Capturing Interactions in Architectural Patterns

Dharmendra K Yadav
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Powai, Mumbai 400076, India
Email: dharmendra@cse.iitb.ac.in

Rushikesh K Joshi
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Powai, Mumbai 400076, India
Email: rkj@cse.iitb.ac.in

***Abstract*—Patterns of software architecture help in describing structural and functional properties of the system in terms of smaller components. The emphasis of this work is on capturing the aspects of pattern descriptions and the properties of inter-component interactions including non-deterministic behavior. Through these descriptions we capture structural and behavioral specifications as well as properties against which the specifications are verified. The patterns covered in this paper are variants of Proxy, Chain, MVC, Acceptor-Connector, Publisher-Subscriber and Dinning Philosopher patterns. While the machines are CCS-based, the properties have been described in Modal μ-Calculus. The approach serves as a framework for precise architectural descriptions.**

I. Introduction

In component/connector based architectural descriptions [6], [13], components are primary entities having identities in the system and connectors provide the means for communication between them. This view is very similar to the abstractions provided by CCS [12], in which, components can be seen as non-movable agents and connectors as channels. Specification of software architectures from designer's point of view occurs at different levels such as process, component, module and object levels. The focus of this work is on modeling some commonly occurring architectural patterns at the component level in terms of CCS. μ-Calculus [8] is used to specify example properties against which the architectural descriptions can be verified. The CCS based approach effectively captures interactions occurring at architectural level through its features such as components and their compositions, input/output actions over channels and non-deterministic behavior. The machines described in this paper have been verified with the Concurrency Workbench [4]. The grammar used in specifications of the properties is as provided in [14].

II. Related Work and Background

Various approaches for specifying and verifying software architectures can be found in the literature. Architecture Description Language (ADL) based approaches consist of languages defined to describe, model and implement software architectures. A classification of ADL based languages can be found in [11]. Some of these languages have formal semantics supporting formal analysis. For example, Darwin [10] is a general purpose configuration language for distributed dynamic system that uses π-calculus to model the component interaction and composition properties. Wright [2], which represents interface points as ports uses a subset of CSP for it's formal semantics. It allows architects to specify temporal communication protocols and check properties such as deadlock freedom. Besides special purpose ADLs, general purpose modeling techniques such as UML have also been found to be useful for modeling high level software architectures [7], [9].

TABLE I
A Summary of CCS Combinators

Primitives & Examples	*Descriptions*	*Architectural Significance*
Prefix (.) p1.$\overline{p2}$	Action sequence	intra-component sequential flow
Summation (+) A1 + A2	Nondeterminism	choice within a component
Composition (|) A1 | A2	Connect matching i/o ports in assembly	multiple connected components
Restriction (\) A\{p1, k1, ..}	Hiding ports from further composition	Internal features
Relabeling ([]) A[new/old, ..]	Renaming of ports	syntactic renaming

We take an ADL-like approach in describing architectures in terms of components and connectors, and use CCS, a process calculus, to model and analyze software architectures as it provides abstractions (agents and channels) which are very similar to components and connector abstractions found commonly in software architecture descriptions.

In CCS, agents have input and output ports. The processes or agent expression for an agent is constructed from a set of atomic actions involving ports. A port name with an overbar such as $\overline{p}$ represents an output port. In the basic form of the language, data values can not be passed unlike in the value passing form as in expression $\overline{p}(x)$. Agents communicate with each other via connected pairs of input and output ports. Basic combinators in CCS are summarized in Table I.

