Formal Specification and Analysis of Software Architectures Using the Chemical Abstract Machine Model

Paola Inverardi

Istituto di Elab. dell Informazione
Consiglio Nazionale dell Ricerche
I-56126 Pisa, Italy

(inverard@iei.pi.cnr.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-709-94 March 1994
(Revised September 1994)

A version of this report to appear in
The IEEE Transactions on Software Engineering

ABSTRACT

We are exploring a novel approach to formally specifying and analyzing software architectures that is based on viewing software systems as chemicals whose reactions are controlled by explicitly stated rules. This powerful metaphor was devised in the domain of theoretical computer science by Banâtre and Le Métayer and then reformulated as the Chemical Abstract Machine, or CHAM, by Berry and Boudol. The CHAM formalism provides a framework for developing operational specifications that does not bias the described system toward any particular computational model and that encourages the construction and use of modular specifications at different levels of detail. We illustrate the use of the CHAM for architectural description and analysis by applying it to two different architectures for a simple, but familiar, software system, the multi-phase compiler.

This work was supported in part by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo.

1 Introduction

Researchers and practitioners have begun to recognize the importance of studying the architectures of large and complex software systems. There is a wide variety of reasons for concentrating effort on this particular area of software engineering, including improved education, increased reuse, reduced development cost, and enhanced quality. Yet before we can begin to attack even one of these goals, software architectures must become describable in some way. And while it is generally agreed that this is a critical element in the study of software architectures, it is still not clear how they can best be consistently and rigorously specified.

A complete specification of a software architecture will draw simultaneously upon many different descriptive techniques, from pictures to text to mathematical formulae, each addressing a specific aspect of the problem. In this paper we develop a novel approach to the formal specification of software architectures. The motivations for using a formal framework here are the same as those for any use of a formal technique, namely semantic precision, uniform description, common basis for formal comparison, susceptibility to formal analysis, and the like. But in choosing a suitable semantic framework for software architecture specification, we are faced with some especially difficult challenges. On the one hand, we need a general and flexible formalism in which it is possible to describe very different kinds of architectures within the same application domain, since we will want to compare the properties of those architectures as part of the process of selecting among them. On the other hand, the description has to be understandable to the widely varying consumers of a specification, who range from implementors to maintainers to purchasers to perhaps even users of the software. This is a quite different and more difficult task than is the task of defining a specification formalism that can be more narrowly targeted or for which we can make simplifying assumptions about the background and training of consumers of the specification.

This aspect of software architecture specification suggests to us that an operational semantic formalism is the most appropriate choice [11, 15], in the belief that operational semantics can be more easily understood by a broader range of practitioners than other, abstract mathematical formalisms. Operational semantics reflects the familiar idea of specifying the computational behavior of a system in terms of the behavior of a more abstract and precisely defined system.

The use of operational semantics in software architecture specification, however, can have a major drawback. Since the behavior of the system is described in terms of the behavior of another system, it can easily happen that the semantic description, and hence the reasoning conducted on that description, can become biased by the operational framework. Consider, for example, the case of an operational description given in terms of an abstract sequential machine. In this case, the description of a sequential architecture can be straightforward, but the description of a parallel architecture will amount to defining an implementation of parallelism in the limiting terms of sequential behavior. The same is true the other way around if we choose an abstract parallel machine as our semantic basis. As another example, consider the choice between a functional and a state-based abstract machine. The ideal approach is to use an operational semantic formalism that is based on a more flexible, relatively neutral computational model.

To that end, we are exploring the suitability of the so-called Chemical Abstract Machine

(CHAM) model for architectural description and analysis. This recent model was proposed by Berry and Boudol in the domain of theoretical computer science [5]. Under the CHAM approach, the abstract machine is fashioned after chemicals and chemical reactions [3]. Metaphorically, the states of the machine are chemical solutions, where floating molecules can only interact according to a stated set of reaction rules. The CHAM formalism is very powerful, having already been used to describe several different and important computational paradigms. It encourages the construction of modular specifications that can be given at various levels of detail. This is particularly important in the area of software architecture, where the interesting architectures will tend to be large, complex, and assembled from existing components. Finally, the CHAM model is based on, and can profitably make use of, the well-established theoretic foundation of *term rewriting systems* [10].

The remainder of this paper is organized as follows. In the next section we review a model of software architecture description introduced by Perry and Wolf [14]. Following that, we review the general CHAM approach to formal specification. We then illustrate how the CHAM approach can be used in architectural specification by formalizing the description of certain critical aspects of two different architectures for a simple, but familiar, software system, the multi-phase compiler. In Section 5 we demonstrate the utility of CHAM-based descriptions in analyzing software architectures. We provide some concluding thoughts about the CHAM model and a discussion of related work in Section 6.

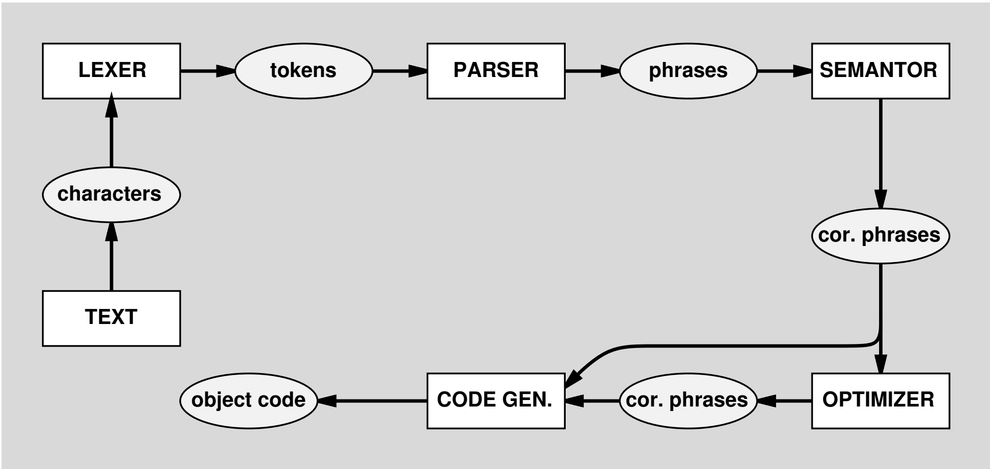
2 A Model of Architectural Description

Motivated by the need to find ways of describing software architectures, Perry and Wolf developed what amounts to a model of architectural description [14]. The purpose of the model is to create a framework within which the designs of proposed specification techniques can be both driven and evaluated. This section reviews the model and provides an example based on one first introduced by Perry and Wolf [14]; we use that example, a multi-phase compiler, in sections 4 and 5 to illustrate our use of the CHAM approach in formally specifying analyzing software architectures.

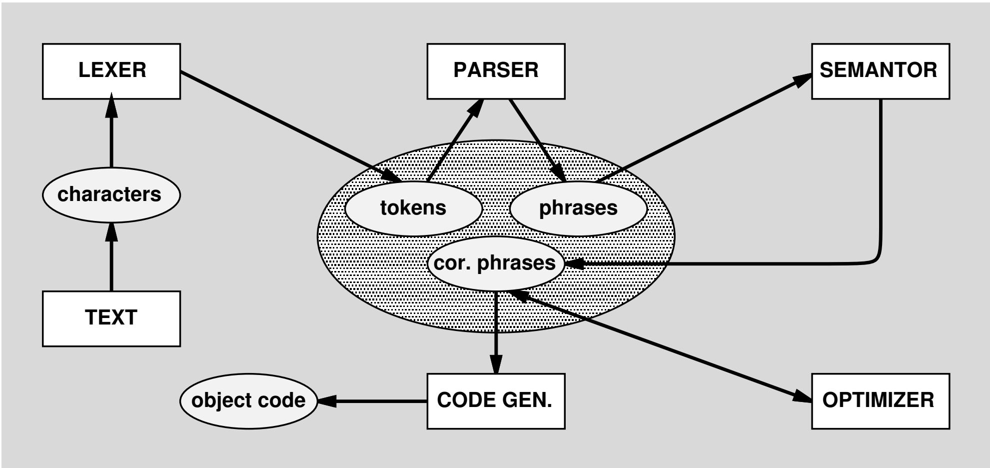
By analogy to civil architecture, a description of a software architecture is a triple consisting of the sets *elements*, *form*, and *rationale*. Elements are the discrete components, or building blocks, of architectures together with intrinsic constraints on their use. Form is the complex of relationships among the elements, and constraints on those relationships, that dictate how the elements are put together. Rationale is the justification for the particular choices of elements and form.

