckham and J. Vera. An Event-based Architecture Definition Language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.

[19] D. Le Métayer. Software Architecture Styles as Graph Grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 15–23. ACM SIGSOFT, October 1996.

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

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

[22] A. Rosetti. Generazione di Test Cases da Specifiche Formali della Architettura Software. Technical Report Tesi di Laurea, Dipartimento di Matematica Pura ed Applicata, L'Aquila, Italy, March 1997.

[23] J.M. Spivey. *Understanding Z: A Specification Language and its Formal Semantics.* Cambridge University Press, 1989.

[24] D. Taubner. *Finite Representations of CCS and TCSP Programs by Automata and Petri Nets.* Number 369 in Lecture Notes in Computer Science. Springer-Verlag, 1989.

A Liveness Properties

In this section we prove that the Non-blocking Compressing Proxy CHAM is free of livelocks.

Proposition 6 *The Non-blocking Compressing Proxy CHAM allows infinite derivation if and only if data for compressing are infinitely available.*

Proof: Input of new external data is modeled by the application of rule T_7 on the molecule $CF_u \diamond o(1)$. We need to show that any derivation in the Non-blocking Compressing Proxy CHAM, since there are no deadlocks and therefore all the derivations are infinite, contains an infinite number of occurrences of rule T_7 applied to the molecule $CF_u \diamond o(1)$. Let us assume that this is not true. Then there must exist an infinite derivation that contains only a finite number of applications of rule T_7 on the molecule $CF_u \diamond o(1)$. This is clearly impossible since after a finite number of reaction steps, the fact that there is no longer an occurrence of the molecule $o(1) \diamond CF_u$ will prevent a reaction with the molecule $i(1) \diamond o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD}$, which in turn will eventually react with **GZ**, thus contradicting the hypothesis. ■

This last proposition can be easily shown to also hold for the first architecture.

Proposition 7 *The Blocking Compressing Proxy CHAM allows infinite derivations if and only if data for compressing are infinitely available.*

Another interesting property that we want to prove is that all the different sets of data that are manipulated by the Non-blocking Compressing Proxy are eventually processed. This fact becomes clearer if we keep in mind that **AD** can operate in parallel and that it performs a sequence of outputs followed by a sequence of inputs when it communicates with **GZ**. This amounts to showing that we cannot have derivations in which there is a molecule representing a pending output of the adaptor, where that molecule is never utilized in a transformation. Of course, we must also prove an analogous property for **GZ**, since its input/output behavior is also modeled by $[I^+O^+]^*$.

Proposition 8 *Let S'_0 be the initial solution of the Non-blocking Compressing Proxy CHAM and let $\delta : S'_0 \longrightarrow S'_1 \longrightarrow \cdots \longrightarrow S'_n \longrightarrow \cdots$ be an infinite derivation from S'_0. Then there exists no $j > 0$ such that $\forall k \geq j$*

1. *the molecule $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \subset S'_k$; and*
2. *the molecule $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \subset S'_k$.*

Proof: Let us consider the first point. We prove the claim by contradiction. Assume that in δ there exists $j > 0$ such that $\forall k \geq j$ $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1) \subset S'_k$. In other words, we have a derivation $S'_j \longrightarrow S'_{j+1} \longrightarrow \cdots$ from S'_j in which $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$ is always present. Therefore no reaction involving this molecule occurs. Since only T_1 can react with it, this amounts to assuming that the molecule $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ}$ never occurs in δ. By case analysis on the structure of Σ_n, we are sure that there exists a maximum number of reaction steps that can be performed before we have a situation from which no further reaction is possible. This fact implies that δ is a normalizing derivation, thus contradicting the hypothesis.

The second point is analogous. For brevity, the proof is not shown. ■

To complete our formal analysis we want to show that the different sets of data manipulated by the Non-blocking Compressing Proxy CHAM do not mix—that is, we want to prove that the external data are actually processed. In fact, because of the concurrent behavior of **AD**, we need to be sure that the data coming from an upstream filter are all sent from **AD** to **GZ**. This fact guarantees that the integrity of the data is preserved.

Proposition 9 *Let* $S'_0 \longrightarrow S'_1 \longrightarrow \cdots \longrightarrow S'_n$ *be a derivation of the Non-blocking Compressing Proxy*
CHAM such that the molecule $o(4) \diamond \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i \subset S'_n$*. Then there exist* S'_r *and* S'_t*, where* $0 < r < t < n$*, such that the subsolution* $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$*,* $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \subset S'_r$ *and the subsolution* $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$*,* $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \subset S'_t$*.*

Proof: Let us assume by contradiction that there does not exist S'_r and S'_t with $0 < r < t < n$ such that $o(2) \diamond \mathbf{end}_o \diamond \mathbf{AD} \diamond i(1)$, $i(2) \diamond \mathbf{end}_i \diamond o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \subset S'_r$ and $i(3) \diamond \mathbf{end}_i \diamond o(4) \diamond \mathbf{AD}$, $o(3) \diamond \mathbf{end}_o \diamond \mathbf{GZ} \diamond i(2) \diamond \mathbf{end}_i \subset S'_t$. We know that the ability of a processing element to communicate is syntactically indicated by the appearance of a communication port in the left-most position of the molecule. Completion of the communication is indicated by a rewriting of the molecule such that the communication port is moved to the right-most position of the molecule. Therefore, keeping in mind the structure of S'_0 and how T_1 describes communication between processing elements, it follows that $o(4) \diamond \mathbf{AD} \diamond i(3) \diamond \mathbf{end}_i \not\subset S'_n$. This leads to a contradiction. ■