# Analyzing and Refining an Architectural Style

Paolo Ciancarini and Cecilia Mascolo

Dipartimento di Scienze dell'Informazione,
Università di Bologna,
e-mail: {cianca,mascolo}@cs.unibo.it

**Abstract.** Architectural styles have been introduced in [1] in order to classify and analyze software architectures. In that paper, Z was used as a notation to specify and study architectural styles, however some problems remained open concerning specification and analysis of their behavioral properties. We use a new operational semantics to describe and analyze an architectural style of distributed systems. We introduce three refinements of a "Message Router" style, useful to describe distributed applications like e-mail or news systems; we also formalize and prove some properties of the style and, henceforth, of derived software architectures.

## 1 Introduction

Software architectures are structures that can help in designing and understanding complex software systems. Notations and tools for specifying and analyzing software architectures are currently widely investigated [17, 14, 18].

Abstractly, software architectures can be classified using the concept of "architectural style", introduced in [17] and formalized using the Z notation [1]. Architectural styles are abstractions including entities like components and connectors, and some composition rules. An architectural style is a sort of skeleton which can help in understanding and analyzing concrete software architectures which are instances of such a style. An architectural style is less constrained and less complete than a specific architecture [17]. For instance, the architectural style of *pipes and filters* can be used to study the architecture of a compiler organized as a pipeline of analysis and translation tools.

There are at least three reasons why it is important and useful to systematically study architectural styles:

1. to build a library of styles to be made available for designers, so that they can choose the most appropriate one;
2. to help designers to recognize the need to apply a specific style in a given design situation;
3. to offer designers analysis methods and tools suitable to deal with concrete software architectures, reasoning on and understanding their properties.

However, using the approach described in [1] most behavioral analyses on formal software architectures are difficult, because Z has no operational semantics.

For instance, to compare a sequential pipeline to a parallel pipeline is difficult, because in Z there is no concept of control, either sequential or parallel.

Instead, operational formalisms like the Chemical Abstract Machine [3] have been successfully used in the specification of software architectures to describe and study their behavioral aspects [16].

We show here how an approach which maps the Z notation to a formal operational semantics based on the CHAM [8] allows us to both specify and analyze dynamic properties of an architectural style using tools for both notations.

The example we consider is a style describing a Message Router, a key component of modern distributed systems. A formal specification and design of this kind of systems has to take into consideration their architectural structure. In fact, the Message Router can be abstracted by a communication style that can be formalized and analyzed as in [1].

We present three refinements of the style and study their properties at different levels of detail: note that, in this context, refinement is thought as incremental definition of a specification, as in [10]. We show that the Z formalization helps to give a modular structure to the architectural style, whereas the operational semantics supports the analysis of the behavioral aspects.

The paper has the following structure: Sec.2 contains a short description of the semantics we associate to Z and the logic we use in reasoning on our formal specifications. Sec.3 defines the architectural approach we consider; Sec.4 contains three refined versions of the Message Router style specified and analyzed. Finally, in Sec.5 we describe some conclusions and future work.

## 2 A chemical semantics for Z and its logic

The elementary components of any Z document are State schemas and Operation schemas. This means that we can see a Z specification as a pair: $< S, O >$, where $S$ is the set of the State schemas and $O$ the set of the Operation schemas.

Semantically, a State schema $s$ in $S$ can be seen as the set of all its possible instantiations: the standard Z semantics [19, 6, 5] is declarative and does not offer any formalization for any behavior, be it concurrent or distributed. For this reason, we introduce an operational semantics based on a chemical model.

In the Chemical Abstract Machine (CHAM) model [3] *Molecules*, *Solutions*, and *Rules* are the fundamental elements. A Chemical Abstract Machine is a triple $(G, C, R)$, where $G$ is a grammar, C is a set of configurations (the language generated by the grammar) or molecules, and $R$ is a set of the rules of the form $condition(C) \times \text{bag } C \times \text{bag } C$.

A solution is a multiset of molecules, namely bag $C$. Two rules can fire concurrently if they do not need the same molecules to react on; hence, several rules can progress simultaneously on a solution. If two rules conflict, in the sense that they "consume" the same molecules, one of them is chosen non deterministically to react.

We give a CHAM interpretation of Z specifications which allows us to deal with concurrent dynamics. Intuitively, an instance of a state schema is associated to a solution where, in some way, each variable is a subsolution (in many cases a single molecule). Instead, an operation schema corresponds to a chemical rule where premises and consequences are solutions composed of pre and post conditions of the operation. We now describe the formalization of such an interpretation.

A molecule is a tuple of a name, a type and a value:

$$MOLECULE == NAME \times TYPE \times VALUE$$

A solution is a bag of molecules:

$$SOLUTION == \mathrm{bag}\, MOLECULE$$

and a rule is composed of a conditional part that define the applicability of the rule and of two solutions that indicate molecules to delete and add to the state solution:

$$RULE == CONDITION \times SOLUTION \times SOLUTION$$

We will call the first $SOLUTION$ "pre-solution" and the second "post-solution" in order to avoid ambiguity.

A rule is applicable to a solution if the solution contains molecules that satisfy the conditional part ($CONDITION$) of the rule and molecules that match the presolutions of the rule.

The function $FSem$ associates a schema_instance to a solution:

$$Fsem : SCHEMA\_INSTANCE \rightarrow SOLUTION$$

Every identifier of the schema instance is associated to a subsolution (not necessarily a single molecule): we remark that Z sets and bags are decomposed by this function in several molecules so as to increase potential concurrency of the model.

$Fsem\_op$ associates a rule to an operation schema:

$$Fsem\_op : SCHEMA\_OP \rightarrow RULE$$

$Fsem\_Op$ associate to pre and postcondition of a schema different part of the rule:

- Every schema postcondition requiring the removal of an element from a set or bag is mapped to a presolution of the rule , namely it becomes a molecule to be deleted.
- Every postcondition that specifies the insertion of an element in a set or bag is mapped to a postsolution of the rule (molecules to be added).
- Every schema precondition that defines a membership predicate ($\in$, in) is mapped to a presolution (a removal) and to a postsolution (reinsertion) if the postcondition does not contain an indication of removal of that element.
- Postconditions containing mathematical operators ($+$, $-$,...) on naturals are encoded deleting one molecule and adding the molecule updated.

