ELSEVIER

Knowledge-Based Systems 12 (1999) 13–26

Knowledge-Based SYSTEMS

# Rule-based systems formalized within a software architectural style[☆]

R.F. Gamble[a,*], P.R. Stiger[a], R.T. Plant[b]

[a]Department of Mathematical and Computer Sciences, University of Tulsa, Tulsa, OK 74104, USA
[b]Department of Computer Information Systems, University of Miami, Coral Gables, FL 33124, USA

Received 24 September 1998; received in revised form 5 January 1999; accepted 5 January 1999

**Abstract**

This article considers the utilization of architectural styles in the formal design of knowledge-based systems. The formal model of a style is an approach to systems modeling that allows software developers to understand and prove properties about the system design in terms of its components, connectors, configurations, and constraints. This allows commonality of design to be easily understood and captured, leading to a better understanding of the role that an architectural abstraction would have in another complex system, embedded context, or system integration. In this article, a formal rule-based architectural style is presented in detail using the Z notation. The benefits of depicting the rule-based system as an architectural style include reusability, understandability, and the allowance for formal software analysis and integration techniques. The ability to define the rule-based architectural style in this way, illustrates the power, clarity, and flexibility of this specification form over traditional formal specification approaches. In addition, it extends current verification approaches for knowledge-based systems beyond the knowledge base only. © 1999 Elsevier Science B.V. All rights reserved.

*Keywords:* Rule-based system; Knowledge-based system; Software architecture; Formal methods

## 1. Introduction

Software systems have primarily been developed in the past within the spectrum of two camps: the *Wet* and the *Dry* as described by Goguen [1]. The *Wet* camp is primarily the ‘hacker’ mentality, in which the system developer wishes to create a system as fast as possible and utilizes as many heuristic design principles as possible to achieve that objective. The system may or may not be documented. In addition, validation of the system can be very difficult to perform completely within the soft specification of requirements. The reality that this approach works, quite often against the odds, is because the designer is not developing a system purely in a random way, but is utilizing heuristic patterns of design, based upon previous experience and anecdotes passed among programmers.

The alternative development camp is the *Dry* community where only formal methodologies are utilized. These design principles have been propagated by ‘the new mathematical puritans’ [2]. Although outside observers may consider these mathematical designs difficult to express and understand upon first examination, more detailed study would show that they are not arbitrary pieces of mathematical specification, but actually follow certain design patterns.

The aim of the research into architectural styles is to bring together these two philosophies in terms of their design principles, both heuristic and formal into a single approach toward solid, correct and rigorous systems design that facilitates stronger verification and validation. Early examples of this confluence between the wet and the dry communities are the design principles laid down by researchers, such as Jackson, who identified a series of rules for constructing ‘structured designs’ from the basic building blocks of programs such as instruction sequence, selection and iteration [3]. These and other design principles have been carried into the research on software architecture and architectural styles [10].

In the same way Jackson attempted to formalize programming heuristics by creating building blocks, or “higher levels of abstraction,” through his sequences selection and iteration “box and data-structured” methodologies, there is a movement in software engineering research to attempt to combine “refined-heuristic” design principles with formal

☆ This research is sponsored in part by the AFOSR (F49620-98-1-0217). The US Govt. is authorized to reproduce and distribute reprints for governmental purposes not withstanding any copyright violation therein. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the AFOSR or the US Govt.

* Corresponding author.
*E-mail address:* gamble@euler.mcs.utulsa.edu (R.F. Gamble)

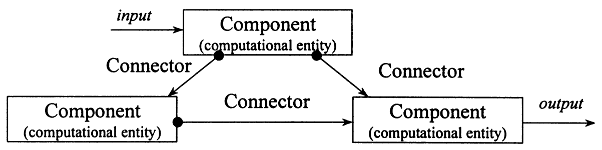

Fig. 1. Architectural styles: components, connectors and configuration.

mathematical notations. One area of concentration for this is in formalizing architectural styles [4–8].

Software architecture provides a way to more comprehensively address the issue of designing a complex software system of component modules [10]. Commonalties in system structure can be exploited through the use of architectural styles, which are defined informally with respect to their components, connectors, configurations, and constraints [24]. Thus, styles provide a common basis for communicating design knowledge. In addition, generic architectural styles can be adapted via specialization for reuse in particular applications.

In this article, we describe a formal model of a rule-based architectural style in terms of its architectural abstractions (components, connectors, and configuration). The use of formal, mathematical modeling of software components has been shown to reduce the ambiguity of a design specification and to contribute to increasing the reliability of the implementation [4–6]. There is a strong need for AI-researchers to have a well-defined architectural style for a rule-based system. Such a style would prove beneficial when, for example, there are multiple rule-based systems contained within or gathered to form a larger complex expert system. Within a common abstraction, the integration and analysis of the connectivity between two or more of the rule-based systems would be greatly facilitated. Another benefit of integration at this more abstract level is that the implementation issues do not impede the developers understanding of the system as a whole. Instead, the architectural abstraction allows developers to concentrate on how the components communicate and what they expect from each other and their environment.

We concentrate on the rule-based architectural style because it is a foundational style for knowledge based systems. Certainly, other forms of knowledge-based systems are being developed, e.g. hybrid, object-oriented, and fuzzy. However, rules remain a major source for knowledge representation. Thus, complex knowledge-based systems may have multiple styles integrated to form a whole system. In this situation, it is even more important to have a common formal abstract representation of the styles to examine and prove properties of performance and quality. We present this work as a reference for determining the answering to the following questions: what is the structural model of a rule-based system, what distinguishes a rule-based system from other architectures, and can a system be proven to be architecturally compliant with a rule-based architectural style.

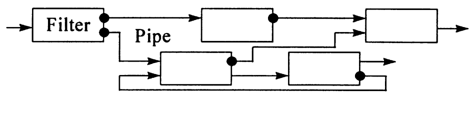

Fig. 2. Diagram for pipe and filter architectural style.

The article is composed of the following sections. Section 2 briefly describes the background of architectural styles and the role of formal methods in those styles. Section 3 presents a formal model of a rule based system in its own style using the Z notation [9]. Section 4 discusses verification and validation of rule-based systems. Finally, Section 5 draws together conclusions to the findings of this work.

## 2. Relevant background

In this section, we present an overview of the role of software architecture, its relationship to design issues in AI-based systems, and the role formal methods plays in architectural description.

### 2.1. Software architecture

As mentioned earlier, the programming community has identified, over time, a series of structures that when used in certain ways lead to predictable and useful behaviors. These structures have become common to solving certain classes of problems, such as object-oriented styles of programming and design, the client–server paradigm, abstraction, layering, etc. The interesting question that a programmer or a designer must ask of themselves is “why would I use one of these approaches in a given situation over another and how can I capture an understanding of each approach to make this comparison?” This is the role of the software architecture or framework. A software architecture depicts how a system is realized by a collection of computational entities (*components*), the interactions among these components (*connectors*), patterns guiding the composition of components and connectors (*configurations*), and *constraints* on the patterns [10] (Fig. 1).

An architectural style has been characterized by Garlan and Shaw [24] as a mechanism for defining “a family of systems in terms of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined” [24]. Thus, for each of the structures that we defined loosely earlier e.g. OOP, client–server, etc. we can now define a set of conditions under which these styles of structure can exist, and apply constraints to determine the behavior of that structure.