The formal semantics of the CCS [12] is given by transitional semantics. In this the general notion of $labeled\ transition\ system$ as given below is used

$$(S, T, \{\xrightarrow{t}: t \in T\})$$

Which consists of a set S of $states$, a set T of $transition\ labels$, and a $transition\ relation$ $\xrightarrow{t} \subseteq S \times S$ for each $t \in T$. In this transition system S is taken to be $\mathcal{E}$, the agent expression, and T is taken to be Act, the $actions$. The

TABLE II
Syntax of modal mu-calculus

prop Φ	::=	$tt \mid \mathit{ff} \mid X \mid \Phi_1 \wedge \Phi_2 \mid \Phi_1 \vee \Phi_2 \mid [K]\Phi \mid$ $\langle K\rangle\Phi \mid \nu X.\Phi \mid \mu X.\Phi \mid \mathbf{not}\ prop \mid (prop) \mid$ $\mathbf{AG}\ prop \mid \mathbf{AF}\ prop \mid \mathbf{A}\ (prop \cup prop\ ) \mid$ $\mathbf{A}\ (prop\ W\ prop\ ) \mid \mathbf{EG}\ prop \mid \mathbf{EF}\ prop \mid$ $\mathbf{E}\ (prop \cup prop\ ) \mid \mathbf{A}\ (prop\ W\ prop\ )$

semantics for $\mathcal{E}$ consists in the definition of each transition $\xrightarrow{\alpha}$ over $\mathcal{E}$. The transitions of each composite agent is defined in terms of the transitions of its component agent or agents. The general rule of inference will be:

$$\text{From } \ E \xrightarrow{\alpha} E' \text{ infer } \ E|F \xrightarrow{\alpha} E'|F$$

and it can be written in the form

$$\frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F}$$

There will be one or more transition rules associated with each combinator. The set of transition rules are as follows:
The names **Act, Sum, Com, Res and Rel** indicates that the rules are associated respectively with Prefix, Summation, Composition, Restriction and Relabeling.

$$\mathbf{Act}\ \frac{}{\alpha.E \xrightarrow{\alpha} E} \quad \mathbf{Sum}_j\ \frac{E_j \xrightarrow{\alpha} E'_j}{\sum_{i\in I} E_i \xrightarrow{\alpha} E'_j}(j \in I)$$

$$\mathbf{Com}_1\ \frac{E \xrightarrow{\alpha} E'}{E|F \xrightarrow{\alpha} E'|F} \quad \mathbf{Com}_2\ \frac{F \xrightarrow{\alpha} F'}{E|F \xrightarrow{\alpha} E|F'} \quad \mathbf{Com}_3\ \frac{E \xrightarrow{l} E' \quad F \xrightarrow{\overline{l}} F'}{E|F \xrightarrow{\tau} E'|F'}$$

$$\mathbf{Rel}\ \frac{E \xrightarrow{\alpha} E'}{E\backslash L \xrightarrow{\alpha} E'\backslash L}(\alpha, \overline{\alpha} \notin L) \quad \mathbf{Res}\ \frac{E \xrightarrow{\alpha} E'}{E[f] \xrightarrow{f(\alpha)} E'[f]}$$

For the specification of properties, Modal μ-Calculus has been used. The modal μ calculus [16], [17] used here is an extension of Henessy-Milner logic with two fixed point operators. These two operators, least fixed $\mu X.\phi(X)$ and greatest fixed points $\nu X.\phi(X)$ allow specification of iterative behavior in the system, where $\phi(X)$ is a state predicate in which state predicate variable X can occur. The syntax of modal μ calculus is summarized in Table II for reference.

III. PROXY

In this section, the interactions occurring in a caching proxy [3] that hides the actual server and also caches information from the server are modeled. When a client makes a request, the reply may be handed over to the client either by the proxy or from the server through the proxy. The CCS model shown in Figure 1 captures the sequence of flow of messages. The CCS model includes the two possibilities through non-deterministic summation. The pattern is a composition of three components. The description of the components is given below.

A. Pattern Description

$$\begin{aligned} Client &= \overline{req}.ans.Client \\ Proxy &= req.\overline{ans}.Proxy + \\ &\quad req.\overline{request}.reply.\overline{ans}.Proxy \\ Server &= request.\overline{reply}.Server \\ Arch &= Client|Proxy|Server \end{aligned}$$