– Preconditions containing relational operators are encoded as conditions, whereas the molecule corresponding to the variable is deleted and added again, as already described. This conforms to the chemical semantics where conditions can only be stated on local molecules involved in a rule [4].

It is possible to define many other functions to describe, for instance, when rules, namely operation schemas, can fire concurrently: it usually depends on their postconditions.

### 2.1 The logic

We define now an *execution model*, that is a way of abstractly executing a Z specification document, and a Unity-like logic [7] to reason on properties exhibited by such a execution model.

The execution model represents the unfolding of the application of the rules of the operational semantics.

For every state schema $s$ an abstract execution tree can be constructed: every node corresponds to a particular instance of the state schema and from each node several different applicable operation sets can exist, (chosen among all the enabled operations on that node), thus introducing non-determinism in the choice of the operations being in conflict. Each branch corresponds to the application of a group of enabled operations which could be applied without conflicts, as dictated by the Cham model.

It is now possible to formalize a logic to reason on the dynamics of a Z specification.

In order to be able to reason on dynamic properties, like deadlock and starvation, we introduce our logic borrowing a few constructs from the Unity logic. Properties are expressed as predicates related by Unity logic operators; predicates now have chemical semantics and are interpreted as chemical solutions. We can state a predicate $p$ is valid on a state solution $s$ if all the molecules in the chemical interpretation of $p$ are also in $s$.

- $p$ **unless** $q$ says that whenever $p$ is true during the execution, surely either $q$ will become true or $p$ continues to hold. In particular, on the tree: if $p$ is true on some nodes then on their children $q$ is true or $p$ still holds.
- **Stable** is an alias for $p$ **unless** false, that is when $p$ becomes true it will hold forever. On the tree: if $p$ is true on a node it will remain true for the whole subtree of that node.
- **Invariant** $p$ says that $p$ is true forever. That is, for every node of the execution tree $p$ is valid.
- $p$ **ensures** $q$ means that when $p$ becomes true then eventually $q$ will hold and before that moment $p$ is still valid. That is, if $p$ is true on a node $N$, then in each branch through $N$ there is a node $M$ below $N$ where $q$ holds and on nodes between nodes $N$ and $M$ in the path, $p$ holds.
- $p$ **leads_to** $q$ has quite the same meaning as **ensures** except that it does not ensure that $p$ is valid until $q$ becomes true. On the tree: if $p$ is true on a

node $N$, then in each branch through $N$ there is a node $M$ below $N$ where $q$ holds.

A formalization of this interpretation can be found in [8]. Having introduced the meaning of these logic predicates on the model, we can reason and define new dynamic properties on the abstract concurrent interpretation of Z specifications.

## 3 The message router as an architectural style

Architectural properties are especially important in the design of a complex software system. This means that great care must be paid to behavioral aspects of the system being designed. The study of the dynamics of a software architecture is a very important phase in the development process. Other approaches to the specification of software architectures put emphasis on the analysis of system dynamics: see for instance [16] for an example of direct application of CHAM for such a task. However, we believe that a simple CHAM specification *per se* is not sufficient to offer a complete view of the style of a system architecture.

We combine the best features of two approaches: in fact we use Z for studying the static, modular architecture, whereas we use CHAM for analyzing its dynamics. In this way we obtain a coherent methodology useful to analyze all aspects of software architectures.

Let us first shortly recall the architectural ontology described in [1], which considers an abstract syntax useful for classifying architectural styles.

Components are active computational entities of an architecture; they accomplish tasks through internal computation and external communication with the rest of the system (the interaction points are the *ports*) [1].

A *component* is

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Component}} \\
\hline
\mathit{ports} : \mathbb{P}\, \mathit{PORT} \\
\mathit{descr} : \mathit{COMPDESC} \\
\hline
\end{array}$$

Connectors define the communication between components. Each connector provides a way for a collection of ports to come into contact.

A *connector* is defined as:

$$\begin{array}{|l}
\multicolumn{1}{l}{\mathit{Connector}} \\
\hline
\mathit{roles} : \mathbb{P}\, \mathit{ROLE} \\
\mathit{descr} : \mathit{CONNDESC} \\
\hline
\end{array}$$

Finally, a *configuration* attaches connectors to components:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Configuration}} \\
\hline
components : COMPNAME \leftrightarrow Component \\
connectors : CONNNAME \leftrightarrow Connector \\
attachment : RoleInst \leftrightarrow PortInst \\
\hline
\forall cn : CONNNAME;\ r : ROLE \mid (cn, r) \in \mathrm{dom}\, attachment \bullet \\
\quad cn \in \mathrm{dom}\, connectors \land r \in (connectors(cn)).roles \\
\forall cn : COMPNAME;\ p : PORT \mid (cn, p) \in \mathrm{ran}\, attachment \bullet \\
\quad cn \in \mathrm{dom}\, components \land p \in (components(cn)).ports \\
\hline
\end{array}
$$

This is a meta-description useful for defining and classifying architectural styles: for instance, it has been used to describe a pipe/filter style [1].

We apply it to the Message Router. Consider a communication network that connecting $N$ senders to $M$ receivers via a message router. Each sender is connected to one of the input ports of the router, whereas each receiver is connected to one output port.

The specifications of the message router given in [11, 10] use pictures to illustrate the architecture of the system and help in the understanding of its properties. In [10] the specification of an abstract router is especially detailed and six refinements are provided and studied using the Unity language.