Examples of common architectural styles include main program/subprogram, pipe and filter (Fig. 2), event-based, blackboard, and rule-based (Fig. 3). There are certain assumptions and constraints associated with each architecture that guide the designer. For example, the working memory of the system and the preconditions of the rules

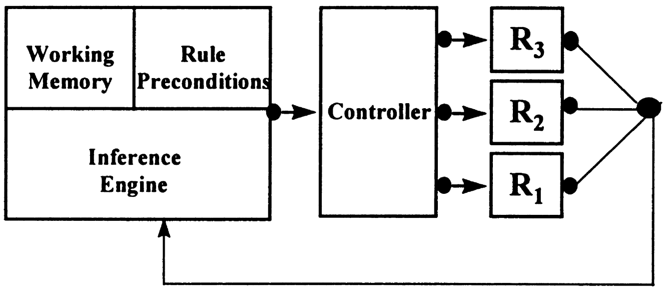

Fig. 3. Rule-based architectural style diagram.

are combined with the inference engine in the rule-based style.

Although depicting a generic architectural style and/or an application in a particular style using a box-and-line diagram provides a way to understand the overall structure of a system, additional text is needed to clarify functionality and constraints. Because the description of a system is at such a high-level, there is a large conceptual gap between the architectural abstractions and the implementation. Formal modeling can bridge the gap owing to its applicability at multiple levels of abstraction and the existence of refinement techniques that transform high-level abstractions into concrete implementations, while preserving correctness.

Until recently AI-based architectures, such as rule-based and blackboard systems, have been largely ignored with respect to this formal modeling framework [7,8]. One reason is that the AI-based architectures are typically more complex than the styles such as pipe and filter and event systems. For example, while within configuration descriptions for pipe and filter, and event systems there is only a single component type and a single connector, within the configuration description of the blackboard and rule-based system styles [11], there exist several distinct component and connector types. Another reason for the complexity is that an AI-based architecture requires some type of an “intelligent” component, that makes decisions or chooses between several alternatives in a context dependent manner and as such, this component can be difficult to formally specify.

## 2.2. Formal methods in software architecture

In this section we will briefly review the literature in terms of the utilization of formal approaches to the specification of architectural styles.

In developing a software architecture, Abowd, et al. [5] formally specify three syntactic classes: components, connectors, and configurations. A component consists of a set of ports (incoming and outgoing) and a description, a connector consists of a set of roles (incoming and outgoing connections to ports) and a description, and a configuration describes a generic attachment function that binds connector roles to component ports. Using these generic primitives as a basis, Abowd, et al. [5] developed formal models in the Z notation [9] of the pipe and filter style, and event system architectural style. We extend their approach to specify the architectural abstractions of the rule-based architectural styles, presented in Section 3.

### 2.2.1. The basic Z notation

Z [9] has a special type constructor, called the *schema*. A schema defines a binding of identifiers or variables to their values in some type. In the notation within the article, user-defined types appear in all uppercase letters. **Bold** font indicates a Z operator. The items above the dividing line are declarations of variables which bind identifiers with values. The items below the dividing line are properties that must hold between the values of identifiers. All common identifiers below the line are scoped by the declarations above the line. Comments appear in italics to the right of the specification entries.

Other Z operations used in the article are as follows: The notation $f{:}X \leftrightarrow Y$ means that $f$ is a relation between elements of type $X$ and $Y$. The notation $f{:}X \nrightarrow Y$ means that $f$ is a partial function from the set $X$ to the set $Y$. If $f$ is a relation or function then **dom** f is the domain of f, and **ran** f is the range of f. If S is a set, then #S is the size of S. If S and T are sequences, then S^T is the concatenation of the two sequences into one. If S is an ordered triple then **S.1** is the first element of the triple, followed by **S.2**, and **S.3**.

Abowd, et al. [5] rely on some Z notational conventions for describing the class of an architectural style component or connector and the behavior of those classes as state machines. For referencing purposes, we term their approach OSS for “Object-State-Step.” The component (connector) *object schema* defines the class of the component (connector) with the attributes each instance of the class must have. These represent the static properties of the component type. The *state schema* for each type represents a binding from identifier to values of the attributes of the class. We can view this binding as the snapshot description of some state machine, that is, the view of the state machine at some point in time [10]. The behavior of the state can be dynamic, as long as it is contained within the bounds of the static properties.

Operations on the state machine are in the form of *step schemas* described as transitions from one legal state to

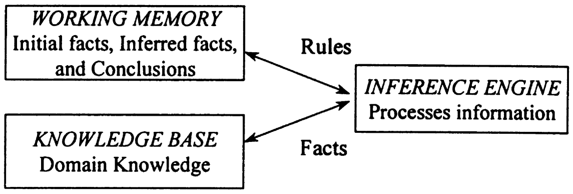

Fig. 4. The basic components of a knowledge-based system.

another, defining the relationship between the values of identifiers before and after the operation. The Δ is used to indicate the changeable variables in the step operations. For instance, if **Rule_Component** is the name of a component schema type, then **RuleState** would be its representative state schema. An instance of the **Rule** class is defined within the **RuleState** schema. **ΔRuleState** is equivalent to two copies of the state schema, one as defined in **RuleState** (called the *before state schema*) and one that has all of the identifiers in **RuleState** decorated with primes ($'$) to define the *after state schema*. The step schema defined for this example would be called **RuleStep** and would most likely operate on **ΔRuleState** to indicate how changes are made to the state variables (see Figs. 13–15).

### *2.2.2. Research in modeling architectural styles*

As stated earlier, OSS [5] provides an initial approach to modeling architectural styles. The approach is demonstrated on two styles: the pipe and filter style and the event system style. For an architectural style, each component type, connector type, and configuration, are defined first as a static class and then operationally with a generic state and generic state transitions. A component is defined to have *ports*, some of which may or may not be explicitly modeled. Not all component ports require connector interaction within the configuration. This allows for input and output from the environment external to the software system. A connector is defined to have *roles*, all of which must be connected to some port in the configuration.

Allen et al. [12] elaborate upon the Abowd, et. al. [5] framework, by providing a formal notation and theory for architectural connectors. They use the WRIGHT architectural description language, which they developed for the specification and analysis of software architectures. The formal modeling of the connectors allows them to check for compatibility of a *port* with a *role*. A port represents the relationship between a component and its environment, i.e. a procedure that can be called, or a variable that can be accessed by interacting with another component [5]. Roles provide the interface to a connector. Each role defines the expected behavior of one of the participants in an interaction [5]. Allen et al. [12] claim that connector models allow for checking whether the configuration is deadlock free, and potentially automate the analysis of architectural descriptions.

We describe the rule-based architectural style based on the guidelines we have developed [11] to extend the OSS approach. Our approach resolves many of the problems with development consistency previously experienced, such as the use of explicit ports and different data types, and provides a clearer pattern of description for directly applying abstractions to other architectural style definitions. For example, there are patterns that determine when individual ports are made explicit in the component type model and when a single collection variable is defined to gather all the information off the ports. In addition to the guidelines, we have developed certain principles of specialization to model subtypes of components and connectors [11].

Abd-Allah [4] creates a formal model in Z of the architectural styles pipe and filter, main program/sub-routine, and event system, constructing a single generic model of the system configuration based on these styles. He divides components and connectors into categories relative to data and control. However, his configuration model is a large conglomeration of the aspects from all of the styles that he examines, such that a specification would include certain aspects, if they were part of that style; but ignore others if they do not pertain to that style. He admits that his approach does not extend to other styles without modification. Gacek elaborates on this approach by adding the blackboard style [25], which requires the introduction of additional variables and constraints into the configuration.

Moriconi, et al. [6] represents a software architecture using concepts that include components, interface, connector, configuration, and mappings of architectural style. Using first order logic he defines some of the entities pertaining to a pipe and filter style. However, Moriconi's research concentrates solely on theories of composition and the method has not been expanded to other styles, additionally not all the entities in the style have been fully defined.