The model identifies three basic kinds of architectural elements: *processing elements*, *data 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, as are the elements that support the overall execution model of the system being described.

Consider now the multi-phase compiler. The processing elements are the lexer (i.e., lexical analyzer), the parser, the semantor (i.e., semantic analyzer), and the code generator. An optional



(a)



(b)

Figure 1: Sequential (a) and Concurrent, Shared Repository (b) Multi-Phase Compiler Architectures.

processing element is the optimizer. The data elements are characters, tokens, phrases, correlated phrases (i.e., phrases signifying name uses related to phrases signifying name declarations), and object code.

The "classical" multi-phase compiler architecture (Figure 1a) fits the processing elements together sequentially in the obvious way, with each processing element running its phase to completion before passing data elements on to the next processing element. A rather different architecture (Figure 1b) fits the processing elements together via concurrent access to a shared repository. The processing elements run their phases opportunistically and in parallel so that, for example, the semantor can correlate phrases at the same time as the lexer is creating new tokens.

Thus, we can see that what distinguishes these two variants of the multi-phase compiler are the connecting elements used in their respective architectures. Our goal is to be able to capture and highlight these sorts of differences within a formal framework.

3 The Chemical Abstract Machine Model

The CHAM model is a rich, multi-faceted formulation that contains many interesting and useful new concepts. 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]. Intuitively, a Γ computation is a set of transformations, or reactions, that consume elements of the multiset and produce new ones according to the rules that constitute the program. Since reactions on disjoint subsets can take place in any order or even simultaneously, the model is inherently parallel. It has been pointed out [5, 6] that Petri nets are a well-known example of the multiset transformation style of programming, where markings are multisets of places and each transition of the net can be seen as a reaction rule to transform the markings. Hence, there is already a large body of experience with this kind of modeling and it has been shown to be a powerful and useful approach.

The CHAM formalism extends the Γ language by specifying a syntax for molecules and by providing a classification scheme for reaction rules. It also introduces the membrane construct, described below, which extends the use of multisets in such a way that they can form parts of molecules. As shown by Berry and Boudol [5], this gives the CHAM formalism the power of classical process calculi and the behavior of concurrent generalizations of the lambda calculus.

In this section we briefly review the CHAM model, limiting ourselves to only those concepts directly required for this paper. The interested reader is referred elsewhere [5] for a complete description of the model and examples of its use in formally capturing the semantics of older, more familiar models, such as the CCS process calculus [12]. Boudol [6] mentions that the CHAM has also been demonstrated as a modeling tool in other areas, from graph reduction to concurrent-language definition and implementation. In sections 4 and 5, we introduce the CHAM into the domain of software architecture specification and analysis.

Basics. A Chemical Abstract Machine is specified by defining *molecules* $m, m', \ldots$ 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 also 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.

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.

Any solution can be considered as a single molecule with respect to other solutions by means of an encapsulation construct called a *membrane*. More importantly, a membrane allows the effects of a transformation to be localized to within that membrane. In other words, the solutions inside a membrane can freely evolve independently of other solutions. A reversible operator called an *airlock* is used to selectively extract molecules from a solution and place the rest of that solution within a membrane. In a sense, the airlock is acting as a membrane constructor. The reversibility of the airlock allows molecules to be "reabsorbed" into the original solution. Finally, membranes are semi-permeable, allowing certain molecules to enter and leave the membrane.

Molecules, Solutions, and Membranes. Molecules are defined as terms of a syntactic algebra that derive from a set of constants and a set of operations specific to a CHAM. Solutions $S, S', \ldots$ are finite multisets of molecules, each denoted as a comma-separated list of molecules

$$m_1, m_2, \ldots, m_n$$

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 $S' = m'_1, \ldots, m'_k$, $S \uplus S' = m_1, \ldots, m_n, m'_1, \ldots, m'_k$ is another solution.

A solution enclosed in $\{\!|\cdot|\!\}$ denotes a membrane. The reversible airlock operator applied to solution $S = m_1, \ldots, m_n$ to extract m_i from S is denoted

$$S = m_i \triangleleft \{\!| m_1, \ldots, m_n |\!\}$$

The CHAM model provides a context abstraction, denoted as $\mathcal{C}[\cdot]$, derived from the λ-calculus. The purpose of this construct is to allow implicit reference to an arbitrary set of molecules $\mathcal{C}$ within which a given molecule can be placed. Examples of its use appear below.

General Laws. CHAMs obey four general laws.

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

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

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

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

The Membrane Law. A subsolution can evolve freely within any context.

$$\frac{S \longrightarrow S'}{\{\!|\mathcal{C}[S]|\!\} \longrightarrow \{\!|\mathcal{C}[S']|\!\}}$$

The Airlock Law. A molecule can always be extracted from, and reabsorbed into, a solution at the same time that its identity as an individual molecule is preserved.

$$m \uplus S \longleftrightarrow m \triangleleft \{\!|S|\!\}$$

Rules. 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 then we have nondeterminism, in which case the CHAM makes a nondeterministic choice as to which transformation to perform.

Berry and Boudol discuss three kinds of rules that appeal to our intuition about chemical reactions and that serve informally to help structure the conception of rules: *heating rules*, *cooling rules*, and *reaction rules*. A rule is a heating if it decomposes a molecule into its constituents. Conversely, a rule is a cooling if it composes a compound molecule. Finally, a rule is a reaction if it actually changes the nature of the solutions. Following the intuition further, a solution is said to be *hot* (*frozen*) if no heating (cooling) rule is applicable. Similarly, a solution is said to be *inert* if no reaction rule is applicable to it or to a structurally equivalent solution.

It turns out that we find it useful to define a somewhat more liberal definition of heating that, while still structural, is not strictly bound to a notion of structural decomposition of a molecule. In the example given in Section 4, we consider heating rules to be those that change the structure of a molecule so that it can become reactive. Our heating rules are not, therefore, strictly classifiable as decomposing a molecule into its constituents.

It is interesting to note that, given the right set of rules, a possibly infinite amount of effort can be spent by a CHAM unproductively looping between heating molecules and then cooling them back down. This is apparently a drawback of the CHAM approach to behavior specification. As observed by Berry and Boudol [5], however, it is a direct consequence of the abstract machine approach, since a machine not only *performs* transformations but *searches* for them as well. Moreover, it is possible to suitably constrain the behavior of a particular CHAM, and so there is no need to do this at the level of the basic mechanisms of the CHAM. In fact, leaving the basic mechanisms unconstrained is important to us because we want to introduce a constraint only if it is useful in the description of a particular software architecture.

4 Specifying Architectures Using CHAMs

In this section we demonstrate how CHAMs can be effectively used to formally specify software architectures. In essence, our approach is to express the structure of processing, data, and connecting elements through definitions of molecules, solutions, membranes, and transformation rules. Below, we give specifications for the multi-phase compiler architectures described in Section 2. The example is purposefully kept simple and focused to highlight the important aspects of our approach. Thus, the formalization concentrates on the distinguishing features of the architectures, namely their different connecting elements. Note that in keeping the example simple, we are specifying the architectures at a rather high, although informative, level. It would certainly be appropriate within the CHAM model to incorporate additional detail into those descriptions.