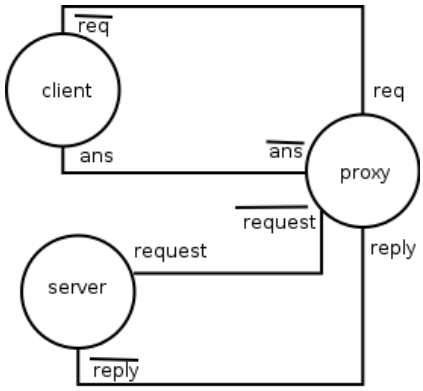

Fig. 1. Caching Proxy Pattern

The desirable properties of interaction can be captured in modal μ calculus. These properties can be used as verification properties to model-check the architectural descriptions. Some properties are outlined below. It can be noted that the properties are expressed in terms of τ actions that occur in the above machine. For each τ action, a separate $observation$ action is introduced in the machine. The properties below use these $observation$ actions. The property-friendly verifiable machine is listed subsequently. The properties mentioned below are necessary properties. They are not sufficient. Through these properties we have tried to capture some of the essential behaviours of the patterns.

prop P_1	=	$AG(not\ \langle treq\rangle tt) \vee AF(\langle tans\rangle tt)$
prop P_2	=	$A(not(\langle tans\rangle tt)W\langle treq\rangle tt)$
prop P_3	=	$AG(not(\ \langle treq\rangle tt) \vee AF(\langle tans\rangle tt \vee$ $\langle trequest\rangle tt))$
prop p_4	=	*max* $X = \langle t\rangle\langle treq\rangle\langle t\rangle\langle trequest\rangle\langle t\rangle\langle treply\rangle\langle t\rangle$ $\langle tans\rangle X$
prop p_5	=	*max* $Y = \langle t\rangle\langle treq\rangle\langle t\rangle\langle tans\rangle Y$

In the verifiable machine below, the observation actions introduced corresponding to the τ actions are $tans$, $treq$, $trequest$, $treply$. The observation actions are introduced as input actions, and they are inserted uniformly immediately following input actions of the corresponding τ actions. The same structure is followed for other patterns described in the subsequent sections.

$$\begin{aligned} Client &= {}'req.ans.tans.Client \\ Proxy &= req.treq.{}'ans.Proxy + \\ &\quad req.treq.{}'request.reply.treply.{}'ans.Proxy \\ Server &= request.trequest.{}'reply.Server \\ Arch &= (Client|Proxy|Server)\backslash \\ &\quad \{req, ans, request, reply\} \end{aligned}$$

The properties can be read as below.

- Whenever a client makes a request, eventually there will always be an answer to the client (P_1).
- Whenever there is an answer to the client, there is a prior request from the client (P_2).
- Immediately after a request, either an answer or a further request will be generated from the proxy component (P_3).
- There is a possibility of a request from client, followed by request generated from the proxy, followed by a reply from the server and finally an answer from the proxy. This sequence of actions may repeat infinitely(P_4).
- There is a possibility of a request from client, followed by an answer from the proxy. This sequence of actions may also repeat infinitely(P_5).

In the subsequent section, the same template is followed for presenting the descriptions of machines and properties of other architectural patterns. A brief description of the pattern to be described is given first. The CCS model characterizes the pattern's components, component behaviors, channels and interactions through the channels. The properties capture the typical interaction properties and the corresponding verifiable machines are also provided.

IV. Model View Controller architectural description

The Model-View-Controller architectural pattern [3] divides an interactive application into three parts. The model contains core functionality and data. The controller changes the model. Whenever there is change in model, the view is required to reflect the current state of model. Thus, the controller controls the model, while the view keeps track of the changes in the model. The components and the interaction channels for an MVC system are depicted in Figure 2.

A. Structural and Behavioral Specification

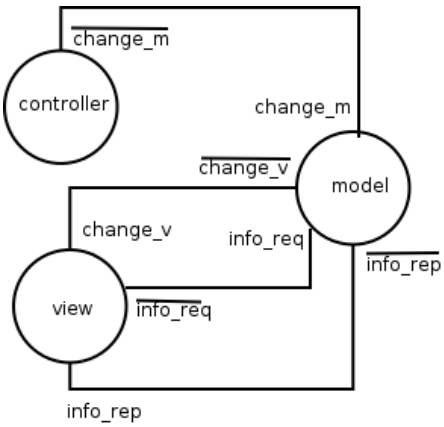

Fig. 2. Model View Controller Pattern

$$
\begin{aligned}
Controller &= \overline{\textit{change-m}}.Controller \\
Model &= \textit{change-m}.\overline{\textit{change-v}}.Model + \\
&\quad \textit{info-req}.\overline{\textit{info-rep}}.Model \\
View &= \textit{change-v}.\overline{\textit{info-req}}.\textit{info-rep}.View \\
Arch &= Controller|Model|View
\end{aligned}
$$