## 3. Rule-based systems

In this section, we present an architectural style, for a

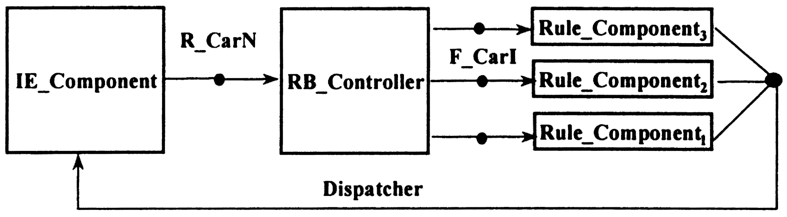

Fig. 5. Architectural abstractions for the rule-based system style.

**IE_Component**

| | |
|---|---|
| actions: **set** ACTION | *All possible actions* |
| facts: **set** FACT | *All possible facts* |
| changes: ACTION ↔ FACT | *All possible incoming action-fact pairs* |
| rule_insts: RID ↔ **set** FACT | *All possible outgoing rule instantiations* |
| states: **set** STATE | *Allowable states* |
| start: STATE | *Beginning state* |
| transition: (STATE × (ACTION ↔ FACT)) ↔ (STATE × RULE-INSTS) | *Valid transitions* |
| start ∈ states | *Start state is valid* |
| changes ⊆ actions × facts | *All incoming data is valid* |
| (⋃ **ran** rule_insts) ⊆ facts | *All outgoing data is valid* |
| ∀ $s_1$,$s_2$: STATE; ch: ACTION ↔ FACT; ri: RULE-INSTS \| (($s_1$,ch),($s_2$, ri)) ∈ transitions • $s_1$,$s_2$ ∈ states ∧ ch ⊆ changes ∧ ri ⊆ rule_inst | *Given the current state and the incoming changes, a new state and the rule instantiations are produced* |

Fig. 6. Inference engine component object schema.

**IEState**

| | |
|---|---|
| ie: IE_Component | *Instance of an IE-Component object* |
| curstate: STATE | *Current internal state* |
| state_vars: IESTATE | *Local variables of instance (part of state)* |
| work_mem: **set** FACT | *Current working memory* |
| precond: **set** PRECONDITION | *Available rule preconditions* |
| instate: ACTION ↔ FACT | *Current input data* |
| outstate: RULE-INSTS | *Current data on output port* |
| curstate = (state_vars,work_mem,precond) | *Curstate is a valid state* |
| curstate ∈ ie.states | *Working memory is valid* |
| work_mem ⊆ ie.facts | |
| instate ⊆ ie.changes | *Input data is contained in changes* |
| outstate ⊆ ie.rule_inst | *Outgoing data is in the outgoing collection variable* |
| (⋃ **ran** outstate) ⊆ work_mem | *Outgoing data is a subset of working memory* |
| instate ≠ ∅ ⇒ outstate = ∅ | *At most one of instate or outstate can have data* |
| outstate ≠ ∅ ⇒ instate = ∅ | |

Fig. 7. Inference engine state specification.

rule-based system in terms of components, connectors, and configuration entities. The usefulness of this style of presentation are that it can be compared directly with other architectural styles for a basis of designing and integrating a system, analysis used for guaranteeing properties of other styles can be extended to this specification and vice versa, and it contributes to a critical mass of models needed for proper formal design of complex systems. In addition, this style shows how distinct component types can be formally described and integrated into a single system, leading to an insight into integrating heterogeneous systems.

## *3.1. Informal system*

Knowledge-based systems are one of the tangible by-products of artificial intelligence research. Essentially, a knowledge-based system attempts to capture expertise in an application area. As knowledge-based systems became more widely used, development products came into existence. Most of these products support at least the three components described in Fig. 4.

Working memory (WM) serves as a repository for the initial information or facts supplied to the KBS. Additional facts may be inferred or deduced by the KBS as it executes and these are stored in WM as well. All conclusions, intermediate and final, are also stored in WM. Representation of WM may be expressed in many formats, such as propositions, predicates, objects, and frames.

The knowledge base (KB) contains the domain knowledge which is often in the form of rules. It does not have to be completely represented as rules. In fact, more robust systems use a hybrid knowledge representation that depends on the available knowledge. A hybrid KB may include rules, procedures, and object messages. We will concentrate our discussion on rules, as they are the most common representation of the KB.

A rule in the KB generally has the form:

$$\text{LHS} \rightarrow \text{RHS}$$

where the left-hand side (LHS) has a set of conditions to be satisfied and the right-hand side (RHS) has a set of actions to perform if the conditions are satisfied.

The inference engine (IE) processes the rules and facts to deduce new facts and conclusions. In a forward chaining system, the LHS of a rule is matched against the available facts in WM. Those rules that match successfully are considered to be instantiated. For serial KBSs, only one rule is selected from the instantiated set for execution.

**IEStep**

| | |
|---|---|
| $\Delta$ IEState | |
| ie′ = ie | *IE object specification remains static. Given a valid current state and valid input, valid output is produced with the update & match functions. The input data is removed from instate and the new output is added to the output port.* |
| $\exists$ out: RULE-INSTS $\bullet$ ((curstate, instate),(curstate′, out)) $\in$ ie.transitions $\wedge$ | |
| instate $\neq \varnothing$ $\wedge$ | |
| work_mem′ = update(instate, work_mem) $\wedge$ | |
| out = match(work_mem′, precond) $\wedge$ instate′ = $\varnothing$ $\wedge$ | |
| outstate′ = outstate $\cup$ out | |

Fig. 8. Inference engine step specification.

The selection is typically based on some form of conflict resolution strategy. Once a rule is chosen, its RHS actions are performed, possibly changing WM and/or the KB. A backward chaining system attempts to match the RHS as goals. Those RHS actions that are sought after goals cause the rule to be placed in an instantiated set. Conflict resolution again plays the part in choosing the executed rule. Execution in backward chaining involves asserting the LHS conditions of the chosen rule as goals to be attained. Some systems perform hybrid chaining to meet their goals. In most cases, the IE halts by explicit command or by the absence of any matching rules.

### *3.2. Formal style*

Our specification of the rule-based architectural style as presented in this section includes the abstraction types of the rule-based system as well as their state and step schemas.

The box-and-line diagram associated with the architectural style we model is shown in Fig. 3. There are multiple component types in the rule base architectural style due to the distinct forms of computation performed by each. These component types are the *Inference Engine* (**IE_Component** in Fig. 5) that performs the matching of the rule preconditions with the contents of working memory, the *Controller* (**RB_Controller** in Fig. 5) that determines the next executing rule, and the *Rules* (**Rule_Component** in Fig. 5) that determines the actions as a result of the input facts from the *Controller*. There are three types of connectors in the rule-based architectural style, however two of them are a specialization of a **Carrier** connector (**R_CarN** and **F_CarI** in Fig. 5). A **Carrier** connector type represents a point-to-point connector that transmits discrete values of data. The **Dispatcher** connector (Fig. 5) also transmits discrete values of data, however it is not point-to-point. In contrast to the existing models of architectural styles the rule-based style is quite complex. Styles such as pipe and filter and event systems have only one component type and one connector type, thus lessening the complexity of their specification.

In the rule-based architecture modeled, the following constraints are assumed:

- Only one pre-condition and only one post-condition is allowed per rule.
- A precondition can contain multiple predicates.
- A postcondition can contain multiple actions.
- An ontology of facts is assumed across the system to maintain semantic consistency.
- Rules are static, although it is possible to specialize the architecture for dynamically changing rules.

We provide the user-defined types, user-defined global functions, and the function types with comments are given later. Because these function are application dependent, and therefore, should be fully modeled at a lower level of abstraction, we do not provide their detailed specification.