4.1 Sequential Multi-Phase Compiler

We begin with the sequential architecture. As discussed above, a chemical abstract machine is specified by defining molecules, solutions, and transformation rules. In order to define molecules, we must define an algebra of molecules or, in other words, a syntax by which molecules can be built. For the multi-phase compiler architecture, we start with a set of constants P representing the processing elements, a set of constants D representing the data elements, and an infix operator "$\diamond$" that we use to express the status of a processing element. The connecting elements for the sequential architecture are given by a third set C consisting of two operations, i and o, that act on the data elements. The syntax Σ_{seq} of molecules M in the sequential architecture is then

$$
\begin{aligned}
M &::= P \mid C \mid M \diamond M \\
P &::= \mathsf{text} \mid \mathsf{lexer} \mid \mathsf{parser} \mid \mathsf{semantor} \mid \mathsf{optimizer} \mid \mathsf{generator} \\
D &::= \mathsf{char} \mid \mathsf{tok} \mid \mathsf{phr} \mid \mathsf{cophr} \mid \mathsf{obj} \\
C &::= i(D) \mid o(D)
\end{aligned}
$$

As usual, we take as the set of syntactic elements the initial algebra in the class of all the Σ_{seq} algebras.

Let us provide some intuition behind this syntax. Each element of D denotes one or more instances of the represented data. Thus, for example, "char" denotes one or more characters and "tok" denotes one or more tokens. We use the two operations i and o to represent data-element communication ports, where i is for input and o is for output. Finally, the infix operator "$\diamond$" is used to express the state of a processing element with respect to its input/output behavior.

Consider, for example, the parser molecule $i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}$. It says that the parser consumes tokens as input and produces phrases as output. The state of the parser is understood by "reading" the molecule from left to right. In this example, the parser is in the state of consuming tokens and must wait to produce phrases until after it has stopped consuming tokens. We are treating the left-most position (i.e., the left operand of the left-most "$\diamond$" operator) in the molecule as special; if this position is occupied by a communication port, then the kind of communication represented by that port can take place. So, for example, the molecule $o(\mathsf{phr}) \diamond \mathsf{parser} \diamond i(\mathsf{tok})$ represents the parser in the state of making phrases available to the environment as output after having consumed all input tokens. The molecule $\mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr})$ represents the parser in the state of having consumed all input tokens and produced all output phrases; in this state, the parser is unable to communicate with other processing elements. As discussed below, the transformation rules syntactically rearrange the molecules to reflect changes in state.

The next step in specifying the sequential compiler architecture is to define an initial solution S_1. This solution is a subset of all possible molecules that can be constructed under Σ_{seq} and corresponds to the initial, static configuration of a system conforming to the architecture. Transformation rules applied to the solution define how the system dynamically evolves from its initial configuration.

$$\begin{aligned} S_1 = {} & o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \end{aligned}$$

Notice that S_1 contains the molecule $o(\mathsf{char}) \diamond \mathsf{text}$ to represent the existence of an initial source text.

The final step is to define two simple transformation rules.

$$T_1 \equiv i(d) \diamond m,\ o(d) \diamond m_1 \longrightarrow m \diamond i(d),\ m_1 \diamond o(d)$$

$$\begin{aligned} T_2 \equiv {} & o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr}) \longrightarrow \\ & o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \end{aligned}$$

where $m, m_1 \in M$ and $d \in D$. The first rule is a reaction rule that generically describes pairwise input/output communication between processing elements. In particular, communication occurs if there is a processing element m that consumes input d produced as output by some other processing element m_1. Recall that the ability of a processing element to communicate data is syntactically indicated by the appearance of a data port 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 data port is moved to the right-most position of the molecule. The second transformation rule restores the processing elements to their initial states after the code generator has completed its task and offered object code as output. This allows the system to begin processing a new source text. T_2 is therefore a heating rule by our definition in Section 3, since it restructures molecules in such a way that they become reactive; this is illustrated below.

Let us trace through applications of the transformation rules to see how our formulation captures the essence of the architecture. It is easy to see that the only possible reactions on S_1 are the ones that reflect the sequentiality in the processing intended for the architecture. Thus, the only reaction that can occur at this point is within the subsolution consisting of molecules $o(\mathsf{char}) \diamond \mathsf{text}$ and $i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer}$.

$$o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer} \xrightarrow{T_1} \mathsf{text} \diamond o(\mathsf{char}),\ o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char})$$

This represents the initial processing performed by the compiler, namely the consumption of characters by the lexer. We use the Chemical Law given in the previous section to say that the whole solution S_1 has evolved by T_1 due to the evolution of a subsolution.

$$\begin{aligned} &S_1 \xrightarrow{T_1} S_2, \text{ where} \\ &S_2 = \mathsf{text} \diamond o(\mathsf{char}),\ o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}),\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ &\qquad i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ &\qquad\quad i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \end{aligned}$$

Notice that the molecule $\mathsf{text} \diamond o(\mathsf{char})$ in S_2 is rendered inert by this reaction, since there is no transformation rule that can be applied to it. This represents the fact that all the text has been consumed by the lexer. The next reaction that can occur is on the subsolution consisting of molecules $o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char})$ and $i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}$, which represents the parser's consumption of tokens produced by the lexer.

$$\begin{aligned} &o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}),\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser} \xrightarrow{T_1} \\ &\quad \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ o(\mathsf{phr}) \diamond \mathsf{parser} \diamond i(\mathsf{tok}) \end{aligned}$$

The reaction causes the molecule $\mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok})$ to become inert, which represents the fact that the lexer has completed its processing. The subsolution consisting of molecules $o(\mathsf{phr}) \diamond \mathsf{parser} \diamond i(\mathsf{tok})$ and $i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor}$ is the next to react in a manner analogous to the previous reactions.

$$\begin{aligned} &o(\mathsf{phr}) \diamond \mathsf{parser} \diamond i(\mathsf{tok}),\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} \xrightarrow{T_1} \\ &\quad \mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr}),\ o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr}) \end{aligned}$$

The original solution S_1 has now evolved several times, resulting in the following solution, S_4.

$$\begin{aligned} &S_1 \xrightarrow{T_1} \cdots \xrightarrow{T_1} S_4, \text{ where} \\ &S_4 = \mathsf{text} \diamond o(\mathsf{char}),\ \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ \mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr}), \\ &\qquad o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ &\qquad\quad i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \end{aligned}$$

We have constrained each processing element in this architecture to react only upon the existence in the environment of some other element that produces the appropriate data element. Moreover, we have caused each processing element to consume and produce its entire input and output in a non-incremental fashion. Of course, this is exactly what we want to express for the concept of a sequential multi-phase architecture.

At this point we can have two different evolutions of the solution S_4. The first possible evolution involves a reaction between the semantor and the optimizer, followed by a reaction between the optimizer and the code generator.

$$o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer},\ i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \xrightarrow{T_1}$$
$$\mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}),\ o(\mathsf{cophr}) \diamond \mathsf{optimizer} \diamond i(\mathsf{cophr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \xrightarrow{T_1}$$
$$\mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}),\ \mathsf{optimizer} \diamond i(\mathsf{cophr}) \diamond o(\mathsf{cophr}),\ o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$$

The second possible evolution from S_4 simply involves a reaction between the semantor and the code generator.

$$o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer},\ i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \xrightarrow{T_1}$$
$$\mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer},\ o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$$

This reflects, through nondeterminism, the fact that an optimizer is an optional processing element in the architecture. Either one of these two possible evolutions of the system will end with a solution containing the molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$, thus representing the fact that the compiler system has reached the final state of producing the object code from the initial source text.

$$S_4 \xrightarrow{T_1} \cdots \xrightarrow{T_1} S_5 \text{ or } S_4 \xrightarrow{T_1} S_5', \text{ where}$$

$$\begin{aligned} S_5 = {} & \mathsf{text} \diamond o(\mathsf{char}),\ \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ \mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr}), \\ & \quad \mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}),\ \mathsf{optimizer} \diamond i(\mathsf{cophr}) \diamond o(\mathsf{cophr}), \\ & \qquad o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr}) \end{aligned}$$

$$\begin{aligned} S_5' = {} & \mathsf{text} \diamond o(\mathsf{char}),\ \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ \mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr}), \\ & \quad \mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}),\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & \qquad o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr}) \end{aligned}$$

Notice that S_5' contains the molecule corresponding to the initial state of the optimizer—that is, $i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}$—which shows that the optimizer has not been used.