B. Verification Properties

some of the interaction properties for the MVC patterns are listed below.

- If there is a request for updated information, eventually there will be a reply (P_1).
- It is not the case that without any request for updated information, there will be reply for the information (P_2).
- If the controller outputs a request for change in model a change in view also takes place (P_3).
- It is not the case that without any change from the controller, there will be change in view (P_4).

$$
\begin{aligned}
\textit{prop } P_1 &= AG(not\ \langle \textit{tinfo-req} \rangle tt) \\
&\quad \vee AF(\langle \textit{tinfo-rep} \rangle tt) \\
\textit{prop } P_2 &= A(not\ (\langle \textit{tinfo-rep} \rangle tt) \\
&\quad W(\langle \textit{tinfo-req} \rangle tt) \\
\textit{prop } P_3 &= AG(not\ \langle \textit{tchange-m} \rangle tt) \\
&\quad \vee AF(\langle \textit{tchange-v} \rangle tt) \\
\textit{prop } P_4 &= A(not\ ((\langle \textit{tchange-v} \rangle tt) \\
&\quad W(\langle \textit{tchange-m} \rangle tt)))
\end{aligned}
$$

The above properties use additional observation actions *tinfo-req*, *tinfo-rep*, *tchange-m*, *tchange-v* corresponding to τ actions on the four links ($info\_req, \overline{info\_req}$), ($info\_rep$, $\overline{info\_rep}$), ($change\_m$, $\overline{change\_m}$) and ($change\_v$, $\overline{change\_v}$). The corresponding verifiable machine is given below.

$$
\begin{aligned}
Controller &= \textit{'change-m}.Controller \\
Model &= \textit{change-m.tchange-m.'change-v}.Model + \\
&\quad \textit{info-req.tinfo-req.'info-rep}.Model \\
View &= \textit{change-v.tchange-v.'info-req.info-rep} \\
&\quad \textit{.tinfo-rep}.View \\
Arch &= (Controller|Model|View)\backslash \\
&\quad \{\textit{change-m}, \textit{change-v}, \textit{info-req}, \textit{info-rep}\}
\end{aligned}
$$

V. Acceptor-Connector

In the acceptor-connector architectural pattern [15], connection establishment and service initialization is done before any processing is performed. This separation of responsibilities is achieved by three components, an acceptor, a connector, and a service handler. Before the client sends a request to server, a connection establishment protocol is initiated. Only when the server approves a connection establishment, it accepts the request from the client and replies to it. The connection establishment protocol involves a client side *connector* and a server side *acceptor* component as captured in the below CCS description.

A. Structural and Behavioral Specification

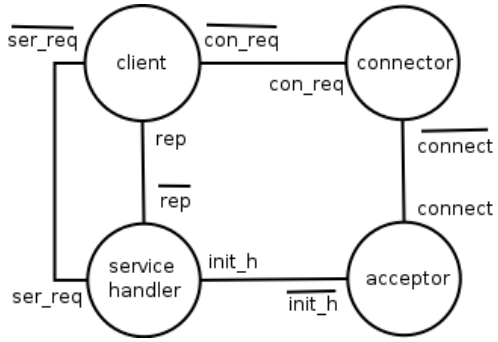

Fig. 3. Acceptor-Connector Pattern

$$
\begin{aligned}
Client &= \overline{\textit{con-req}}.\overline{\textit{serv-req}}.\textit{rep}.Client \\
Connector &= \textit{con-req}.\overline{\textit{connect}}.Connector \\
Acceptor &= \textit{connect}.\overline{\textit{init-h}}.Acceptor \\
ServiceHandler &= \textit{init-h}.\textit{serv-req}.\overline{\textit{rep}}.ServiceHandler \\
Arch &= Client|Connector|Acceptor| \\
&\quad ServiceHandler
\end{aligned}
$$

B. Verification Properties

Some of the properties of the acceptor-connector pattern are captured below.