In the first version we give here each message consists of a single packet and the router as a black box delivering messages. Messages are actually sequences composed of a sender, a receiver, a number, and a content; the number identifies different messages exchanged between the same pair sender/receiver.

$$MESSAGE == NAME \times NAME \times \mathbb{N} \times DATA$$

These are the structures of the senders and the receivers:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Sender\_Struct}} \\
\hline
Sending : \mathbb{P}\, MESSAGE \\
PermMsg : \mathbb{P}\, MESSAGE \\
Sender : \mathbb{P}\, NAME \\
\hline
\end{array}
$$

*PermMsg* permanently records sent messages.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Receiver\_Struct}} \\
\hline
Receiving : \mathbb{P}\, MESSAGE \\
Receiver : \mathbb{P}\, NAME \\
\hline
\end{array}
$$

The Router is composed of a table that records the numbers of the last sent messages from each sender to each receiver *NumMR*; *Router* stores routed messages.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Router\_Struct}} \\
\hline
NumMR : \mathbb{P}(NAME \times NAME \times \mathbb{N}) \\
Router : \mathbb{P}\, MESSAGE \\
\hline
\end{array}
$$

The *Generic_Router* is composed of:
*Sender_Struct*, *Receiver_Struct*, and *Router_Struct*.

$$\begin{array}{|l}
\hline \textit{Generic_Router} \\
\textit{Sender_Struct} \\
\textit{Receiver_Struct} \\
\textit{Router_Struct} \\
\hline
\end{array}$$

For this style components are *Sender_Struct* and *Receiver_Struct*, whereas the connector is *Router_Struct*. Component ports are *Sending* and *Receiving* variables that record the messages to be sent and the received ones for each sender and receiver; roles are associated with *Router* variables where the routed messages for each pair of sender and receiver are stored. The configuration is associated to the *Generic_Router* schema: ports and roles are attached by messages sent for each pair of sender and receiver.

The meaning function helps in clearly define the structure of the architecture and the points of interaction.

Following [1], we give the formalization of the mapping from the abstract syntax to our specification:

$$\begin{array}{|l}
\mathcal{M}_{Comp} : Component \rightarrow Struct \\
\mathcal{M}_{port} : PortInst \rightarrow MESSAGE \\
\hline
\forall\, c : Component;\ n : COMPNAME;\ st : Struct; \\
\quad s : Sender_Struct \mid c \in \operatorname{dom} \mathcal{M}_{Comp} \land \\
\quad S(s) = st \bullet s.Sending = \mathcal{M}_{port}(\!|\ \{n\} \times c.ports\ |\!) \\
\forall\, c : Component;\ n : COMPNAME;\ st : Struct; \\
\quad r : Receiver_Struct \mid c \in \operatorname{dom} \mathcal{M}_{Comp} \land \\
\quad R(r) = st \bullet r.Receiving = \mathcal{M}_{port}(\!|\ \{n\} \times c.ports\ |\!)
\end{array}$$

where

$$PortInst == COMPNAME \times PORT$$

$$\begin{array}{l}
Struct ::= S\langle\!\langle Sender_Struct \rangle\!\rangle \\
\qquad\quad \mid\ R\langle\!\langle Receiver_Struct \rangle\!\rangle
\end{array}$$

The mapping function $\mathcal{M}$ associates the variables *Sending* and *Receiving* to component ports.

$$\begin{array}{|l}
\mathcal{M}_{Conn} : Connector \rightarrow Router_Struct \\
\mathcal{M}_{role} : RoleInst \rightarrow MESSAGE \\
\hline
\forall\, c : Connector;\ n : CONNNAME;\ r : Router_Struct \mid \\
\quad c \in \operatorname{dom} \mathcal{M}_{Conn} \land r = \mathcal{M}_{Conn}(c) \\
\quad \bullet\ r.Router = \mathcal{M}_{role}(\!|\ \{n\} \times c.roles\ |\!)
\end{array}$$

where

$$RoleInst == CONNNAME \times ROLE$$

The following function associates variable *messages* to a connector role.

$$\mathcal{M}_{Conf} : Configuration \rightarrow Generic\_Router$$

$$\begin{array}{l}
\forall\, cfg : \mathrm{dom}\, \mathcal{M}_{Conf} \bullet (\mathcal{M}_{Conf}.Sender_Struct = \\
\quad \{n : COMPNAME;\ c : Component;\ s : Sender_Struct \mid \\
\quad (n, c) \in cfg.components \wedge s \in \mathcal{M}_{Comp}(c) \\
\quad \wedge\ s.Sending = \mathcal{M}_{port}(\!|\ \{n\} \times c.ports\ |\!) \bullet s\}) \\
\quad \wedge\ Same_definition_for_Receiver_Struct \\
\quad \wedge\ (\mathcal{M}_{Conf}.Router_Struct = \\
\quad \{n : CONNNAME;\ c : Connector;\ s : Router_Struct \mid \\
\quad (n, c) \in cfg.connectors \wedge s \in \mathcal{M}_{Conn}(c) \\
\quad \wedge\ s.Router = \mathcal{M}_{roles}(\!|\ \{n\} \times c.roles\ |\!) \bullet s\})
\end{array}$$

*Send_Message* is the schema describing the operation to transmit a message from a sender to a receiver, provided that such a message is not the first one from that sender to that receiver (a special operation is defined for the first message).

*Send_Message*

$$\Delta Generic\_Router$$

$$\begin{array}{l}
\exists\, s, r : NAME;\ n : \mathbb{N};\ dat : DATA \mid (s, r, n, dat) \in Sending\ \wedge \\
\quad s \in Sender \wedge (s, r, n-1) \in NumMR \bullet \\
\quad Router' = Router \cup \{(s, r, n, dat)\} \\
\quad Sending' = Sending \setminus \{(s, r, n, dat)\}
\end{array}$$

This is the chemical rule which denotes the semantics of such an operation schema.