It is now possible to apply the transformation T_2 to the solution, either in its S_5 or S_5' form, due to the presence of the molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$. As we mention above, this transformation rule can be considered a heating rule; it allows the solution to reach a state from which other reactions can start. In practice it corresponds to the iterative behavior of a compiler system, which when terminated after a compilation can actually start again compiling a completely new source text. Let us apply rule T_2 to S_5'.

$$S_5' \xrightarrow{T_2} S_6, \text{ where}$$

$$\begin{aligned} S_6 = {} & o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & \mathsf{text} \diamond o(\mathsf{char}),\ \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ \mathsf{parser} \diamond i(\mathsf{tok}) \diamond o(\mathsf{phr}), \\ & \mathsf{semantor} \diamond i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \end{aligned}$$

Notice that the solution now contains a certain amount of "junk". One bit of junk is the unused optimizer molecule left over from the previous reactions, which results in a duplication of the reactive optimizer molecule. The reactions we are modeling are strictly sequential and, therefore, the existence of two identical molecules does not increase the reactive potential of the system. This is not in general true and is why solutions are defined as multisets, not simply sets. For this example, however, it would be useful to remove the redundant molecule through a rule that "cleans" the solution. For this purpose we introduce T_3.

$$\begin{aligned} T_3 \equiv {} & i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer} \longrightarrow \\ & i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer} \end{aligned}$$

With this rule, the redundant molecule can be eliminated from consideration. The other bits of junk in the solution are the inert molecules representing the "spent" processing elements. These, too, are unnecessary and could be removed with the introduction of one or more additional cleaning rules that, for brevity, we do not show.

Of course, we could have chosen to fold all cleaning into the definition of T_2; we chose to use separate cleaning rules for expository purposes. For brevity, we do not make use of cleaning rules in the remainder of this paper, since their effect is merely to keep the solution minimal with respect to its reactive capabilities and have no material impact on the modeling of the architectures. Indeed, from a formal perspective, the Chemical Law allows us to restrict our attention to just the relevant (non-junk) molecules.

Finally, it is worth noting that we have overspecified the sequential compiler architecture. In order to describe the way processing and data elements are connected and behave in this architecture, there is really no need to model the iterative behavior nor, consequently, the cleaning of solutions. In fact, rule T_1, together with the molecule syntax Σ_{seq} and the initial solution S_1, are enough by themselves. We have performed this extra degree of specification simply to demonstrate more of the modeling power of CHAMs.

To summarize, the specification of the sequential multi-phase compiler architecture is composed of molecular syntax Σ_{seq}, initial solution S_1, and transformation rules T_1 and T_2. We do not include T_3 in the specification because, as explained above, it is a cleaning rule that does not affect the reactive capability of the CHAM.

4.2 Concurrent, Shared Repository Multi-Phase Compiler

Let us turn to the second multi-phase compiler architecture and see what changes are necessary in the CHAM model we developed for the first architecture. The primary difference that we must account for in the second architecture is that the processing elements can act incrementally on their data and in parallel. This is made possible by the shared repository (see Figure 1). Intuitively, we know that a token is derived from one or more characters, that a phrase is derived from one or more tokens, and so on. Moreover, we would like a given processing element to work on as much or as little of its available input data as it would like, and to do so independently of other processing elements. In essence, then, the repository should behave as an infinite capacity buffer with respect to the input/output behavior of each processing element.

To begin the specification, we must introduce new elements to represent the use of the repository. In particular, we add a repository element and two new operations, ir and or, whose roles are analogous to the i and o operations already introduced, but apply to the repository rather than to the processing elements. In addition, we enrich the structure of the molecules by introducing an infix operator "$\|$" to syntactically represent a complexly composed molecule that can be broken down into parallel subcomponents, thus allowing multiple reactions to occur simultaneously. In more familiar terms, "$\|$" can be intuitively interpreted as a parallel operator.

Again, we need a syntax of molecules for the architecture. We can formulate it by simply augmenting Σ_{seq}. Let us call the new syntax Σ_{con} and define it as follows.

$$
\begin{aligned}
M &::= P \mid C \mid M \diamond M \mid M \parallel M \\
P &::= \mathsf{text} \mid \mathsf{lexer} \mid \mathsf{parser} \mid \mathsf{semantor} \mid \mathsf{optimizer} \mid \mathsf{generator} \\
D &::= \mathsf{char} \mid \mathsf{tok} \mid \mathsf{phr} \mid \mathsf{cophr} \mid \mathsf{obj} \\
C &::= i(D) \mid o(D) \mid ir(D) \mid or(D) \mid \mathsf{repository}
\end{aligned}
$$

Comparing Σ_{seq} and Σ_{con} we see that only the terms associated with the connecting elements C and the highest-level molecule syntax generator M are different.

Next, we need a solution to represent the initial configuration. Let us call it S_1'.

$$
\begin{aligned}
S_1' = \ & o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\
& i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\
& i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator}, \\
& ir(\mathsf{tok}) \diamond \mathsf{repository} \parallel ir(\mathsf{phr}) \diamond \mathsf{repository} \parallel ir(\mathsf{cophr}) \diamond \mathsf{repository}
\end{aligned}
$$

The solution contains a molecule that represents concurrent access to the repository at the granularity of the three kinds of data elements stored in the repository.

To complete the specification, we define a new set of transformation rules.[1]

$$
\begin{aligned}
T_4 &\equiv i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ o(\mathsf{char}) \diamond \mathsf{text} \longrightarrow o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}),\ \mathsf{text} \diamond o(\mathsf{char}) \\
T_5 &\equiv ir(d) \diamond \mathsf{repository},\ o(d) \diamond m \longrightarrow ir(d) \diamond \mathsf{repository},\ m \diamond o(d),\ or(d) \diamond \mathsf{repository} \\
T_6 &\equiv i(d) \diamond m,\ or(d) \diamond \mathsf{repository} \longrightarrow m \diamond i(d),\ or(d) \diamond \mathsf{repository} \\
T_7 &\equiv i(d) \diamond m,\ or(d) \diamond \mathsf{repository} \longrightarrow m \diamond i(d) \\
T_8 &\equiv m_1 \parallel m_2 \parallel \cdots \parallel m_n \longrightarrow m_1,\ m_2, \ldots, m_n \\
T_9 &\equiv p \diamond i(d) \diamond o(d_1) \longrightarrow i(d) \diamond o(d_1) \diamond p \\
T_{10} &\equiv m \diamond i(d) \longrightarrow i(d) \diamond m \\
T_{11} &\equiv \mathsf{text} \diamond o(\mathsf{char}) \longrightarrow o(\mathsf{char}) \diamond \mathsf{text} \\
T_{12} &\equiv \mathsf{text} \diamond o(\mathsf{char}) \longrightarrow \mathsf{text}
\end{aligned}
$$

where $m, m_1, \ldots, m_n \in M$, $d, d_1 \in D$, and $p \in P$. T_4 is a specialization of the generic reaction rule T_1 from the sequential compiler architecture that describes the input of text to the lexer. T_5 through T_7 are reaction rules that generically describe the communication of data elements between the repository and the processing elements; flow of data from a processing element into the repository is described by T_5, while T_6 and T_7 describe the reverse. T_8 through T_{11} are all heating rules: T_8 breaks apart a complex molecule into its (parallel) components, which can then participate in (parallel) reactions;[2] T_9 reactivates an inert processing element; T_{10} reactivates the input port for a processing element before that processing element performs output; and T_{11} makes more of the source text available for compilation, thereby continuing the whole compilation process for a given source text. Finally, T_{12} is a reaction rule that terminates character input, indicating that the source text has been exhausted. The effect of T_{12} is to eventually cause all processing of a particular source text to stop; the proof of this appears in Section 5. A rule that is missing is one analogous to rule T_2 of the sequential architecture, namely a heating rule that serves to "reset" the compiler for operation on a new source text; such a rule is easily constructed.

There are several important things to notice about this specification, particularly in regard to the interaction of the processing elements and the repository. First, the repository molecule in the initial solution S_1' does not contain any output ports for data elements. Instead, the output ports become available through T_5 once some input to the repository has occurred. Second, for a given kind of data element, there are separate repository molecules for input and for output to reflect the independence of those operations. This is not true of the processing elements, where we assume that a processing element always performs some sequence of inputs, followed by a single output, followed

[1] Although we continue the sequential numbering of transformation rules, T_1 through T_3 do not apply to this second CHAM.