- If there is a connection request, followed by service request from client, the client eventually receives a reply (P_1).
- It is not the case that client receives a reply without service request from the client (P_2).
- If there is connection request from the client, service handler will get initialized (P_3).
- If there is a connection request from the client, a connection request on the acceptor-connector link follows (P_4).
- If there is connection request from the client eventually service-handler will be initialized (P_5).
- There is a possibility of a connection request from client, followed by connect request generated from the connector, followed by a initialization of handler, request to server from the client and finally reply to the client. This sequence of actions may repeat infinitely(P_6).

$$
\begin{aligned}
\textit{prop } P_1 &= AG(not(\langle \textit{tcon-req}\rangle\langle t\rangle\langle \textit{tserv-req}\rangle tt) \\
&\quad \vee AF(\langle \textit{trep}\rangle tt)) \\
\textit{prop } P_2 &= A(not(\langle \textit{trep}\rangle tt) \\
&\quad W(\langle \textit{tserv-req}\rangle tt) \\
\textit{prop } P_3 &= AG(not(\langle \textit{tcon-req}\rangle tt \vee AF(\langle \textit{tinit-h}\rangle tt)) \\
\textit{prop } P_4 &= AG(not(\langle \textit{tcon-req}\rangle tt \vee AF(\langle \textit{tconnect}\rangle tt)) \\
\textit{prop } P_5 &= AG(not(\langle \textit{tconnect}\rangle tt) \vee AF(\langle \textit{tinit-h}\rangle tt)) \\
\textit{prop } p_6 &= \textit{max } X = \langle t\rangle\langle \textit{tcon-req}\rangle\langle t\rangle\langle \textit{tconnect}\rangle \\
&\quad \langle t\rangle\langle \textit{tinit-h}\rangle\langle t\rangle\langle \textit{tserv-req}\rangle\langle t\rangle\langle \textit{trep}\rangle X
\end{aligned}
$$

The above properties use observation actions *tserv-req*, *tcon-req*, *tconnect*, *tinit-h* and *trep*, respectively corresponding to links *client-service handler*, *client-connector*, *connector-acceptor*, *acceptor-service handler* and *client-service handler*. The verifiable machine including these actions is listed below.

$$
\begin{aligned}
Client &= \textit{'con-req.'serv-req.rep.trep}.Client \\
Connector &= \textit{con-req.tcon-req.'connect}.Connector \\
Acceptor &= \textit{connect.tconnect.'init-h}.Acceptor \\
ServiceHandler &= \textit{init-h.tinit-h.serv-req.tserv-req.'rep}. \\
&\quad ServiceHandler \\
Arch &= (Client|Connector|Acceptor| \\
&\quad ServiceHandler)\backslash\{\textit{con-req}, \textit{rep}, \\
&\quad \textit{connect}, \textit{tinit-h}, \textit{serv-req}\}
\end{aligned}
$$

VI. The Chain of Responsibility pattern

In the chain of responsibility pattern [5], the coupling between sender of a request to its specific receiver is avoided. A request send by the client is dropped into a chain of handler objects. A receiving object either handles the request or forwards it into the chain until an object handles it. The terminal handler always handles its incoming requests.