**Type Declarations**

[FACT, RID, ACTION, PRECONDITION, PORT, STATE, TEMPLATE] *user-defined types*
RID a *rule component identifier that is a subtype of PORT*

**Distributor – CIM**

| | |
|---|---|
| indata: ALLDATA | *All possible input data* |
| outports: set CID | *Output ports* |
| alphabet: CID $\rightarrowtail$ ALLDATA | *Allowable data on output ports* |
| states: set STATE | *Allowable internal states* |
| start: STATE | *A start state* |
| transitions: (STATE × ALLDATA) $\leftrightarrow$ STATE × (CID $\rightarrowtail$ ALLDATA)) | *Valid transition* |
| start $\in$ states | *Start state is valid* |
| # outports > 1 | *Multiple output ports* |
| outports = dom alphabet | *Output ports are valid* |
| $\forall$ s1, s2: STATE; intran : ALLDATA; outtran: CID $\rightarrowtail$ ALLDATA; | *Valid transition ,states are valid, output ports are valid and incoming and outgoing data is valid* |
| \| ((s1,intran), (s2,outtran)) $\in$ transitions | |
| $\bullet$ s1,s2 $\in$ states $\wedge$ related(intran,indata) $\wedge$ | |
| dom outtran $\subseteq$ outports $\wedge$ outtran $\subseteq$ alphabet | |

Fig. 9. Formal model of the class of distributor CIMs.

```
RB_Controller
┌──────────────────────────────────────────────────────────────
│ Distributor_CIM                                              Includes all properties from Distributor_CIM
│ indata: RULE_INSTS                                           Allowable input
│ outports: set RID                                            Allowable output ports
│ alphabet: RID ⇸ RULE_INSTS                                   Data type restrictions on outports
│ transition: (STATE × (RULE_INSTS))                           Valid transition type definition
│         ↔ (STATE × (RID ↔ RULE_INSTS))
├──────────────────────────────────────────────────────────────
│ ∀ s1,s2: STATE; intran : RULE_INSTS ,outtran: RID ↔ RULE_INSTS   Valid transition, ports are valid, , and
│      | ((s1,intran),(s2,outtran)) ∈ transitions                  outgoing rule instantiations are subset of
│      • s1,s2 ∈ states ∧ intran ⊆ indata ∧ dom outtran ⊆ outports ∧   incoming instantiations
│      ran outtran ⊆ alphabet ∧ ran outtran ⊆ intran
└──────────────────────────────────────────────────────────────
```

Fig. 10. Rule based controller model.

PRECONDITION = RID ↔ TEMPLATE *a relation between a rule identifier and the template of conditions elements that form the LHS of the rule*
RULE-INSTS = RID ↔ set FACT *a relation that associates a rule with available WMES for instantiation*

**User-defined functions (types only)**

**update**: ((ACTION ↔ FACT) × **set** FACT) ⇸ **set** FACT
*updates the contents of working memory*
**unifiedset**: (RULE-INSTS × **set** PRECONDITION) ⇸ (ACTION ↔ FACT)
*produces new changes for IE*
**select**: (RULE-INSTS × STRATEGY) ⇸ RULE-INSTS
*chooses which rule instantiations should fire*
**match**: (**set** FACT × **set** PRECONDITION) ⇸ RULE-INSTS
*matches working memory with rule preconditions to produce new rule instantiations*

### *3.3. The component type specifications*

We will first concentrate on component models, followed by connectors, and then model the overall configuration. For each component type, we employ a pattern of specification to define the model that includes the information on the ports, all possible states the component could reach, a start state, and a legal transition type [5,11].

#### *3.3.1. Inference engine component type*

The inference engine component is a type of repository component that also performs the match phase of the KBS. Therefore, it must collect the incoming changes from one or more rules, if any changes are made, and process them. The variable *changes* declared in Fig. 6 holds a collection of changes off the incoming ports of the IE component. This variable can have any value of action-fact pairs as long as they are part of the port alphabet. The port alphabet described by the cross product of *actions* and *facts*, which represent all possible values of actions and facts, respectively. The variable *rule_insts* is an explicitly defined port variable for the single outgoing port of the IE component that holds the matched preconditions for transmission to the controller.

The allowable state machine behavior of the inference engine is contained within three variables: *states* for all possible states the component can encounter, *start* for a particular start state, and *transitions*. The *transitions* function takes an internal state and a set of action-fact pairs and results in a new internal state and outgoing rule instantiations. These are declared above the midline in the schema **IE_Component** in Fig. 6. The constraints on these variables, which appear in the lower half of the schema in Fig. 6, are that:

- the start state is valid,
- incoming and outgoing data are valid, and
- the state change examines input *changes* and results in output *rule_insts*.

Embedded in the *transitions* function are the actions of matching and selecting appropriate rule instantiations. However, these are modeled within the state schema for this component type.

The state schema for the **IE_Component**, called **IEState**, is specified in Fig. 7. An instance of the **IE_Component** is declared by the statement

ie. **IE_Component**

The state schema asserts that, at any point in a computation, the **IE_Component** is defined by its current state variable, *curstate*, and the incoming and outgoing data, i.e. *instate* and *outstate*, respectively. As defined in the lower half of the state schema in Fig. 7, the *curstate* of the **IE_Component** is a tuple with the composite values of (1) the local variables represented by *state_vars*, (2) the KBS working memory represented by *work_mem*, and (3) the preconditions of the rules represented by *precond*. This explicit state definition is one extension of the original approach by Abowd, et al. [5].

Also in the constraint part below the midline of the IEState specification is that

*curstate* ∈ *ie.states*

This constraint means that *curstate* must be a valid state for an instance of the IE component type. The dot notation

**RB_ControllerState**

| | |
|---|---|
| c: RB_Controller | *Instance of rule-based controller* |
| curstate: STATE | *Current state of the instance* |
| state_vars: RCSTATE | *Current state of the local variables* |
| strategies: **set** STRATEGY | *Valid strategies* |
| instate : RULE_INSTS | *Current incoming and outgoing data* |
| oustate: RID $\leftrightarrow$ RULE_INSTS | |
| curstate = (state_vars, strategies) | *Current state is equivalent to combined state info* |
| curstate $\in$ c.state | *Current state is valid* |
| instate $\subseteq$ c.indata | *Input data is valid* |
| **dom** outstate $\subseteq$ c.outports | *Output data is valid* |
| **ran** outstate $\subseteq$ c.alphabet | |
| instate $\neq \varnothing \Rightarrow$ outstate $= \varnothing$ | *At most one of instate or outstate can have data* |
| outstate $\neq \varnothing \Rightarrow$ instate $= \varnothing$ | |

Fig. 11. Rule based controller specification.

in *ie.states* allows a binding between the actual state definition and the attributes modeling in the component type. This binding forms the essence of the constraints found in the state schema.

As Z is a typed specification language, subtyping is needed to provide more detailed specification while still adhering to the desired component type constraints. For convenience, we use the notation

$\prec$ where A $\prec$ B means that A is a subtype of B

For example, IESTATE $\times$ **set** FACT $\times$ **set** PRECONDITION $\prec$ STATE. This allows *curstate* to be simply typed as the maximal type STATE yet be defined as constrained to the tuple format above. The benefit of using maximal types and subtypes in this manner is that it allows for representational consistency across component specification. This representation consistency faciltates later integration among heterogeneous components [11].

The current incoming data to be processed appears in *instate*, and the current outgoing data appears in *outstate*. As a rule based system computes using discrete cycles, there can only be data on input ports or on output ports but not for both. This is expressed in the last constraint in the schema.