[2] Why provide a special parallel operator if the components simply become "ordinary" molecules in the solution anyway? Because the operator serves to indicate in a syntactically convenient way the fact that they are indeed related components of a larger molecule that can act in parallel.

by another sequence of inputs, and so on. In essence, we are modeling the input/output behavior of processing elements as $[I^+O]^*$ and that of the repository as $[I^+O^+]^*$. Third, because input to the repository is incremental, T_5 maintains the viability of an input port after a reaction; T_6 similarly maintains the viability of an output port. T_7, on the other hand, indicates the exhaustion of a given kind of data element in the repository by eliminating the output port from a solution. Only an appropriate T_5 reaction can create the possibility of further output activity. Finally, T_{10} describes the incremental nature of input to a processing element, by allowing multiple inputs before an output, where the potential for multiple inputs arises from multiple T_{10} reactions.

Again, let us trace through some applications of the transformation rules to see how our formulation captures the behavioral aspects of the architecture. The initial solution S'_1 admits to two possible reactions and, since they do not conflict, they can occur in parallel. One reaction is T_4, the input of text by the lexer. The other is T_8, the decomposition of the repository into parallel reactants. Let us assume that both reactions occur and result in solution S'_2.

$$S'_1 \xrightarrow{T_{4,8}} S'_2, \text{ where}$$

$$\begin{aligned} S'_2 = {} & \mathsf{text} \diamond o(\mathsf{char}),\ o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}),\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator}, \\ & ir(\mathsf{tok}) \diamond \mathsf{repository},\ ir(\mathsf{phr}) \diamond \mathsf{repository},\ ir(\mathsf{cophr}) \diamond \mathsf{repository} \end{aligned}$$

Given solution S'_2, we can start applying rule T_5, which expresses the synchronization between the repository and a processing element inserting data of a particular kind into the repository. But we could apply rule T_{11} as well, thus modeling the fact that new characters are available for input. Let us assume that both reactions occur and result in solution S'_3.

$$S'_2 \xrightarrow{T_{5,11}} S'_3, \text{ where}$$

$$\begin{aligned} S'_3 = {} & o(\mathsf{char}) \diamond \mathsf{text},\ \mathsf{lexer} \diamond i(\mathsf{char}) \diamond o(\mathsf{tok}),\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator}, \\ & ir(\mathsf{tok}) \diamond \mathsf{repository},\ ir(\mathsf{phr}) \diamond \mathsf{repository},\ ir(\mathsf{cophr}) \diamond \mathsf{repository}, \\ & or(\mathsf{tok}) \diamond \mathsf{repository} \end{aligned}$$

The solution now contains molecules that can react according to rule T_9 and rules T_6 or T_7. T_9 heats the lexer molecule, while either of T_6 and T_7 provides tokens from the repository to the parser. The more interesting of the three at this point is the subsolution reaction of T_6, which, unlike T_7, does not exhaust the available data elements.

$$i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ or(\mathsf{tok}) \diamond \mathsf{repository} \xrightarrow{T_6} o(\mathsf{phr}) \diamond \mathsf{parser} \diamond i(\mathsf{tok}),\ or(\mathsf{tok}) \diamond \mathsf{repository}$$

From this state, rule T_{10} can be applied to reactivate the input port of the parser and, thus, allow the parser to extract more tokens from the repository. In fact, this T_6/T_{10} cycle can continue unabated until a reaction involving T_7, instead of T_6, occurs or until a reaction involving T_5 occurs to indicate that the parser has placed one or more phrases into the repository. This would still give rise to numerous potential reactions all starting from the same solution. As an extreme example, let us take solution S'_i.

$$\begin{aligned}S'_i = {} & o(\mathsf{char}) \diamond \mathsf{text},\ i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser}, \\ & \quad i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor},\ i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer}, \\ & \quad\quad i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator}, \\ & \quad\quad\quad ir(\mathsf{tok}) \diamond \mathsf{repository},\ ir(\mathsf{phr}) \diamond \mathsf{repository},\ ir(\mathsf{cophr}) \diamond \mathsf{repository}, \\ & \quad\quad\quad\quad or(\mathsf{tok}) \diamond \mathsf{repository},\ or(\mathsf{phr}) \diamond \mathsf{repository},\ or(\mathsf{cophr}) \diamond \mathsf{repository}\end{aligned}$$

Many different rules can be applied simultaneously to S'_i involving all the processing elements in parallel.

To summarize, the specification of the concurrent, shared repository multi-phase compiler architecture is composed of molecular syntax Σ_{con}, initial solution S'_1, and transformation rules T_4 through T_{12}.

4.3 Adding Modularity to an Architecture

The previous descriptions of the two compiler architectures give a simple, flat structure to the elements. Realistically, architectures of any serious complexity will require organization into a richer structure. We now demonstrate the power of the CHAM model in describing a modular structure for an architecture. In particular, we make use of the membrane and airlock constructs discussed in Section 3 to define *modules* and *interfaces* [13] for the sequential architecture. The use of the membrane construct for this purpose is rather natural, since it is the construct expressly provided by the CHAM to abstractly treat solutions as single molecules.

Figure 2 depicts a reconfiguration of the sequential architecture to include two modules, a *front end* and a *back end*. The front end of a compiler is responsible for language-dependent lexical, syntactic, and semantic analyses, while the back end is responsible for machine-dependent code generation optionally preceded by optimization. In practice, this is a very common structure for compilers, since it encourages both the reuse of front-end modules for different target machines and the reuse of back-end modules for different languages.

Notice in Figure 2 that the front- and back-end modules are acting as individual, yet complex, entities whose external behavior is simply a sequential interaction through the exchange of correlated phrases. The internal behavior of one module is hidden from the other, which allows the elements inside the modules to evolve independently. Of course, in this architecture, that evolution is also sequential.

To formalize this structure, we use membranes to encapsulate molecules into solutions that, in turn, can be considered single molecules. Note that it is not necessary to alter Σ_{seq}, since

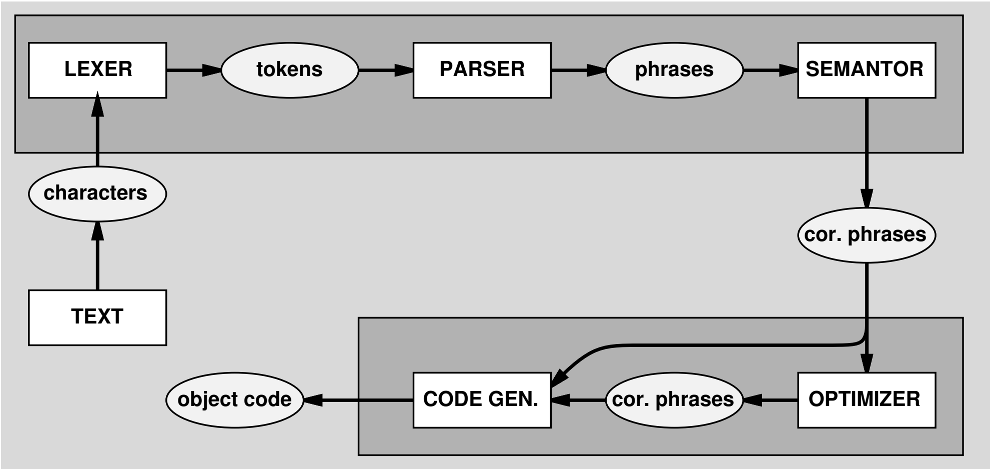


Figure 2: Sequential Multi-Phase Compiler Architecture of Figure 1a Reconfigured with Front- and Back-end Modules.

membranes (and airlocks) are part of the basic CHAM definitions. Let S_1'' be the initial solution for the modular version of the sequential compiler architecture, defined as follows.

$$S_1'' = o(\mathsf{char}) \diamond \mathsf{text},\ S_1^{fe},\ S_1^{be}$$

where S_1^{fe} is the initial solution for the front-end and S_1^{be} is the initial solution for the back-end.

$$S_1^{fe} = \{\!|\, i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer},\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} \,|\!\}$$

$$S_1^{be} = \{\!|\, i(\mathsf{cophr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{optimizer},\ i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator} \,|\!\}$$

In this simple way we have expressed the encapsulation of the processing elements into modules.