This is the pre-solution:

$$\begin{array}{l}
(Sending, \mathbb{P}\, MESSAGE, (s, r, n, dat)), \\
(Sender, \mathbb{P}\, NAME, s), \\
(NumMR, \mathbb{P}(NAME \times NAME \times \mathbb{N}), (s, r, n-1))
\end{array}$$

This is the post-solution:

$$\begin{array}{l}
(Router, \mathbb{P}\, MESSAGE, (s, r, n, dat)), \\
(Sender, \mathbb{P}\, NAME, s), \\
(NumMR, \mathbb{P}(NAME \times NAME \times \mathbb{N}), (s, r, n-1))
\end{array}$$

The following operation delivers the message:

*Deliver_Message*

$$\Delta Generic\_Router$$

$$\begin{array}{l}
\exists\, s, r : NAME;\ n : \mathbb{N};\ dat : DATA \mid (s, r, n, dat) \in Router\ \wedge \\
\quad r \in Receiver \wedge (s, r, n-1) \in NumMR \bullet \\
\quad Router' = Router \setminus \{(s, r, n, dat)\}\ \wedge \\
\quad Receiving' = Receiving \cup \{(s, r, n, dat)\} \\
\quad \wedge\ NumMR' = NumMR \setminus \{(s, r, n-1)\} \cup \{(s, r, n)\}
\end{array}$$

This is the chemical rule which denotes the semantics of such an operation schema. This is the pre-solution:

$(Router, \mathbb{P}\, MESSAGE, (s, r, n, dat)),$
$(Receiver, \mathbb{P}\, NAME, r),$
$(NumMR, \mathbb{P}(NAME \times NAME \times \mathbb{N}), (s, r, n-1))$

This is the post-solution:

$(Receiving, \mathbb{P}\, MESSAGE, (s, r, n, dat)),$
$(Receiver, \mathbb{P}\, NAME, r),$
$(NumMR, \mathbb{P}(NAME \times NAME \times \mathbb{N}), (s, r, n))$

We omit the formalization of the initialization operation introducing the senders, the receivers, and the messages to be sent.

### 3.1 Analysis

We consider some classes of properties already introduced in [11]. However, our analysis is quite different; in fact, in [11] properties themselves define the specification of the system, while here they are introduced only to analyze behavioral features of a Z specification document.

We will study the same properties for all refinements, showing how the granularity of detail used for describing the software architecture influences its analysis.

The classes of properties we consider are the following:

1. **Eventual Delivery**: every sent message will eventually be delivered.
2. **Order Preserving**: the order of the messages sent by the same sender to the same receiver will be preserved.
3. **Prefix Invariant**: delivered messages have been messages to be sent.

This is the analysis of the first refinement:

- **Class Eventual Delivery**

  **Theorem 1.** $(s, r, n, dat) \in Sending$ **ensures** $(s, r, n, dat) \in Router$

  That is: a message to be sent will eventually be routed.

  **Theorem 2.** $(s, r, n, dat) \in Router$ **ensures** $(s, r, n, dat) \in Receiving$

  That is: a message routed will eventually be delivered.

  **Theorem 3.** $(s, r, n, dat) \in Sending$ **leads_to** $(s, r, n, dat) \in Receiving$

That is: a message to be sent will eventually be delivered; this theorem in some sense summarizes the previous two theorems.

**Proof of Theorem 1**:

To prove $p$ **ensures** $q$

(where $p$ is $(s, r, n, dat) \in Sending$ and $q$ is $(s, r, n, dat) \in Router$) we must prove $p$ **unless** $q$ and that a set of operations exists, which when it is applied to a state where $p \wedge \neg q$ is valid leads to a state where $q$ holds.

We now prove $p$ **unless** $q$; for every operations set enabled on the solution containing the molecule $(Sending, \mathbb{P}\, MESSAGE, (s, r, n, dat))$ the application has to generate a state where the molecule is not present and $(Router, \mathbb{P}\, MESSAGE, (s, r, n, dat))$ is present or the previous molecule is still in the solution (in fact, this is the semantics of **unless**).

Considering our case study, we remark that applying operations on a different sender or receiver we reach a state where $p$ still holds, while applying operation *Send_Message* with sender $s$ and receiver $r$, we obtain a state where $q$ holds. Then $p$ **unless** $q$ holds.

We now prove the second part of our theorem: given a state where $p \wedge \neg q$ holds, there exists an enabled set of applicable operations, which generates a state where $q$ holds. Such a set is composed of operation *Send_Message* acting on the molecule

$(Sending, \mathbb{P}\, MESSAGE, (s, r, n, dat))$

and other operations not acting on that molecule. Since our CHAM model is *fair*, we can state that the set will eventually be applied. This completes the proof.

Proofs of Theorems 2 and 3 are similar: we omit them.

– **Class Order Preserving**

**Theorem 4.** $(s, r, n, dat) \in Sending$ **unless** $(s, r, n-1, dat1) \in Receiving$

That is: a message will be delivered only after the message previously sent from that sender to the same receiver has been delivered.

**Proof of Theorem 4**:

Let $p$ be the predicate $(s, r, n, dat) \in Sending$ and $q$ be $(s, r, n-1, dat) \in Receiving$.

We have to prove that, given a state where $p$ holds, every set of applicable operations generates a state where $p$ is still valid or $q$ holds. The premise of the operation *Send_Message* forbids the sending of a message if the previous message from that sender to that receiver has not been sent (the check is on variable *NumMR*). Then, no operations set can act on message $(s, r, n, dat)$ until the message $(s, r, n-1, dat1)$ has been sent.

Predicate $p$ continues to hold for operations in the set which do not act on messages from same sender to same receiver. In the other cases, acting on message $(s, r, n-1, dat')$, will make true $q$.

– **Class Prefix Invariant**

**Theorem 5. Invariant** $Receiving \subseteq PermSMsg$

That is: messages received had previously been in the set of messages to be sent. Variable *PermSMsg* records messages to be sent but it differs from *SendMsg* because messages are not removed from it: it plays the role of a log file.

**Proof of Theorem 5**:

Let us name $p$ the invariant. The initialization operation sets the variable *Receiving* to void. Thus, after the initialization the property holds. Then we have to prove that the property is stable, that is, when $p$ holds on a state the application of every set of operations leads to a state where $p$ still holds. This is true because the operation that increases the cardinality of *Receiving*(*Deliver_Message*) checks that the message has been in *Router* set. The operation that increases the cardinality of *Router* (*Send_Message*) also checks that the message has been in *Sending*. Finally, the operation updating *Sending* (not formally described here) also updates *PermSMsg*. Hence, the theorem follows.

## 4 Two refinements

In the second version the messages are no more monolithic packets: they are composed of a header packet, a set of data packets, and a trailer packet.

In the third refined version the router itself is refined and seen as a grid on which packets flow and switch to reach their destinations.