A single computational step for the **IE_Component** transforms the incoming *changes* tuple into the outgoing *rule_inst* using two user-defined functions whose types were declared earlier. The *update* function changes the contents of working memory according to the input *changes*. The *match* function uses the updated working memory and the rule preconditions (*precond*) to produce all possible rule instantiations that can fire, which is placed in *rule_inst* for delivery to the **RB_Controller**.

The step or transition schema, called **IEStep**, is defined in Fig. 8. The use of $\triangle$**IEState** means that the before and after states of the variables declared in **IEState** are defined for **IEStep**. All **IEState** (and **IEState**$'$) constraints hold, making them state invariants. The result of the computation is the removal of action-fact tuples from the input collection variable, and the addition of the new information to the output variable. Time-stamping can be included into the rule-based system by the type being embedded in the facts and then **IEStep** schema can create the time stamp.

### 3.3.2. The rule-based controller component type

The main function of the **RB_Controller** component is to determine the next executing rule or set of rules. This determination is performed using a set of conflict resolution strategies designed to guide the system efficiently through the search space toward a solution. For instance, two conflict strategies often employed in rule-based systems are (1) *distinctiveness*, which eliminates rules firing repeatedly with the same, exact facts, and (2) *specificity*, which when presented with two rules, one of which has a LHS that is a superset of the other, the more specific rule (with the superset) will be chosen. Individual conflict resolution strategies can be ordered and combined, such as the MEA or LEX strategies in OPS5.

Because of its definition, the **RB_Controller** component type is a special type of *controller integration mechanism* or CIM [11]. A CIM provides decision-based control of some kind among multiple components within an architectural configuration. It can be specialized for many situations in which integration via control is required. For example, in real-time system and repository architectural styles, control in the form of scheduling or conflict resolution is needed. In the same respect, a mediator gateway [26] can be described

**RB_Controller Step**

| | |
|---|---|
| $\Delta$ Rule_Based Controller State | |
| c = c$'$ | *Controllerobject characteristics does not change* |
| $\exists$ out: RID $\leftrightarrow$ RULE_INSTS; s: STRATEGY $\bullet$ | *Given a valid curstate and valid input, valid output* |
| ((curstate,instate),(curstate$'$,out)) $\in$ c.transitions $\wedge$ s $\in$ strategies $\wedge$ | *is produced with the select function that takes as its* |
| out = select(instate,s) $\wedge$ instate$'$ = $\varnothing$ $\wedge$ outstate = outstate $\cup$ out | *arguments rule instantiations and a strategy and* |
| | *chooses a rule instantiation.* |

Fig. 12. Rule based controller step specification.

**Rule_Component**

| | |
|---|---|
| rulename: RID | *Name of Rule* |
| insts: **set** FACT | *All possible facts from rule instantiations (incoming data)* |
| changes: ACTION $\leftrightarrow$ FACT | *All possible changes (outgoing data)* |
| actions : **set** ACTION | *All actions applicable to a single rule* |
| facts: **set** FACT | *All facts applicable to a single rule* |
| states: **set** STATE | *Allowable states* |
| start: STATE | *Beginning state* |
| transition: (STATE $\times$ **set** FACT) | *Valid transition* |
| $\leftrightarrow$ (STATE $\times$ (ACTION $\leftrightarrow$ FACT)) | |
| start $\in$ states | *start state is valid* |
| changes $\subseteq$ actions $\times$ facts | *All incoming data is valid* |
| insts $\subseteq$ facts | *All outgoing data is valid* |
| $\forall s_1, s_2$: STATE; ri: **set** FACT; ch: ACTION $\leftrightarrow$ FACT $\mid$ | *Given the current state, rule instantiations, and the set of* |
| $((s_1, ri)(s_2, ch)) \in$ transitions | *preconditions a new state with actionfacts(changes) are produced..* |
| $\bullet\ s_1, s_2 \in$ states $\wedge$ ri $\subseteq$ insts $\wedge$ ch $\subseteq$ changes | |

Fig. 13. The **Rule_Component** object schema.

as a CIM that insulates certain components from changes being made to others. Another type of gateway controller may gather information from disparate sources to coordinate before passing it on to a single component [26]. In addition, a CIM can define global constructs related to an integrated system composed of the same or different architectural styles such as the broker integration strategy [13].

A CIM is modeled as a component in an architectural style because it performs decision-based computation using internal strategies. **The RB_Controller** is a specialization of a generic controller called a **Distributor_CIM**. The formal model of a **Distributor_CIM** can also be expressed using the extended OSS template [11]. The use of maximal types in the generic definition allows the specialization of components, such as the **RB_Controller**, to incorporate more meaningful types.

For example, the user-defined types for the **Distributor_-CIM** as depicted in Fig. 9 are ALLDATA, CID, STATE, STRATEGY, and DCSTATE. ALLDATA is perhaps the most important maximal type to allow for component specialization from the **Distributor_CIM** because it encompasses all possible data types. The data types are further constrained within the specializations. The type CID stands for "component ID". This type is important to a CIM because CIMs must have knowledge of the components for which they make decisions. Thus, in the model of the **Distributor_CIM**, the CIDs are used as explicit port references instead of random ports. The user-defined type STATE is common among all component models that are based on state-machines. STRATEGY and DCSTATE are specific to the state of the **Distributor_CIM**, where the embedded strategies for decision-making reside to form the specific state of the CIM.

The **Distributor_CIM** in general has a collection variable for input called *indata* of type ALLDATA to hold the data off of the port. Multiple output ports are expressed explicitly because the **Distributor_CIM** makes some type of choice among them. Though not modeled in Fig. 9, the order of the information sending may be important. This can be specialized as another CIM. The output ports are defined by *outports*, a set of type CID. The *alphabet* for the output ports is the maximal type ALLDATA contained on those ports, expressed as a partial function from a component identifier to all data types. The variable *transitions* takes an initial state and part or all of the incoming data, and produces a new state and all of the outgoing data on the specified ports. These represent the static properties of this type of CIM.

There are several constraints on the **Distributor_CIM** model. The start state must be valid. There must be at least one output port, otherwise no decision-making would be necessary. The *outports* must have valid alphabets. The

**RuleState**

| | |
|---|---|
| r: Rule_Component | *Instance of a rule* |
| curstate: STATE | *Current internal state* |
| state_vars: RSTATE | *Local variables* |
| precond: PRECONDITION | *Rule precondition* |
| instate: **set** FACT | *Current data on input port* |
| outstate: ACTION $\leftrightarrow$ FACT | *Current data on output port* |
| curstate = (state_vars, precond) | *Current state is equal to all state info* |
| precond.1 = r.rulename | *Precondition is valid* |
| curstate $\in$ r.states | *Curstate is a valid state for a rule* |
| instate $\subseteq$ r. insts | *Incoming data is valid* |
| outstate $\subseteq$ r.changes | *Outgoing data is valid* |
| instate $\neq \varnothing \Rightarrow$ outstate $= \varnothing$ | *At most one can have data at any time* |
| outstate $\neq \varnothing \Rightarrow$ instate $= \varnothing$ | |

Fig. 14. **RuleState** schema.

```
RuleStep
Δ RuleState
r′ = r                                                     Rule information does not change
∃ out: ACTION ↔ FACT                                       Given a valid curstate and valid input, valid
    | ((curstate,in),(curstate′,out)) ∈ r.transitions      output is produced with the unifiedset function
    • out= unifiedset(in,precond) ∧ instate ′ = ∅ ∧ outstate ′ = outstate ∪ out
```

Fig. 15. **RuleStep** schema.

final constraint in the schema presented in Fig. 9 defines a valid transition for the **Distributed_CIM**. The function *related* is used in the transition statement in place of a specific operation that is fully defined within a specialization of this component model.