The task now is to define the interaction between the modules and their external environment. First, however, notice that we can build upon the specification for the flat sequential architecture given in Section 4.1, since the processing, data, and connecting elements remain the same, as does the basic sequential behavior of the architecture defined by the transformation rule T_1. The problem we face is that some of the molecules that freely interact in the flat architecture are now isolated from one another in the modular architecture. For example, processing cannot begin until there is a reaction between $o(\mathsf{char}) \diamond \mathsf{text}$ and $i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer}$, but the membrane blocks this reaction. What we need to do is to make the membranes semi-permeable to allow the appropriate reactions to take place. In other words, we need to define an interface for the modules.

Consider the front-end module, which uses characters as input from the environment and produces correlated phrases as output. Stated in terms of Σ_{seq}, the front-end module should offer

$i(\mathsf{char})$, which is the input port associated with the lexer molecule, and $o(\mathsf{cophr})$, which is the output port associated with the semantor molecule. This requires a two step process. The first step makes use of the Airlock Law to extract a molecule of interest out of the membrane. The second step associates the communication capability of that molecule with the module as a whole. We take the lexer as our example. Applying the Airlock Law to S_1^{fe} we can obtain the following solution.

$$\begin{aligned} S_2^{fe} = \{\!|\, & i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer} \\ & \lhd \{\!|\, i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} |\!\}\, |\!\} \end{aligned}$$

Unfortunately, the solution still does not permit interaction between the lexer and the text. For that, we must introduce a new transformation rule T_{13}.

$$T_{13} \equiv \{\!|\, m \diamond m_1 \lhd \{\!|\, m_2, \ldots, m_k |\!\}\, |\!\} \longleftrightarrow m \diamond \{\!|\, m \diamond m_1 \lhd \{\!|\, m_2, \ldots, m_k |\!\}\, |\!\}$$

Applying T_{13} to S_2^{fe}, we obtain the following.

$$S_2^{fe} \xrightarrow{T_{13}} S_3^{fe}, \text{ where}$$

$$\begin{aligned} S_2^{fe} = i(\mathsf{char}) \diamond \{\!|\, & i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer} \\ & \lhd \{\!|\, i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} |\!\}\, |\!\} \end{aligned}$$

The solution representing the front-end module has now reached a state where the appropriate communication is possible.

We can continue our trace to see what happens during and after the communication between the lexer and the text. In particular, S_2^{fe} can react with the text molecule using the familiar reaction rule T_1, resulting in the following solution.

$$\mathsf{o(char)} \diamond \mathsf{text},\ S_3^{fe} \xrightarrow{T_1} \mathsf{text} \diamond o(\mathsf{char}),\ S_4^{fe}, \text{ where}$$

$$\begin{aligned} S_4^{fe} = \{\!|\, & i(\mathsf{char}) \diamond o(\mathsf{tok}) \diamond \mathsf{lexer} \\ & \lhd \{\!|\, i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} |\!\}\, |\!\} \diamond i(\mathsf{char}) \end{aligned}$$

At this point the appropriate reaction has taken place, but that fact is not reflected in the internal state of the membrane. We introduce one more transformation rule for this purpose.

$$T_{14} \equiv \{\!|\ m.m_1 \lhd \{\!|\, m_2, \ldots, m_k |\!\}\, |\!\} \diamond m \longrightarrow \{\!|\, m_1 \diamond m \lhd \{\!|\, m_2, \ldots, m_k |\!\}\, |\!\}$$

Applying T_{14} to S_4^{fe}, we obtain the following.

$$S_4^{fe} \xrightarrow{T_{14}} S_5^{fe}, \text{ where}$$

$$\begin{aligned} S_5^{fe} = \{\!|\, & o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}) \\ & \lhd \{\!|\, i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor} |\!\}\, |\!\} \end{aligned}$$

Finally, the Airlock Law is used to reabsorb the lexer back into the main solution so that the processing can continue.

$$S_6^{fe} = \{\!|o(\mathsf{tok}) \diamond \mathsf{lexer} \diamond i(\mathsf{char}),\ i(\mathsf{tok}) \diamond o(\mathsf{phr}) \diamond \mathsf{parser},\ i(\mathsf{phr}) \diamond o(\mathsf{cophr}) \diamond \mathsf{semantor}|\!\}$$

As in the flat sequential architecture, rule T_1 continues that processing until the semantor is in the state where it is ready to offer correlated phrases to the environment. Exactly as we did for the lexer and its input, we can do for the semantor and its output. For brevity, that derivation is not shown.

We should point out that there is one important difference between the standard notion of module interface and the one embodied in this example. In particular, we use generic rules that allow all molecules within a membrane to be offered to the environment, not just a select few. This works here because the reaction rule T_1 together with the initial solution S_1'' fix the actual interactions that can legally take place. Moreover, as mentioned in Section 3, a CHAM searches for reactions to apply and, in the case of the reversible Airlock Law, can cause a molecule to be reabsorbed into the membrane if no reaction is possible.

Despite the correctness of this overly liberal view of interface, in general it might be desirable to express precisely the interface of a module. This can easily be done in our formalism through specialization of the rules to the interface requirements of a given module. The idiom for this is to define a set to contain the exact subset of elements that should be visible through the interface and then define rules derived from the generic rules that are restricted to operate on this set.

Summarizing our addition of modularity to the sequential compiler architecture, we simply begin with the specification of the flat sequential architecture, alter the initial solution to reflect the presence of membranes, and add two rules to reflect the need to move information across interfaces. In fact, the new rules are concerned only with the static structure of the system and not with its dynamic behavior, which remains unchanged.

Finally, to conclude our discussion of modularity, we consider how the various formulations we have presented might be combined and recombined in interesting new ways. This is a natural consequence of using modules, since we should be able to replace the internal behavior of a module without disturbing that module's users. Moreover, we should be able to reuse the results of previous specification activities.

As an example, consider how we might introduce a degree of concurrency into the front end of the sequential compiler architecture. To do this, we simply add a suitable molecule to the initial solution for the front-end membrane S_1^{fe}.

$$S_1^{fe'} = S_1^{fe} \uplus ir(\mathsf{tok}) \diamond \mathsf{repository} \parallel ir(\mathsf{phr}) \diamond \mathsf{repository}$$

The additional molecule is trivially derived from the one developed for the concurrent, shared repository compiler CHAM and represents an internal repository for the tokens and phrases shared by the lexer, parser, and semantor. To make use of this new molecule, we also need to add the appropriate rules from that CHAM, namely T_5 through T_{10}. With these simple additions borrowed from a previously established specification, we have easily introduced a small degree of concurrency into the architecture that is bounded only by the capacity of the back-end module to process correlated phrases.

Another, possibly simpler, way to introduce concurrency into the sequential compiler architecture that also takes advantage of the module structure and previously developed specifications is to allow greater independence in the behaviors of the front- and back-ends. In particular, we can simply add two laws from the concurrent, shared repository CHAM, namely T_9 and T_{11}, and change the behavior into more of a pipeline software architecture [1]. Concurrency arises at the granularity of the modules, since these two heating rules allow the front end to begin processing a new source text at the same time as the back end is still processing the correlated phrases associated with the previous source text. Thus, the rules permit the iterative behavior of the modules to become evident in the architecture.

5 Analyzing CHAM-specified Architectures

In this section we demonstrate the suitability of our approach in the analysis of software architectures. Basically, we can carry out two kinds of useful analysis. The first is a common technique in which a semi-formal, yet rigorous, analysis is performed on specifications by simply reflecting upon them and upon the process that led to their formulation. A second kind of analysis is a truly formal one in which we prove properties by strict mathematical reasoning.