### 4.1 Second version: token-level routing

In this version the router is still a black box delivering messages, but messages are refined: they are decomposed in packets. Thus, each message consists of a header packet, a sequence of data packets, and a trailer packet.

We study the architectural style of this refinement and notice that components still correspond to *Sender_Struct* and *Receiver_Struct*, and connectors to *Router_Struct*. However, there are some changes in the ports and roles structures. In fact we introduce different ports and roles for different packet types.

Components ports are variables *HeaderSend*, *ValueSend*, *TrailerSend*, *HeaderRec*, *ValueRec*, *TrailerRec* and connector roles are *HeaderRouter*, *ValueRouter*, and *TrailerRouter*. We omit the formalization of the mapping function.

$$HEADER == NAME \times NAME \times \mathbb{N}$$
$$VALUE == NAME \times NAME \times \mathbb{N} \times DATA \times \mathbb{N}$$
$$TRAILER == NAME \times NAME \times \mathbb{N}$$

*Sender* records the number of packets sent for each type and sender.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Sender_Struct \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
HeaderSend : \mathbb{P}\ HEADER \\
ValueSend : \mathbb{P}\ VALUE \\
TrailerSend : \mathbb{P}\ TRAILER \\
PermSH : \mathbb{P}\ HEADER \\
PermSV : \mathbb{P}\ VALUE \\
PermST : \mathbb{P}\ TRAILER \\
Sender : \mathbb{P}\ NAME \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\hline
\end{array}$$

"$PermXX$" variables record packets sent.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Receiver_Struct \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
HeaderRec : \mathbb{P}\ HEADER \\
ValueRec : \mathbb{P}\ VALUE \\
TrailerRec : \mathbb{P}\ TRAILER \\
Receiver : \mathbb{P}\ NAME \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \\
\hline
\end{array}$$

The router keeps trace of the last packets sent from any sender to any receiver.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Router_Struct \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
HeaderRouter : \mathbb{P}\ HEADER \\
ValueRouter : \mathbb{P}\ VALUE \\
TrailerRouter : \mathbb{P}\ TRAILER \\
\hline
\end{array}$$

The transmission operation for the header packet is described in the following schema.

$$\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Transmit_Header \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta Generic_Router \\
\hline
\exists s, r : NAME;\ n : \mathbb{N} \mid (s, r, n) \in HeaderSend \land s \in Sender \\
\qquad \land (s, r, n-1) \in TrailerRouter \bullet \\
\qquad HeaderSend' = HeaderSend \setminus \{(s, r, n)\} \\
\qquad \land HeaderRouter' = HeaderRouter \cup \{(s, r, n)\} \setminus \{(s, r, n-1)\} \\
\hline
\end{array}$$

This is the corresponding rule with pre-solution:

$$\begin{array}{l}
(HeaderSend, \mathbb{P}\ HEADER, (s, r, n)), \\
(Sender, \mathbb{P}\ NAME, s), \\
(TrailerRouter, \mathbb{P}\ TRAILER, (s, r, n-1)), \\
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n-1))
\end{array}$$

and post-solutions:

$$\begin{array}{l}
(Sender, \mathbb{P}\ NAME, s), \\
(TrailerRouter, \mathbb{P}\ TRAILER, (s, r, n-1)), \\
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n))
\end{array}$$

The following schema specifies the delivery of a header packet:

*Deliver_Header*

$$\Delta Generic\_Router$$

$$\begin{array}{l}
\exists\, s, r : NAME;\ n : \mathbb{N} \mid (s, r, n) \in HeaderRouter \land r \in Receiver \bullet \\
\qquad HeaderRec' = HeaderRec \cup \{(s, r, n)\}
\end{array}$$

This is the corresponding CHAM rule with pre-solution:

$$\begin{array}{l}
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n)), \\
(Receiver, \mathbb{P}\ NAME, r),
\end{array}$$

and post-solution:

$$\begin{array}{l}
(Receiver, \mathbb{P}\ NAME, r), \\
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n)) \\
(HeaderRec, \mathbb{P}\ HEADER, (s, r, n))
\end{array}$$

The following schema specifies the sending of data packets.

*Transmit_Data*

$$\Delta Generic\_Router$$

$$\begin{array}{l}
\exists\, s, r : NAME;\ n, d : \mathbb{N};\ dat, dat1 : DATA \mid \\
\qquad (s, r, n, dat, d) \in ValueSend \land \\
\qquad s \in Sender \land (s, r, n) \in HeaderRouter \land \\
\qquad (s, r, n, dat1, d - 1) \in ValueRouter \bullet \\
\qquad ValueRouter' = ValueRouter \cup \{(s, r, n, dat, d)\} \land \\
\qquad ValueSend' = ValueSend \setminus \{(s, r, n, dat, d)\}
\end{array}$$

Here is the corresponding rule with pre-solution:

$$\begin{array}{l}
(ValueSend, \mathbb{P}\ VALUE, (s, r, n, dat, d)), \\
(Sender, NAME, s), \\
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n)), \\
(ValueRouter, \mathbb{P}\ VALUE, (s, r, n, dat1, d - 1))
\end{array}$$

and post-solution:

$$\begin{array}{l}
(ValueRouter, \mathbb{P}\ VALUE, (s, r, n, dat, d)), \\
(Sender, NAME, s), \\
(HeaderRouter, \mathbb{P}\ HEADER, (s, r, n))
\end{array}$$

The following schema defines the delivery of value packets:

*Deliver_Data*

$$\Delta Generic\_Router$$

$$\begin{array}{l}
\exists\, s, r : NAME;\ n, d : \mathbb{N};\ dat, dat1 : DATA \mid \\
\qquad (s, r, n, dat, d) \in ValueRouter \land \\
\qquad r \in Receiver \bullet ValueRec' = ValueRec \cup \{(s, r, n, dat, d)\}
\end{array}$$

Here is the corresponding rule with pre-solution:

$(Receiver, NAME, r)$,
$(ValueRouter, \mathbb{P}\ VALUE, (s, r, n, dat, d))$

and post-solution:

$(Receiver, NAME, r)$,
$(ValueRouter, \mathbb{P}\ VALUE, (s, r, n, dat, d))$,
$(ValueRec, \mathbb{P}\ VALUE, (s, r, n, dat, d))$

We omit the specification of the transmission of Trailer packets.