Using Fig. 9, we next describe the specialized **RB_Controller** from these static properties. This component type schema uses Z schema inclusion to bring in the declarations and constraints of the **Distributor_CIM**. Schema inclusion can be used as long as (1) the newly declared types are subtypes of the included declaration types and (2) any added or amended constraints do not conflict with the included constraints. The component schema for **RB_Controller** is presented in Fig. 10. To define the specialization using schema inclusion, we note that for the user-defined types RID and **set** FACT:

$$\mathrm{RID} \prec \mathrm{CID}$$

$$\mathrm{RULE\_INSTS} \prec \mathrm{ALLDATA}$$

The other user-defined types STATE, STRATEGY, and RCSTATE are analogous to their **Distributor_CIM** counterparts in which

$$\mathrm{RCSTATE} \times \textbf{set}\ \mathrm{STRATEGY} \prec \mathrm{STATE}$$

Using the subtyping declarations above, *indata*, *outports*, *alphabet*, and *transition* are redeclared in **RB_Controller**. The added and amended constraints on the class of **RB_Controller** components provide for a *transition* function redefinition. First, the subset operator details the more generic *related* function from the **Distributor_CIM**, i.e.

$$\mathrm{related}(\mathrm{intran}, \mathrm{indata}) \Leftrightarrow (\mathrm{intran} \subseteq \mathrm{indata})$$

A constraint that the outgoing data must be a subset of incoming data is added. This constraint is required because the functionality of the **RB_Controller** is to simply select which incoming rule instantiations (from the **IE_Component**) should be fired, not to alter the data in any way. For a sequential rule-based system the controller would need an additional constraint restricting the number of outgoing rule instantiations.

Following a similar pattern in modeling the **IE_Component**, we define the state, **RB_ControllerState**, of an instance of the **RB_Controller** component type and the state transition or step, **RB_ControllerStep**. By the specialization, these state and step definitions logically imply those modeled for the class of **Distributor_CIM** components [11].

Fig. 11 describes the **RB_ControllerState**. An instance of **RB_Controller** is defined by

c : **RB_Controller**

linking the static properties of the component object with the state. The current internal state, *curstate*, is a tuple containing *state_vars* combined with the applicable decision strategies, *strategies*, for the component instance. The current incoming data is represented as *instate*, and the current outgoing data is represented as *outstate*. The first two constraints in Fig. 11 describes the valid states of the **RB_Controller**. The latter three constraints detail the content restrictions on *instate* and *outstate*.

A single computational step for the **RB_Controller** component is defined in the schema **RB_ControllerStep** in Fig. 12. This step schema models the transformation of all the incoming data into the outgoing data that is placed on the output ports using the select function. The *select* function in Fig. 12 applies the embedded decision strategies of the **RB_ControllerState** to the input to produce the appropriate output. The incoming data is removed from *instate* and the new output is added to the output ports via *outstate*.

### *3.3.3. The rule component type*

The next component type that is modeled as part of the rule-based architectural style is the rule. Because there is a decision as to which rules are fired, each rule must have a unique name, i.e. *rulename*, in the component type schema **Rule_Component** (Fig. 13). The variable *facts* represents all possible facts that could appear on the input port that are predetermined to instantiate the rule. The connector (as described later) between the **RB_Controller** and the **Rule_Component** strips away the RID embedded in the RULE_INSTS type from the output of the **RB_Controller**. Thus the rule only works with *facts* of type **set** FACT as input.

The set of possible changes that can be made by the RHS actions of a rule are embodied in *changes*. All possible *actions* and *facts* for the class are also defined such that the input *insts* and the output *changes* can be validated, as seen in the constraints on **Rule_Component**. The rule transitions by mapping an internal state and the value of *insts* to a new internal state with outgoing *changes*. Thus, in the constraints, the *transition* is determined by looking at incoming *insts* and results in the outgoing *changes*. In most applications, the rule does not change state. However, we allow for this flexibility in the model.

The state schema for the class of **Rule_Component** is

**R_CarN**

| | |
|---|---|
| inrule,outrule : RULE_INSTS | *Input and output collection variable* |

**R_CarNState**

| | |
|---|---|
| rc : R_CarN | *Instance of R_CARN Connector* |
| indata,outdata : RULE_INSTS | *Incoming and Outgoing data* |
| $indata \subseteq rc.inrule$ | *Input data is valid* |
| $outdata \subseteq rc.outrule$ | *Output data is valid* |
| $indata \neq \varnothing \Rightarrow outdata = \varnothing$ | *At most one of indata or outdata can hold data at any time* |
| $outdata \neq \varnothing \Rightarrow indata = \varnothing$ | |

**R_CarNStep**

| | |
|---|---|
| $\Delta$R_CarNState | |
| $rc = rc'$ | *R_CARN information does not change* |
| $outdata' = outdata \cup indata$ | *Data is removed from indata and delivered to outdata* |
| $indata = \varnothing$ | |

**F_CarI**

| | |
|---|---|
| inport : RID | *Explicit input port* |
| alphabet : RULE_INSTS | *All possible data that may be transmitted on the port* |
| outfact : **set** FACT | *All valid output* |
| **dom** alphabet = {inport} | *The rule instantiations must be for the correct rule* |

**F_CarIState**

| | |
|---|---|
| fc : F_CarI | *Instance of F_CARI Connector* |
| indata : RULE_INSTS | *Incoming instantiation on port* |
| outdata : **set** FACT | *Outgoing collection of facts to particular rule* |
| $indata \subseteq fc.alphabet$ | *Input data is valid* |
| $outdata \subseteq fc.outfact$ | *Output data is valid* |
| $indata \neq \varnothing \Rightarrow outdata = \varnothing$ | *At most one of indata or outdata can hold data at any time* |
| $outdata \neq \varnothing \Rightarrow indata = \varnothing$ | |

**F_CarIStep**

| | |
|---|---|
| $\Delta$F_CarI | |
| $fc = fc'$ | *Point-to-Point information does not change* |
| $outdata' = outdata \cup$ **ran** indata | *All facts are removed from indata's rule instantiation and delivered to outdata* |
| $indata = \varnothing$ | |

Fig. 16. Specialized carrier connectors: **R_CARN** and **F_CARI**.

defined in Fig. 14, as **RuleState**. An instance of the **Rule_-Component** class is created in a similar fashion to the two earlier state schemas defined. The state schema asserts that at any point in a computation, the rule is defined by its current internal state, and the incoming and outgoing data. The variable *curstate* in the **RuleState** schema combines the value of the local variables (*state_vars*) with the internally defined precondition of the rule (*precond*), i.e. its LHS. *Curstate* must be a valid state for an instance of the **Rule_-Component** class, requiring the subtyping declared as:

$$\mathrm{RSTATE} \times \mathrm{PRECONDITION} < \mathrm{STATE}$$

The second constraint, namely

$$\mathrm{precond.1} = \mathrm{r.rulename}$$

is stated to ensure that the rule identifier that maps to the actual template of working memory elements of the precondition is the same as the instance of the rule component defined in the state. The current incoming data to be processed appears in *instate* and the current outgoing data appears in *outstate*. As a rule-based system works in discrete cycles, there can only be data on input ports or output ports but not both. This constraint is defined in the schema.

A single computational step for the class of **Rule_Component** transforms the incoming *insts* into the outgoing *changes* using the user-defined function, *unifiedset*, declared earlier. The *unifiedset* function unifies the incoming *insts* with the *precond*, produces a set of *changes*, and makes those *changes* available for delivery to the **IE_Component**. The results of the computation step are the removal of the rule instantiation from the incoming data and the addition of the new changes to the outgoing data. The step or transition schema, called **RuleStep**, is described in Fig. 15.