The first kind of analysis is possible under our approach due to the high level of abstraction and conciseness of CHAM descriptions. For example, the CHAM descriptions of the previous section consist of a syntax definition, a set of transformation rules, and an initial solution that occupies less than a page each of text. On the other hand, when a formal analysis is required, the mathematical foundation of CHAM specifications becomes evident. In particular, molecules are defined in terms of an algebraic initial structure that fosters structural inductive proofs. Furthermore, the transformation rules are defined as standard rewriting rules, thus allowing for the application of well-known results and techniques from the field of term rewriting systems [7, 10]. Below, we illustrate both kinds of analysis applied to the specifications of the multi-phase compiler architecture given in the previous section.

Let us begin by performing a semi-formal comparison between the specifications of the two flat multi-phase compiler architectures given in sections 4.1 and 4.2 to see what similarities and differences in the architectures can be found. The first thing we can see is that our specifications make use of exactly the same set of processing and data elements. Where they differ is in the way those elements are connected, which is clearly reflected in the addition of connecting elements and an alteration to the set of transformation rules found in the second CHAM.

Another observation is that in the second architecture we must model the iterative behavior of the individual processing elements in order to obtain the desired concurrent behavior of the whole system. This is in contrast to the first architecture, where the iterative behavior is of the global system (see rule T_2) and not strictly necessary for the specification of the sequential architecture. This is quite interesting, since it shows that a functionality of the processing elements that could be kept hidden by those elements in the first architecture had to be made explicit in the second architecture as part of its correct description.

A third observation we can make is that in the second architecture the connections are *point-to-point* communications between each processing element and the shared repository. Further, this is achieved by defining private communication channels between each processing element and the repository so that they can be accessed concurrently.

We are also able to see that, among all the possible reactions performable by the second architecture, there exists one that gives exactly the same sequence of solutions as the first architecture under a suitable abstraction mechanism. Of course, this is something that one reasonably expects.

This last, semi-formal observation can in fact be quite easily verified formally by proving the following result.

> *Proposition.* Let $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n$ be a sequence of rule derivations on the sequential compiler CHAM starting from the initial solution S_1. Then there exists a derivation $S'_1 \rightarrow S'_2 \rightarrow \cdots \rightarrow S'_m$ in the concurrent, shared repository CHAM starting from the initial solution S'_1, such that $S_1 \subset S'_1, \ldots, S_i \subset S'_j, S_{i+1} \subset S'_{j+k}, \ldots, S_n \subset S'_m$.
>
> *Proof.* The proof is by induction on the length n in the rule derivation of the sequential CHAM.
>
> For $n = 1$, the proof is immediate from the definitions of S_1 and S'_1. Let us now assume the proposition is true for n and prove it is true for $n+1$. In other words, we have the sequence $S_1 \rightarrow S_2 \rightarrow \cdots \rightarrow S_n \rightarrow S_{n+1}$ and consider only $S_n \rightarrow S_{n+1}$. The transformation can be performed either by applying T_1 or T_2. First consider T_1. $S_n \xrightarrow{T_1} S_{n+1}$ means that in S_n there is a pair of molecules that can react. Let $i(d) \diamond m$ and $o(d) \diamond m_1$ be the two molecules. By the inductive hypothesis we know that there exists a solution S'_m in the second CHAM such that $S_n \subset S'_m$. Therefore in S'_m there exists the same pair of molecules as in S_n. In order to prove the proposition, we must show that in S'_m it is possible either to apply T_4 or to apply T_5 and T_6 or T_7 on the same molecules that are reacting in S_n. This amounts to showing that from S'_m it is possible to reach a solution in which the molecule $ir(d) \diamond \mathsf{repository}$ exists, the application of T_4 is in fact straightforward. We know that in S'_1 there exists a repository molecule and that there are two possible formats of the repository molecule in S'_m. The first is that the repository molecule is still in its initial format, that is, as a parallel molecule $m_1 \parallel m_2 \parallel \cdots \parallel m_n$. In this case it is enough to perform a transformation on S'_m using T_8, which results in a solution S'_{m+1} in which it is possible to apply T_5 and then T_6 or T_7. The second case is when S'_m is already hot with respect to the repository molecule, which leaves the solution in the same condition as S'_{m+1}.
>
> The case for $S_n \xrightarrow{T_2} S_{n+1}$ is analogous and for brevity is not shown. □

We have now proven that the sequential behavior of the first CHAM can be simulated by one of the possible behaviors of the second CHAM.

The previous proposition is concerned with a formal comparison of software architectures. We can also formally prove properties with regard to a specific architecture.

A property that we may wish to prove about the concurrent, shared repository architecture is that it allows for infinite derivations from the initial solution if and only if data are infinite, that is either the input text is itself infinite or the data in the repository are not finitely consumed by an application of rule T_7. The utility of such a property should be obvious. First we introduce a definition.

Definition. A reaction derivation $S_1 \to S_2 \to \cdots \to S_n$ is *normalizing* if S_n is inert.

The definition means that a given derivation terminates, since the fact that S_n is inert means that there is no other reaction rule that can be applied to it. Thus, a normalizing derivation is a derivation that *terminates.* Now we can state the property that we wish to prove.

Proposition. Let S_1 be the initial solution of the concurrent, shared repository CHAM and $\delta : S_1 \to S_2 \to \cdots \to S_n$ be a derivation from S_1. Then any derivation δ' from S_n is normalizing if and only if there exists in δ a solution S_i such that $S_i \xrightarrow{T_{12}} S_{i+1}$, S_n does not contain occurrences of the molecule $or(d) \diamond \mathsf{repository}$ for any $d \in D$, and rule T_5 cannot react in S_n.

Proof. Let us first consider whether any normalizing derivation δ' from S_n implies the existence of $S_i \xrightarrow{T_{12}} S_{i+1}$ in δ and that S_n does not contain, or can create, occurrences of the molecule $or(d) \diamond \mathsf{repository}$ for any $d \in D$. If $i = 0$ (i.e., δ is normalizing), then this means that S_n is inert—that is, no more reactions can take place. Looking at the rules it is easy to see that the application of the heating rules T_9 and T_{11} can only be prevented if a data element is no longer available, and that this is achieved by applying T_{12}, which is the only rule permanently removing data. On the other hand, the potentially infinite application of T_{10} is prevented by the fact that no occurrence of the molecule $or(d) \diamond \mathsf{repository}$ exists in S_n or can be created. If $i \neq 0$ then S_n is not inert and some further reactions can be performed. These cannot involve T_5, however, because otherwise a non-terminating derivation can be easily built, thus contradicting the hypothesis. The same argument applies to T_{12} because if it can still be applied—that is, it is not contained in δ—then a non-terminating derivation can be built.

We now consider the reverse condition. Let us assume that a step with T_{12} has been performed in δ and that S_n does not contain any $or(d) \diamond \mathsf{repository}$ molecules. We then have to show that any δ' is normalizing. $S_i \xrightarrow{T_{12}} S_{i+1}$ means that in S_{i+1} the text molecule has been rendered inert. Now, this means that in S_n the text molecule is present and no $or(d) \diamond \mathsf{repository}$ molecules exist. Depending on the rest of the solution, there exists a maximum number of reaction steps that can be further performed on S_n. In fact, rules T_4, T_8, T_{11}, and T_{12} cannot be applied anymore. Rule T_5 and rules T_6 and T_7 cannot be applied by hypothesis. Therefore, only rules T_9 and T_{10} can be applied, but without any further consequent reaction, since no occurrence of $or(d) \diamond \mathsf{repository}$ exists or can be created. □

Another interesting and useful property to prove is that the modular version of the sequential compiler architecture actually contains modules that can communicate. In other words, we would like to show the correctness of the front- and back-end module interfaces. This can be formally proven with the following series of propositions.