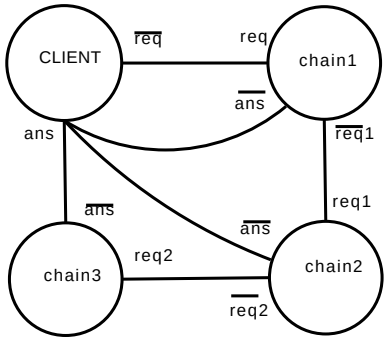

Fig. 4. Chain of Responsibility Pattern

1) Structural and Behavioral Specification:

$$
\begin{aligned}
Client &= \overline{req}.ans.Client \\
Chain_1 &= req.\overline{ans}.Chain_1 + req.\overline{req1}.Chain_1 \\
Chain_2 &= req1.\overline{ans}.Chain_2 + req1.\overline{req2}.Chain_2 \\
Chain_3 &= req2.\overline{ans}.Chain_3 \\
Arch &= Client|Chain_1|Chain_2|Chain_3
\end{aligned}
$$

2) Verification Properties: Some of the properties of the chain of responsibility pattern are as follows.

- If request is answered by chain-3 then no other object had answered the request (P_1).
- It is not the case that without a request from the client, It receives the answer (P_2).
- If request is answered by chain3 then all objects including the terminator receive the request earlier (P_3).

prop P_1	=	$(not\ \langle$*tans*$\rangle tt) \vee (\langle treq\rangle\langle treq1\rangle\langle treq2\rangle \langle tans\rangle tt)$
prop P_2	=	$A(not(\langle$*tans*$\rangle tt) W \langle$*treq*$\rangle tt)$
prop P_3	=	$(not\ \langle$*tans*$\rangle tt) \vee (\langle treq\rangle\langle treq1\rangle\langle treq2\rangle tt)$

The actions *treq*, *treq1*, *treq2* and *tans* are introduced as observable actions between connectors associated with ports req, $req1$, $req2$ and ans respectively. The verifiable machine that uses these actions is listed below.

$$
\begin{aligned}
Client &= {}'req.ans.tans.Client \\
Chain_1 &= req.treq.'ans.Chain_1 + \\
&\quad req.treq.'req1.Chain_1 \\
Chain_2 &= req1.treq1.'ans.Chain_2 + \\
&\quad req1.treq1.'req2.Chain_2 \\
Chain_3 &= req2.treq2.'ans.Chain_3 \\
Arch &= (Client|Chain_1|Chain_2| \\
&\quad Chain_3)\backslash\{req, req1, req2, ans\}
\end{aligned}
$$

VII. Publisher-Subscriber architectural description

A variation of the pattern that uses an intermediate event channels and a push-push model is modeled. A more general pattern of this kind can be found in the CORBA event service description [1]. Whenever a publisher publishes an event, the subscribers are required to receive a notification. Our assumption in modeling this architectural description is that subscribers are pre-subscribed.

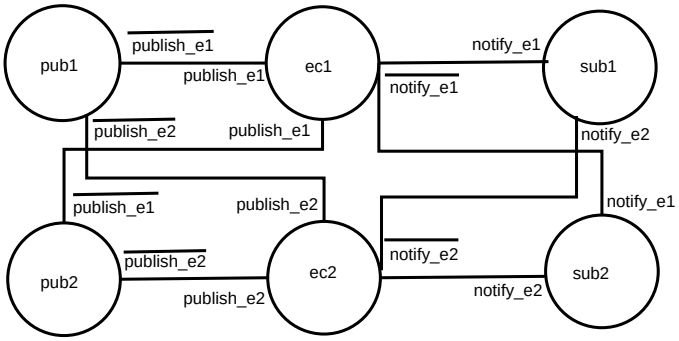

Fig. 5. Publisher Subscriber Pattern

3) Structural and Behavioral Specification:

$$
\begin{aligned}
Pub1 &= \overline{\textit{publish-e1}}.Pub1 + \overline{\textit{publish-e2}}.Pub1 \\
Pub2 &= \overline{\textit{publish-e1}}.Pub2 + \overline{\textit{publish-e2}}.Pub2 \\
Sub1 &= \textit{notify-e1}.Sub1 + \textit{notify-e2}.Sub1 \\
Sub2 &= \textit{notify-e1}.Sub2 + \textit{notify-e2}.Sub2 \\
Ec1 &= \textit{publish-e1}.\overline{\textit{notify-e1}}.Ec1 \\
Ec2 &= \textit{publish-e2}.\overline{\textit{notify-e2}}.Ec2 \\
\text{Arch} &= Pub1 \mid Pub2 \mid Sub1 \mid Sub2 \mid Ec1 \mid Ec2
\end{aligned}
$$

4) Verification Properties: Some interaction properties of the publisher subscriber pattern are described below. The publishers publish into the intermediate channel, and the channel generates notifications.