### 4.2 Analysis

- **Class Eventual Delivery**
  Theorem 1 in the first refinement version is translated with three theorems depending on the different kind of packets.
  - **Theorem 6.** $(s, r, n) \in HeaderSend$ **ensures** $(s, r, n) \in HeaderRouter$
  - **Theorem 7.** $(s, r, n, dat, d) \in ValueSend$ **ensures** $(s, r, n, dat, d) \in ValueRouter$
  - **Theorem 8.** $(s, r, n) \in TrailerSend$ **ensures** $(s, r, n) \in TrailerRouter$

  These properties simply ensure that for every kind of packets, a packet to be sent will eventually be in the set of routed packets.
  **Proof of Theorem 6**:
  Let $p$ is $(s, r, n) \in HeaderSend$. First we want to prove $p$ **unless** $q$. For every operations set applicable to a solution containing molecule
  $(HeaderSend, \mathbb{P}\ Header, (s, r, n))$,
  we have to prove that $p$ still holds after the application. This is true: in fact, all the operations not acting on that header packet do not add that message to the set of routed messages, while the operation *Transmit_Header* acting on that header, modifies the solution making $q$ valid. Thus $p$ **unless** $q$ holds. Furthermore, the set of operations that from a state where $p$ holds generates a state where $q$ holds is the set containing operation *Transmit_Header* acting on the considered molecule.
  The proofs of theorems 7 and 8 are similar: we omit them.
  - **Theorem 9.** $(s, r, n) \in HeaderRouter$ **ensures** $(s, r, n) \in HeaderRec$
  - **Theorem 10.** $(s, r, n, dat, d) \in ValueRouter$ **ensures** $(s, r, n, dat, d) \in ValueRec$
  - **Theorem 11.** $(s, r, n) \in TrailerRouter$ **ensures** $(s, r, n) \in TrailerRec$
  - **Theorem 12.** $(s, r, n) \in HeaderSend$ **leads_to** $(s, r, n) \in HeaderRec$

  The proofs of these theorems are similar to the one of Theorem 6.
- **Class Order Preserving**
  Theorem 4 is refined here in four properties.
  - **Theorem 13.** $(s, r, n) \in HeaderSend$ **unless** $(s, r, n-1) \in TrailerRec$
  - **Theorem 14.** $(s, r, n, dat, 1) \in ValueSend$ **unless** $(s, r, n) \in HeaderRec$

  - **Theorem 15.** $(s, r, n, dat, d) \in ValueSend$ **unless** $(s, r, n, dat, d-1) \in ValueRec$
  - **Theorem 16.** $(s, r, n) \in TrailerSend$ **unless** $(s, r, n, dat, d) \in ValueRec$

  The meaning of theorem 9 is: a header packet to be sent is not delivered until the trailer packet of the previous message from that sender to that receiver has been delivered as well. The other theorems have similar meanings except that they refer to different kinds of packets.
- **Class Prefix Invariant**
  - **Theorem 17.** **invariant** $HeaderRec \subseteq PermSH$
  - **Theorem 18.** **invariant** $ValueRec \subseteq PermSV$
  - **Theorem 19.** **invariant** $TrailerRec \subseteq PermST$

  These invariants are refinements of the invariants of Theorem 4.

### 4.3 Third version: grid-level routing

In this third version the router is a grid where packets flow until they do reach their destination. Packets first move right on the grid; then, when the column corresponding to the receiver is reached, they switch. *Sender_Struct* and *Receiver_Struct* are the same as in the second version, except that $s$ and $r$ are identified by a name and a number (row for sender and column for receiver).

Here is the refined version for the router: *HeaderTrans*, *ValueTrans* and *TrailerTrans* help to store the passage of variables through the grid nodes.

$$
\begin{array}{|l}
\hline
\textit{Router\_Struct} \\
HeaderRouter : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER) \\
ValueRouter : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE) \\
TrailerRouter : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER) \\
HeaderTran : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER) \\
ValueTran : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE) \\
TrailerTran : \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER) \\
\hline
\end{array}
$$

The formalization of the architectural style is refined as well: in fact the ports of the components are attached to the external nodes of the grid. The meaning function shows how interactions between components and connectors have been modified:

$$
\begin{array}{|l}
\mathcal{M}_{Conn} : Connector \rightarrow Router\_Struct \\
\mathcal{M}_{role} : RoleInst \rightarrow MESSAGE \\
\hline
\forall\, c : Connector;\ n : CONNNAME;\ r : Router\_Struct \mid \\
\quad c \in \operatorname{dom} \mathcal{M}_{Conn} \wedge r = \mathcal{M}_{Conn}(c) \\
\quad \wedge\ HeadSub = \{\forall\, n : \mathbb{N};\ h : HEADER \mid n < NumReceiver \bullet (n, 1, h)\} \\
\quad \wedge\ ValueSub = \{\forall\, n : \mathbb{N};\ v : VALUE \mid n < NumReceiver \bullet (n, 1, v)\} \\
\quad \wedge\ TrailSub = \{\forall\, n : \mathbb{N};\ t : TRAILER \mid n < NumReceiver \bullet (n, 1, t)\} \\
\quad \wedge\ HeadSub \subset r.HeaderRouter \wedge ValueSub \subset r.ValueRouter \wedge \\
\quad TrailSub \subseteq TrailerRouter \bullet \\
\quad (HeadSub \cup ValueSub \cup TrailSub) = \mathcal{M}_{role}(\!|\ \{n\} \times c.roles\ |\!)
\end{array}
$$