## 3.4. The connector models

As shown in Fig. 5, connectors pass information among components. The rule-based architectural style requires three types of connectors. Two of the connectors are specializations of the Carrier connector [11]. The **Carrier** connector is a point-to-point connector that transports discrete data. This means that this generic class of

**Dispatcher**

| | |
|---|---|
| inchanges,outchanges: ACTION $\leftrightarrow$ FACT | *Input and Output collection variable* |

**DispatcherState**

| | |
|---|---|
| ds : Dispatcher | *Instance of Carrier Connector* |
| indata,outdata : ACTION $\leftrightarrow$ FACT | *Incoming and outgoing data* |
| indata $\subseteq$ ds.inchanges | *Input data is valid* |
| outdata $\subseteq$ ds.outchanges | *Output data is valid* |
| indata $\neq \varnothing \Rightarrow$ outdata $= \varnothing$ | |
| outdata $\neq \varnothing \Rightarrow$ indata $= \varnothing$ | |

**DispatcherStep**

| | |
|---|---|
| $\Delta$DispatcherState | |
| ds = ds′ | *Dispatcher does not change* |
| outdata′ = outdata $\cup$ indata | *Data delivered to outdata* |
| indata′ $= \varnothing$ | *Data is removed from indata* |

Fig. 17. **Dispatcher** connector.

connectors can either be in a state in which (a) there is no data incoming or outgoing or (b) there is either incoming data or outgoing data, but not both. There are four types of **Carrier** connectors: (1) **CarB** with both input and output ports explicitly defined, (2) **CarI** with only an input port explicitly defined, (3) **CarO** with only an output port explicitly defined, and (4) **CarN** with no ports explicitly defined. The two **Carrier** connector types used in the rule-based architectural style are named **F_CarI** (a specialization of the **CarI** connector that transports facts) and **R_CarN** (a specialization of the **CarN** connector that transports rule instantiations). The third connector type used is called the **Dispatcher** that collects changes from the rules and passes them to the **IE_Component**.

The **R_CarN** connector transmits *rule_inst* from the **IE_Component** to the **RB_Controller** (Fig. 16). This type of connector class is defined by an input collection variable *inrule*, which contains all possible incoming rule instantiations, and an output collection variable *outrule*, which contains all possible outgoing rule instantiations. The behavior of **R_CarN** is defined by giving its transmission policy within the **R_CarNState** schema in Fig. 16, which is consistent with the discrete transmission of the generic **Carrier** connector. A single step in the behavior of the class of **R_CarN** connectors, as modeled by **R_CarNStep**, results in the incoming rule instantiations, *indata*, being removed and being delivered to the outgoing rule instantiations, *outdata*.

The **F_CarI** connector type is located between the **RB_Controller** and the rules. It transports the **RB_Controller** *rule_inst* as a set of facts to the selected rules as modeled in Fig. 16. An **F_CarI** connector is defined by an input port of type RID, the type of data that is carried on the input port (*alphabet*) of type RULE_INSTS, and the output collection variable *outfact* which contains all possible outgoing facts (of type **set** FACT). The behavior of this type of connector is defined by its transmission policy modeled in **F_CarIState** in Fig. 16 which is consistent with the discrete transmission of the generic **Carrier** connector. A single step in this connector type results in the incoming rule instantiations, *indata*, being removed, and being delivered as facts to the outgoing data as modeled in **F_CarIStep**.

The third type of connector class, **Dispatcher**, has an input collection variable *inchanges*, representing all possible incoming changes, and an output collection variable *outchanges*, representing all possible outgoing changes (Fig. 17). The **Dispatcher** connector collects and transports the changes from the individual **Rule_Component** instances to the **IE_Component**. It is assumed that the order of rule changes in a single cycle is not important. The behavior of a **Dispatcher** is defined by giving its

**InteractingRB_Set**

| | |
|---|---|
| ie : IE_Component | *Instance of the Inference Engine* |
| r_set : **set** Rule_Component | *Instances of Rules* |
| ct : RB_Controller | *Instance of the Controller* |
| rc : R_CarN | *Instance of the connector between IE and Controller* |
| fc : **set** F_CarI | *Set of connector instances from Controller to Rules* |
| ds : Dispatcher | *Connector instance between Rules and Inference Engine* |
| rc.inrule = ie.rule_inst | *Binding R_CarN connector to IE* |
| rc.outrule $\subseteq$ ct.indata | *Binding R_CarNr connector to Controller* |
| $\forall$ f : fc $\bullet$ f.inport $\in$ ct.outports $\wedge$ ct.alphabet(f.inport) = f.alphabet | *Binding F_CarI connector to Controller* |
| $\forall$ f : fc $\bullet$ ($\exists$r : r_set $\bullet$ r.rulename = f.inport $\wedge$ r. insts = f.outfact) | *Binding F_CarI Connector to Rules* |
| ds.inchanges $\subseteq \bigcup_{r:\,R\_set}$\{af : r.actions $\leftrightarrow$ r.facts $\bullet$ af\} | *Binding dispatcher Connector to Rules* |
| ds.outchanges = ie.changes | *Binding dispatcher Connector to IE* |

Fig. 18. Configuration of rule-based architectural style.

transmission policy as modeled by the state schema **DispatcherState**. At any point in time, the **Dispatcher** at most one of *indata* or *outdata* may have data. In this sense, it is very similar to the **Carrier** connector because of its discrete transformation, but it is not point-to-point (see Fig. 5). A single step (**DispatcherStep**) in the behavior of a **Dispatcher** results in the incoming changes, *indata*, being removed from the input variable and being delivered as outgoing changes to *outdata*.

### *3.5. The configuration specification*

Following is the model of the configuration of components and connectors into an interacting set that describes the overall functionality of the rule-based architecture (Fig. 18). The variable declarations are instances of the components and connectors that we have previously modeled. There is one **IE_Component**, one **RB_Controller**, a set of **Rule_Components**, one **R_CarN** connector, a set of **F_CarI** connectors, and one **Dispatcher** connector. The constraints that appear in the schema establish the correct bindings between the connectors and the components. The first constraint restricts the **R_CarN** connector's incoming data represented by *inrule* to be the same as the **IE_Component** outgoing data represented by *rule_inst*. The second constraint confines the **R_CarN** connector's outgoing data represented by *outrule* to be the same as the **RB_Controller** incoming data represented by *indata*. The next two invariants bind the set of **F_CarI** connectors to the **RB_Controller** and the individual **Rule_Components**. In the first of these constraints, the input port represented by *inport* of all **F_CarI** connectors must be represented as **RB_Controller** output ports in *outports*. Also, within this constraint, data on an **F_CarI** port, represented by its *alphabet*, must be equivalent to the data in the **RB_Controller** *alphabet* for that port (RID). In the next constraint, there must be a rule whose RID matches an **F_CarI** connector *inport*. Also in this constraint, the output data of the **F_CarI** connector, represented by *outfact*, is equivalent to the incoming data represented by *insts* of this **Rule_Component** instance. The last two constraints bind the **Dispatcher** connector to the individual rules and the **IE_Component**. The **Dispatcher** incoming data, represented by *inchanges*, is restricted, by the generalized union operator, to be a subset of the actions and facts of the rules. The **Dispatcher** outgoing data, represented by *outchange*, is equivalent to the **IE_Component** incoming data, represented by *changes*.

The behavior of the state for the configuration defined in Fig. 18 is defined as the behaviors of all the components and connectors previously defined. The state of the system can be formulated by identifying the component and connector states with components and connectors in the system. The step of a traditional rule-based system is one of the match-select-act. In the defined architectural style, the match is performed as part of the **IE_Component** step, the select is performed as part of the **RB_Controller** step, and the act is performed as part of a **Rule-Component** step. The configuration of the rule-based architectural style includes all of the architectural abstractions. Thus, the step of the configuration would include those transmissions made by connectors, as well as the transitions of components. However, because of the discrete nature of the rule-based system, only one step in the configuration can be performed at a time, with all else remaining the same.