- After the publication of an event, it should be received by the subscribers (P_1).
- It is not the case that without publication, it is recieved by the subscribers (P_2).
- There exists a repeating sequence of publication of an event e1 and its notification (P_3).
- There exists a repeating sequence of publication of an event e2 and its notification (P_4).

prop P_1	=	$(not\ \langle$*tpublish-e1*$\rangle tt) \vee EF(\langle$*tnotify-e1*$\rangle tt)$
prop P_2	=	$A(not(\langle$*tnotify-e1*$\rangle tt)\ W(\langle$*tpublish-e1*$\rangle tt))$
prop p_3	=	*max X =* $\langle$*tpublish-e1*$\rangle\langle$*tnotify-e1*$\rangle X$
prop p_4	=	*max Y =* $\langle$*tpublish-e2*$\rangle\langle$*tnotify-e2*$\rangle Y$

The above properties use observable actions *tpublish-e1*, *tpublish-e2*, *tnotify-e1* and *tnotify-e2* corresponding to connector groups associated with ports *publish_e1*, *publish_e2*, *notify_e1* and *notify_e2*. The verifiable machine written in terms of these actions is provided below.

$$
\begin{aligned}
Pub1 &= \textit{'publish-e1}.Pub1 + \textit{'publish-e2}.Pub1 \\
Pub2 &= \textit{'publish-e1}.Pub2 + \textit{'publish-e2}.Pub2 \\
Sub1 &= \textit{notify-e1.tnotify-e1}.Sub1 \\
&\quad + \textit{notify-e2.tnotify-e2}.Sub1 \\
Sub2 &= \textit{notify-e1.tnotify-e1}.Sub2 \\
&\quad + \textit{notify-e2.tnotify-e2}.Sub2 \\
Ec1 &= \textit{publish-e1.tpublish-e1.'notify-e1}.Ec1 \\
Ec2 &= \textit{publish-e2.tpublish-e2.'notify-e2}.Ec2 \\
\text{Arch} &= (Pub1 \mid Pub2 \mid Sub1 \mid Sub2 \mid Ec1 \mid Ec2)\backslash \\
&\quad \{\textit{publish-e1}, \textit{publish-e2}, \textit{notify-e1}, \textit{notify-e2}\}
\end{aligned}
$$

VIII. Dining Philosopher Problem

The dining philosopher problem is modeled using CCS. CCS model is presented that simulates the behavior of two philosophers. They are gathered around a table to think and eat. Each philosopher thinks for a while, then eats, then thinks again, and so on, independently of the others. When a philosopher wants to eat, he picks the fork on his left, if it's available, then the fork on his right, eats, and then puts both forks back. It may happen that both philosophers simultaneously want to eat. They pick their left fork, and find the right fork already picked by their right neighbor. In this case the philosophers may stay in that situation (deadlock) indefinitely.

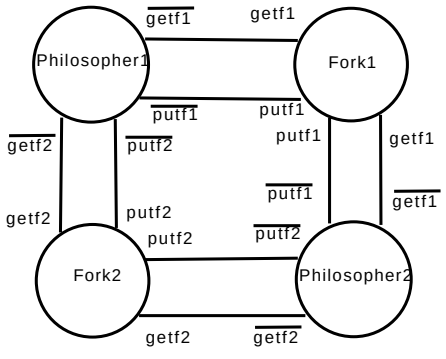

Fig. 6. Dinning Philosopher Problem

A. Structural and Behavioral Specification

$$
\begin{aligned}
P_1 &= \overline{getf_1}.\overline{getf_2}.\overline{putf_1}.\overline{putf_2}.P_1 \\
P_2 &= \overline{getf_2}.\overline{getf_1}.\overline{putf_2}.\overline{putf_1}.P_2 \\
F_1 &= getf_1.tget_1.putf_1.F_1 \\
F_2 &= getf_2.tgetf_2.putf_2.F_2 \\
dpp &= (P_1|P_2|F_1|F_2)\backslash \\
&\quad \{getf_1, getf_2, putf_1, putf_2\}
\end{aligned}
$$

1) Verification Properties: Two important properties for dining philosopher problem are as given below.