The following schema defines the transmission of header packets toward the right node:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ \mathit{Transmit\_Header\_Right}\ \_\_\_\_} \\
\Delta \mathit{Generic\_Router} \\
\hline
\exists\, s, r : \mathit{NAME};\ i, j, n, nr : \mathbb{N} \mid (i, j, (s, r, n)) \in \mathit{HeaderRouter} \land \\
\quad j \neq \mathit{ReceiverNum} \land (r, nr) \in \mathit{Receiver} \land j < nr \land \\
\quad (i, j+1, (s, r, n-1)) \in \mathit{TrailerTran} \bullet \\
\quad \mathit{HeaderTran}' = \mathit{HeaderTran} \cup \{(i, j, (s, r, n))\} \land \\
\quad \mathit{HeaderRouter}' = \mathit{HeaderRouter} \setminus \{(i, j, (s, r, n))\} \\
\quad \cup \{(i, j+1, (s, r, n))\} \\
\hline
\end{array}
$$

Here is the corresponding rule with condition:

$j \neq number \land nr > j$

pre-solution:

$(HeaderRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, j, (s, r, n))),$
$(ReceiverNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(NAME, \mathbb{N}), (r, nr)),$
$(TrailerRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER), (i, j+1, (s, r, n-1)))$

and post-solution:

$(ReceiverNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(NAME, \mathbb{N}), (r, nr)),$
$(TrailerRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER), (i, j+1, (s, r, n-1)))$
$(HeaderTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, j, (s, r, n))),$
$(HeaderRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, j+1, (s, r, n)))$

The following schema specifies the transmission of header packets toward the down node:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\_\ \mathit{Transmit\_Header\_Down}\ \_\_\_\_} \\
\Delta \mathit{Generic\_Router} \\
\hline
\exists\, s, r : \mathit{NAME};\ i, n, nr : \mathbb{N} \mid (i, nr, (s, r, n)) \in \mathit{HeaderRouter} \land \\
\quad i \neq \mathit{SenderNum} \land (r, nr) \in \mathit{Receiver} \land \\
\quad (i+1, nr, (s, r, n-1)) \in \mathit{TrailerTran} \bullet \\
\quad \mathit{HeaderTran}' = \mathit{HeaderTran} \cup \{(i, nr, (s, r, n))\} \land \\
\quad \mathit{HeaderRouter}' = \mathit{HeaderRouter} \setminus \{(i, nr, (s, r, n))\} \\
\quad \cup \{(i+1, nr, (s, r, n))\} \\
\hline
\end{array}
$$

Here is the corresponding rule with condition:

$i \neq number,$

pre-solution:

$(HeaderRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, nr, (s, r, n))),$
$(SenderNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(NAME \times \mathbb{N}), (r, nr)),$
$(TrailerTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER), (i + 1, nr, (s, r, n - 1))$

and post-solution:

$(SenderNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(NAME \times \mathbb{N}), (r, nr)),$
$(TrailerTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times TRAILER), (i + 1, nr, (s, r, n - 1))$
$(HeaderTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, nr, (s, r, n))),$
$(HeaderRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i + 1, nr, (s, r, n)))$

The following schema defines the sending of data packets toward right:

*Transmit_Data_Right*

$\Delta Generic\_Router$

---

$\exists\, s, r : NAME;\ i, j, n, nr, d : \mathbb{N};\ dat, dat1 : DATA \mid$
$\quad (i, j, (s, r, n, dat, d)) \in ValueRouter \land$
$\quad (i, j, (s, r, n)) \in HeaderTran \land$
$\quad (i, j, (s, r, n, dat1, d - 1)) \in ValueTran \land j \neq ReceiverNum \land$
$\quad (r, nr) \in Receiver \land j < nr \bullet$
$\quad ValueTran' = ValueTran \cup \{(i, j, (s, r, n, dat, d))\} \land$
$\quad ValueRouter' = ValueRouter \setminus \{(i, j, (s, r, n, dat, d))\}$
$\quad \cup \{(i, j + 1, (s, r, n, dat, d))\}$

Here is the corresponding rule with condition:
$number \neq i \land nr > j$
pre-solution:

$(ReceiverNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(SENDREC \times \mathbb{N}), (r, m, m1, m2, nr)),$
$(HeaderTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, j, (s, r, n))),$
$(ValueTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE), (i, j, (s, r, n, dat1, d - 1))),$
$(ValueRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE), (i, j, (s, r, n, dat, d)))$

and post-solution:

$(ReceiverNum, \mathbb{N}, number),$
$(Receiver, \mathbb{P}(SENDREC \times \mathbb{N}), (r, m, m1, m2, nr)),$
$(HeaderTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times HEADER), (i, j, (s, r, n))),$
$(ValueTran, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE), (i, j, (s, r, n, dat, d))),$
$(ValueRouter, \mathbb{P}(\mathbb{N} \times \mathbb{N} \times VALUE), (i, j + 1, (s, r, n, dat, d)))$

Transmission of data packets toward the down node and transmission of the trailer packets is similar: we omit it to save space.

### 4.4 Analysis

- **Class Eventual Delivery**
  - **Theorem 20.** $(s, r, n) \in HeaderSend \wedge (s, b) \in Sender$ **ensures** $(b, 1, (s, r, n)) \in HeaderRouter$
    We can write similar theorems for every node in the grid. We here write only the one concerning nodes on the last row:
  - **Theorem 21.** $(s, r, n) \in HeaderSend \wedge (s, b) \in Sender \wedge (r, m) \in Receiver$ **ensures** $(b, m, (s, r, n)) \in HeaderRouter$
  - **Theorem 22.** $(s, r, n) \in HeaderSend$ **leads_to** $(s, r, n) \in HeaderRec$

  Theorems 20, 21 and 22 can be stated and proved also for value and trailer packets.
- **Class Order Preserving**
  - **Theorem 23.** $(s, r, n) \in HeaderSend$ **unless** $(ns, 1, (s, r, n-1)) \in TrailerTran$
  - **Theorem 24.** $(s, r, n, dat, 1) \in ValueSend$ **unless** $(ns, 1, (s, r, n)) \in HeaderTran$
  - **Theorem 25.** $(s, r, n, dat, d) \in ValueSend$ **unless** $(ns, 1, (s, r, n, dat, d-1)) \in ValueTran$
  - **Theorem 26.** $(s, r, n) \in TrailerSend$ **unless** $(ns, 1, (s, r, n, dat, d)) \in ValueTran$

  These properties ensure that each packet can be routed on the first node on the grid only if the previous packet has been transmitted ahead. These properties can be stated for every node of the grid and for any kind of packets.
- **Class Prefix Invariant**
  - **Theorem 27.** **invariant** $HeaderRec \subseteq PermSH$
  - **Theorem 28.** **invariant** $ValueRec \subseteq PermSV$
  - **Theorem 29.** **invariant** $TrailerRec \subseteq PermST$

  These are the usual properties already stated for the other refinements.