> *Proposition.* Let S_1 be the initial solution of the modular sequential compiler CHAM. Then there exists a derivation $S_1 \to S_2 \to \cdots \to S_i \to S_{i+1} \cdots \to S_n$ such that
>
> $$S_i \quad = o(d) \diamond \{\!|m_i \triangleleft \{\!|M_i|\!\}|\!\},\ i(d) \diamond \{\!|m'_i \triangleleft \{\!|M'_i|\!\}|\!\}$$
>
> $$S_{i+1} = \{\!|m_i \diamond o(d) \triangleleft \{\!|M_i|\!\}|\!\},\ \{\!|m'_i \diamond i(d) \triangleleft \{\!|M'_i|\!\}|\!\}$$
>
> *Proof.* We must show that a communication between the front end and back end is possible. The proof consists of building such a derivation starting from the initial solution S_1.
>
> From S_1 we first apply all the possible reactions inside the front end, which amounts to applying T_1 until the molecule $o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr})$ exists in the front end. We then apply T_{13} and T_{14} to both the front and back ends on the molecules $o(\mathsf{cophr}) \diamond \mathsf{semantor} \diamond i(\mathsf{phr})$ and $i(\mathsf{cophr}) \diamond o(\mathsf{obj}) \diamond \mathsf{generator}$, respectively. This leads to the desired solution. □

While useful, this result can be further strengthened, as follows.

> *Proposition.* Let S_1 be the initial solution of the modular sequential compiler CHAM. Then there exists a derivation $S_1 \to S_2 \to \cdots \to S_n$ such that molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$ is in S_n.
>
> *Proof.* We must simply show that it is possible to build such a derivation. This is straightforward using the two rules T_{13} and T_{14} to reach a solution in which T_1 can be applied. This can be repeated until the $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$ molecule is produced. □

We have now shown that text will eventually be processed. But, again, we can prove an even stronger result that shows that any derivation that terminates will terminate properly by containing the molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$.

Before we can state and prove the final proposition, however, we must examine the modular sequential CHAM a bit closer. First, notice that it makes use of two reversible transformation rules, namely the Airlock Law and rule T_{13}. These two rules can cycle infinitely with no real progress and, moreover, have no real effect on the behavior of the system. Therefore, in order to reach inert solutions, we must restrict consideration to derivations that do make progress. This means that we consider those derivations that can partially cycle, but then eventually move to a new solution by applying a reaction step such as T_1. Second, notice that we are not interested in derivations containing reaction steps with T_2, because the application condition for T_2 is exactly the subsolution we are trying to show to be reachable from S_1, that is, a solution containing the molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$.

Proposition. Let S_1 be the initial solution of the modular sequential compiler CHAM. We restrict consideration to derivations that contain only T_1, T_{13}, and T_{14}. Then any normalizing derivation $S_1 \to S_2 \to \cdots \to S_n$ is a derivation containing molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$ in S_n.

Proof. Let $S_1 \to S_2 \to \cdots \to S_n$ be the normalizing derivation. We show that in S_n there exists the molecule $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$.

By looking at the reaction rules, we can see that the only rules that can be applied an infinite number of times are the Airlock Law and T_{13}. From the initial solution S_1 only these two rules can be applied and, since the derivation is normalizing, there are only a finite number of possible solutions that can be obtained before a step with T_1 is performed. This process ends only when a solution is reached that is inert with respect to T_1 and this can only happen when $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$ is generated. After the last T_1 reaction step, there can be other reaction steps with the Airlock Law and T_{14} that do not modify the molecules in the solutions but only modify their membrane structure. Thus, all of those possible solutions must contain $o(\mathsf{obj}) \diamond \mathsf{generator} \diamond i(\mathsf{cophr})$. By our hypothesis, these steps are finite and the proposition follows. □

The result proved here is quite interesting because it tells us that if we consider the modular sequential CHAM, then the interesting derivations are exactly the ones that modify the solutions in the same manner as the flat sequential CHAM behavior—that is, from a dynamic point of view, the modular CHAM behaves exactly like the flat one. While the modular version might spend some more time in finding the right interfaces to communicate, once the communication takes place, the behavior is same as in the flat CHAM.

The various analyses that we carry out above demonstrates that, despite its apparent simplicity, the CHAM model allows us to specify and analyze a number of critical properties at the level of the system's software architecture and, moreover, to formally compare corresponding properties in different architectures.

6 Conclusion

In this paper we have presented an approach to the formal description of software architectures. It is based on an operational framework, which for some aspects of architectural description seems to us to be more easily accessible than other frameworks. Our approach is a first attempt at using an intriguing new model for describing architectures, the Chemical Abstract Machine. The CHAM is a reasonably simple model, yet very expressive. CHAM descriptions, because of their formality, lend themselves both to correctness analyses within a single specification as well as to comparative analyses across different specifications. Finally, CHAM descriptions are inherently modular, allowing the refinement of a description to proceed by the addition of molecules and transformation rules. Similarly, families of architectures, or *architectural styles* [14] can be handled by simply defining general molecules and rules that get refined by definitions of additional specialized molecules and rules. This is essentially what happens in the multi-phase compiler example presented in Section 4.

In addition to our investigation of the CHAM model, research into formal architectural description is causing an examination of various other semantic models as suitable foundations. A considerable amount of work has involved the Z specification language [16]. In contrast to the operational CHAM model, Z is a model-based, set-theoretic formalism. One weakness that we perceive in using Z for architectural description is that it is not well suited to the specification of dynamic or behavioral aspects of an architecture, including concurrency and nondeterminism, since it relies on a variety-based denotational semantics [17]. On the other hand, Z has proven to be quite useful for the description and analysis of certain static properties of software architectures [1, 8].

Another semantic framework that is being used in formal architectural description, one that is better able to describe dynamic properties of architectures, is CSP [9]. In particular, the Wright architectural description language uses a subset of CSP to specify connecting elements [2]. Given a grounding in CSP, analysis of Wright specifications can reveal certain important correctness properties, such as freedom from deadlock.

What is generally true about all these approaches, however, is that there is not yet enough experience in using them for architectural description to be able to draw definitive conclusions about which one is superior. In fact, the question of which is superior is not even appropriate. A better question seeks to identify those aspects of architectural description for which one or the other is better suited and to understand how the approaches might be productively combined.

One important consideration in the development or even selection of a descriptive approach is that it should allow an appropriate level of abstraction in a formal specification. An approach that forces the inclusion of inconsequential details too early in the development process, that is not amenable to incremental inclusion of important details, or that requires elaborate formal machinery to express simple concepts, is in our estimation inappropriate for specification of software architectures. Of course, judging an approach on these grounds is clearly subjective and not unrelated to the question of where architecture ends and low-level design begins. Nevertheless, we believe that as experience is gained in using a variety of approaches on a multiplicity of architectures, a consensus will emerge. This consensus will serve to define an appropriate place for software architecture description within the spectrum of system specifications at the same time as it will encourage the appropriate level of abstraction in the natural expression of architectural descriptions.

We reiterate the point, however, that no one technique will be appropriate for all the varied, and sometimes conflicting, requirements of software architecture description and analysis. Formal techniques, such as the one introduced in this paper or developed by others, must be used in conjunction with other, formal and informal techniques. We take as one simple bit of evidence the fact that people always use informal pictures to supplement otherwise formal descriptions of systems. The purpose of this paper is to suggest that the CHAM model might be one useful tool in the software architect's chest of useful tools.

Acknowledgments

We thank Dewayne Perry for sharing with us his ideas on the principles of software architecture description. We also thank David Garlan, Dewayne Perry, and the anonymous reviewers for their helpful comments on earlier drafts of this paper.

REFERENCES

[1] R. Allen and D. Garlan. A Formal Approach to Software Architectures. In *Proceedings of the IFIP Congress*. Elsevier, September 1992.

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

[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] N. Dershowitz and J.-P. Jouannaud. Rewrite Systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics*, pages 243–320. North Holland, Amsterdam, 1990.

[8] D. Garlan and D. Notkin. Formalizing Design Spaces: Implicit Invocation Mechanisms. In *Proceedings of VDM '91: Formal Software Development Methods*, October 1991.

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

[10] J.W. Klop and R.C. de Vrijer. *Term Rewriting Systems*. Cambridge University Press, 1993.

[11] P. Landin. The Mechanical Evaluation of Expressions. *Computer Journal*, 6:308–320, 1964.

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

[13] D.L. Parnas. On the Criteria to be Used in Decomposing Systems into Modules. *Communications of the ACM*, 15(12):1053–1058, December 1972.

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

[15] G. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN–19, University of Aarhus, 1981.

[16] J.M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, 1989.

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