- The mutual exclusion property ensures that the fork is not assigned to two philosophers at the same time (P_1).
- Deadlock property checks whether there is deadlock in the system. (P_2).

$$
\begin{aligned}
\textit{prop } p_1 &= AG(\neg(\langle tgetf_1\rangle\langle tgetf_1\rangle tt) \\
&\quad \wedge\neg(\langle tgetf_2\rangle\langle tgetf_2\rangle tt)) \\
\textit{prop } p_2 &= \textit{min } X = [-]ff \vee \langle - \rangle X
\end{aligned}
$$

Specifying Invariants for Dining Philosophers

The above properties use observable actions *tgetf*$_1$, *tgetf*$_2$, corresponding to connector groups associated with ports *tgetf*$_1$, *tgetf*$_2$. The verifiable machine written in terms of these actions is provided below.

$$
\begin{aligned}
P_1 &= \overline{getf_1}.\overline{getf_2}.\overline{putf_1}.\overline{putf_2}.P_1 \\
P_2 &= \overline{getf_2}.\overline{getf_1}.\overline{putf_2}.\overline{putf_1}.P_2 \\
F_1 &= getf_1.tgetf_1.putf_1.F_1 \\
F_2 &= getf_2.tgetf_2.putf_2.F_2 \\
dpp &= (P_1|P_2|F_1|F_2)\backslash \\
&\quad \{getf_1, getf_2, putf_1, putf_2\}
\end{aligned}
$$

IX. Conclusion

Variants of Proxy, Chain, MVC, Acceptor-Connector, Publisher-Subscriber and dinning philosopher patterns have been modeled in the CCS process calculus with components as agents and inter-component interactions as a result of the composition with restriction. Non-determinism in CCS can be used to bring out alternative interaction sequences among the components. Modal μ calculus was used for specifying the desirable interaction properties of the system.

With the restriction operator in place, the internal actions become unobservable and hence for specifying the verification properties in terms of the τ actions, additional corresponding observation actions were introduced in the CCS models. The additional actions were introduced as input actions with corresponding input ports being unrestricted. With this technique, the number of labels used in the properties becomes the same as the number of types of links present in the CCS model.

The CCS models precisely capture the interfaces of various components in an abstract sense through the ports, and the interactions among them through input and output actions.

References

[1] Event service specification, object management group, October 2004.

[2] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, 1997.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[4] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench. In *Proceedings of the international workshop on Automatic verification methods for finite state systems*, pages 24–37, New York, NY, USA, 1990. Springer-Verlag New York, Inc.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] 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 1–39, Singapore, 1993. World Scientific Publishing Company.

[7] M. M. Kandé and A. Strohmeier. Towards a UML profile for software architecture descriptions. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 513–527. Springer, 2000.

[8] D. Kozen. Results on the propositional mu-calculus. *Theor. Comput. Sci.*, 27:333–354, 1983.

[9] C. Lange, M. Chaudron, and J. Muskens. In practice: Uml software architecture and design description. *IEEE Software*, 23(2):40–46, March-April 2006.

[10] J. Magee and J. Kramer. Dynamic structure in software architectures. *SIGSOFT Softw. Eng. Notes*, 21(6):3–14, 1996.

[11] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Trans. Softw. Eng.*, 26(1):70–93, 2000.

[12] R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.

[13] D. E. Perry and A. L. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992.

[14] T. L. Rance Cleaveland and S. Sims. The concurrency workbench of the new century, user's manual. June 6, 2000.

[15] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.

[16] C. Stirling. An introduction to modal and temporal logics for ccs. In *Proceedings of the UK/Japan workshop on Concurrency : theory, language, and architecture*, pages 2–20, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[17] C. Stirling. *Modal and temporal properties of processes*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.