## 5 Conclusions and future work

The introduction of a chemical semantics for Z offers a formal basis to analyze dynamic properties of non sequential systems [8]. In particular, in this paper we have shown how we can analyze the dynamics of a distributed software architecture specified in Z.

Z was suggested quite early as a means for specifying software architectures; for instance, in [9] it was used to study blackboard systems, that are quite common for some classes of AI applications, as those concerning speech recognition. Even the use of Z to specify architectural styles is not new: for instance, in [1] two different styles are described using Z. However, we think that our new chemical semantics helps in specifying better and more precisely behavioral aspects of architectures. The CHAM itself has been used to study software architectures

[16], however such a notation is not as expressive and modular as Z; the CHAM lacks features especially for describing abstract data structures and their properties. We have chosen an integration approach, which combines advantages of both notations to obtain useful architectural styles descriptions.

Our work can be compared with works which integrate Z with other notations. For instance, in [2] Z schemas are combined with annotations written in Hoare's CSP. CSP is used to specify behavior in form of abstract operations, while Z is used for detailing design features of the system architecture and its main data structures. Such an integration is very low level and not formally specified. In [15] Petri Nets are used to formalize control flows, causal relations, and dynamic behavior of systems statically specified using Z; however, there is no formalization of the interaction between the two notations. [13] offers a more formal model of integration of Z with Petri Nets: Petri Nets are mapped on Z specifications so that graphical representation given by Petri Nets can be used to visualize Z specified systems, yet we think this approach is not giving a formal semantics basis for Z but only a visualizing method. A simpler approach has been suggested by Evans himself in [12]. He uses a Unity like logic [7] to formalize properties on the behavior of systems; an interleaving model with atomic operation interpretation is given but not formalized. Our approach shows how the simplicity and conciseness of Unity logic constructs fit quite well with the operational semantics based on CHAM.

The software architecture we have considered, namely the Message Router, has been studied in [11] and [10]. We deal with it as an architectural style because we have found that it can be the basis for designing other software systems like e-mail systems or news systems.

We are developing a model checking framework based on the CHAM semantics and on the logic given in order to make the verification process automatic; we have already built a symbolic animator of specifications based on the chemical semantics [8].

**Acknowledgements.** Partial support for this work was provided by the Commission of European Union under ESPRIT Programme Basic Research Project 9102 (COORDINATION), and by the Italian MURST 40%- "Progetto Ingegneria del Software".

## References

1. G. 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. M. Benjamin. A Message Passing System. An example of combining Z and CSP. In J. Nicholls, editor, *Proc. 4th Z Users Workshop (ZUM89)*, Workshops in Computing, pages 221–228, Oxford, 1989. Springer-Verlag, Berlin.
3. G. Berry and G. Boudol. The Chemical Abstract Machine. *Theoretical Computer Science*, 96:217–248, 1992.
4. G. Boudol. Some Chemical Abstract Machines. In J. deBakker, W. deRoever, and G. Rozenberg, editors, *A Decade of Concurrency*, volume 803 of *Lecture Notes in*

*Computer Science*, pages 92–123. Springer-Verlag, Berlin, 1993.

5. P. Breuer and J. Bowen. Towards Correct Executable Semantics for Z. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 185–212, Cambridge, 1994. Springer-Verlag, Berlin.
6. S. Brien and J. Nicholls. Z Base Standard, November 1992. Programming Research Group.
7. K. M. Chandy and J. Misra. *Parallel Programming Design.* Addison-Wesley, 1988.
8. P. Ciancarini, S. Cimato, and C. Mascolo. Engineering Formal Requirements: an Analysis and Testing Method for Z Documents. *Annals of Software Engineering*, (to appear), 1997.
9. I. Craig. *Formal Specification of Advanced AI Architectures.* Ellis Horwood, Chichester, 1991.
10. C. Creveuil and G. Roman. Formal Specification and Design of a Message Router. *ACM Transactions on Software Engineering and Methodology*, 3(4):271–307, October 1994.
11. H. Cunningham and Y. Cai. Specification and Refinement of a Message Router. In *Proc. 7th IEEE Int. Workshop on Sw Specification and Design*, pages 20–29. IEEE Computer Society Press, December 1993.
12. A. Evans. Specifying and Verifying Concurrent Systems Using Z. In M. Bertran, T. Denvir, and M. Naftalin, editors, *Proc. FME'94 Industrial Benefits of Formal Methods*, volume 873 of *Lecture Notes in Computer Science*, pages 366–380. Springer-Verlag, Berlin, 1994.
13. A. Evans. Visualizing Concurrent Z Specifications. In J. Bowen and J. Hall, editors, *Proc. 8th Z Users Workshop (ZUM94)*, Workshops in Computing, pages 269–281, Cambridge, 1994. Springer-Verlag, Berlin.
14. D. Garlan and M. Shaw. An Introduction to Software Architecture. In V. Ambriola and G. Tortora, editors, *Advances in Software Engineering and Knowledge Engineering*, pages 1–40. World Scientific Publishing Co., 1992.
15. X. He. PZ Nets: A Formal Method Integrating Petri Nets with Z. In *Proc. 7th Int. Conf. on Software Engineering and Knowledge Engineering*, pages 173–180, Rockville, Maryland, 1995. Knowledge Systems Institute.
16. P. Inverardi and A. 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.
17. D. Perry and A. Wolf. Foundations for the Study of Software Architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, October 1992.
18. M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In J. vanLeeuwen, editor, *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer-Verlag, Berlin, 1995.
19. J. Spivey. *Understanding Z.* Cambridge Tracts in Theoretical Computer Science. Cambridge University Press, 1988.

This article was processed using the LaTeX macro package with LLNCS style