## 4. Verification and validation of knowledge based systems

The use of architectural style models affords the knowledge engineer the opportunity to verify the complete underlying system behavior while allowing the developers to alter the knowledge base at will. The modeling approach presented in this article provides a rigorous and comprehensive analysis foundation of the overall knowledge based system functionality. The style definition provides information on integration together with the minimal requirements for rule and controller entrance into a larger, more complex, system architecture.

The ability to understand the system at this level allows the knowledge engineer to then utilize traditional verification and validation techniques at detailed application levels. These techniques have been extensively documented and vary from the formal approaches to the experimental [14–21].

There has been a considerable amount of tool development research performed with over 40 tools occurring in the literature (an extensive survey of the tools can be found in [22]). The use of the architecture abstraction also enables the knowledge engineer to consider the overall system from a clear abstract specification and, having considered the verification and validation requirements, determine the most applicable tools with which to analyze the system.

The use of the rule-based architectural style is also beneficial in the area of heterogeneous systems where multiple knowledge bases may be integrated together or indirectly interact [23]. Validation and verification research in this area is beginning to emerge, as the existing techniques have traditionally been limited to static aspects of systems with single knowledge bases. For instance, O'Leary has considered the problem of multiple knowledge bases systems being integrated together assuming the same ontology [23]. We feel that the architectural approach would be a suitable complimentary technique to employ to this class of problems.

## 5. Conclusions

This article has described a formal model for the rule-based architectural style. This form of specification allows for a complex knowledge-based system representation to be created and developed in a straightforward manner. The styles based specification is clearly advantageous over traditional forms of specification, in that it facilitates commonalties of design, acts as an enabling vehicle for communicating about the design, and as such leads to a robust and constructive methodology for the creation of knowledge-based systems specifications.

The article built upon the work of previous researchers such as Shaw and Garlan [10] in the utilization of their components, connectors and configurations, in conjunction with the use of Z to specify elements of these architectural abstractions. The aim is to create not only a formal architecture of the rule-based system, but to illustrate the correspondence between the structural and semantic aspects of the components and their interactions. This is a first step towards showing the correspondence at the next level of abstraction, between the system requirements and the implemented system.

The utilization of architectural styles can be seen, therefore, to aid in the areas of design, development and maintenance. The design is enhanced through better notations, the ability to have more correspondence between abstractions and the ability to perform better analysis upon these designs. In this respect, the validation process is strengthened. The development process is enhanced through the ability to show correspondence between levels of abstraction and ultimately through formal requirements. Hence, this strengthens the verification process. Finally, the maintenance process is enhanced through a better documented, more understandable, reusable form. Thus, a more reliable form of the validation and verification process can be implemented over the life of the system and its specification.

The creation of the architectural style was performed on the rule-based system in order to demonstrate the feasibility of the approach with respect to a fundamental representational form. Achieving this style allows for the system to be implemented or utilized as a part of another structure, such as a heterogeneous style. Hence, it can be seen that architectural styles such as the one presented are the building blocks of future, more complex systems. Systems that otherwise could not be built to acceptable levels of validation and verification, due to their complexity, and the associated difficulty of creating large complex formal specifications and the costs associated with the traditional approach to building these formal models. We see architectural styles as the premier approach to the construction of large complex systems in the future.

## References

[1] J. Goguen, The dry and wet. Monograph PRG-100, Programming Research Goup, Oxford University Computer Laboratory England.
[2] D. Ince, Software development: Fashioning the Baroque, Oxford Science Publications, Oxford University Press, New York, 1988.
[3] R. Storer, M.A. Jackson, Pratical Program Development Using JSP: A Manual of Program Design Using the Design Method Developed by M.A. Jackson, Blackwell Scientific Press, Oxford, England, 1986.
[4] A. Abd-Allah, Composing Heterogeneous Software Architectures, Ph.D. Dissertation, Department of Computer Science, University of Southern California, August 1996.
[5] G. Abowd, R. Allen, D. Garlan, Formalizing Style to Understand Description of Software Architecture, ACM TOSEM, submitted for publication.
[6] M. Moriconi, X. Qian, R.A. Riemenschneider, Correct architecture refinement, IEEE Transactions on Software Engineering 21 (4) (1995) 356–372.
[7] S. Murrell, R. Plant, R. Gamble, Defining architectural styles for knowledge-based systems, AAAI-95, Workshop on Verification and Validation of Knowledge Based Systems and Subsystems, August 1996, pp. 51–58.
[8] P.R. Stiger, R.F. Gamble, Blackboard Systems Formalized within a Software Architectural Style, International Conference on Systems, Man, Cybernetics, October 1997.
[9] J. Spivey, Introducing Z: A Specification Language and its Formal Semantics, Cambridge University Press, Cambridge, 1988.
[10] M. Shaw, D. Garlan, Software Architecture: Perspectives on an Emerging Discipline, Prentice Hall, Englewood Cliffs, NJ, 1996.
[11] P.R. Stiger, An assessment of architectural styles and integration components, MS Thesis, Department of Mathematical and Computer Sciences, University of Tulsa, January 1998.
[12] R. Allen, D. Garlan, A Formal Basis for Architecture Connection, ACM TOSEM, 1997, submitted for publication.
[13] D. Mularz, Pattern-based integration architectures, PloP 1994.
[14] A.D. Preece, Validation and Verification of Knowledge-based Systems, Working Notes AAAI 1993, Washington D.C.
[15] R.T. Plant, Validation and Verification of Knowledge-based Systems, Working Notes AAAI 1994, Seattle, Washington.
[16] R.Gamble, C. Landauer, Validation and Verification of Knowledge-based Systems, Working Notes IJCAI 1995, Montreal, Quebec, Canada.
[17] J. Schmolze, Vermesan, A., Validation and Verification of Knowledge-based Systems, Working Notes AAAI 1996, Portland, Oregon.
[18] R.T. Plant, G. Antoniou, Validation and Verification of Knowledge-based Systems, Working Notes AAAI 1997, Providence, RI.
[19] J. Cardenosa, P. Meseguer, Proceedings of EUROVAV 1993: European Symposium on the Verification and Validation of Knowledge-based Systems, Univ. Polit. De Madrid, Spain.
[20] M. Ayel, M. Rousset, Proceedings of EUROVAV 1995: European Symposium on the Verification and Validation of Knowledge-based Systems, St Baldoph-Chambery, France.
[21] J. Vanthienen, F. van Harmelen, Proceedings of EUROVAV 1997: European Symposium on the Verification and Validation of Knowledge-based Systems, Leuven, Belgium.
[22] S. Murrell, R.T. Plant, A Survey of Tools for the Validation and Verification of Knowledge-based Systems: 1985–1995. Decision Support Systems 633 (1997).
[23] D.E. O'Leary, Verification of Multiple Agent Knowledge-Based Systems, AAAI-97, Workshop on Verification and Validation of Knowledge Based Systems and Subsystems, August 1997, pp. 13–23.
[24] D. Garlan, M. Shaw, Advances in Software Engineering and Knowledge Engineering, An introduction to software architecture, World Scientific, Singapore, 1993.
[25] C. Gacek, Detecting architectural mismatches during systems composition, TR USC/CSE-97-TR-506. University of Southern California, Center for Software Engineering, 1997.
[26] M. Brodie, M. Stonebroker, Migrating Legacy Systems, Morgan Kaufmann, San Francisco, CA, 1995.