Automatic synthesis of coordinators for COTS group-ware applications: an example

Paola Inverardi
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
inverard@di.univaq.it

Massimo Tivoli
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
tivoli@di.univaq.it

Antonio Bucchiarone
University of L'Aquila
Dip. Informatica
via Vetoio 1, 67100 L'Aquila
bucchiarone@di.univaq.it

Abstract

The coordination of concurrent activities in collaborative environments is a very important and difficult task. Many approaches for the construction of large-scale flexible group-ware applications there exist in the literature. They provide valid support to modelling, analysis and to a white-box developing of coordination protocols for computer supported cooperative applications. Little attention has been dedicated so far to group-ware applications built by assembling third-party components. In this paper by means of an explanatory example, we apply a software architecture based approach to the group-ware systems development. The software architecture imposed on the coordinating part of the system, allows for detection and recovery of possible and unpredictable concurrent activities conflicts. Moreover, the approach allows the enforcing of coordination policies on the composed system by automatically synthesizing the policy-satisfying assembly code.

1 Introduction

Computer Supported Cooperative Work (CSCW) [21] is a multi-disciplinary research area mainly focused on effective methods of sharing information and coordinating concurrent activities. The coordination of these activities is a very important and difficult task. Many approaches for the construction of large-scale flexible (in coordination) group-ware applications there exist in the literature. Most of them are inspired by coordination languages and models [2, 18] which propose the separation of computation from coordination for multi-threaded applications. These approaches are mainly concerned with design and coordination architectural models [19, 13, 16], coordination protocols [6] and languages [3]. They provide support for developing flexible group-ware applications, specifying and controlling the interaction among collaborative activities, evaluating the behavior of a CSCW system before its implementation, modelling, analyzing and prototyping the coordination policies of a collaboration system, and designing CSCW systems. These approaches work in a white-box components setting and are flexible in terms of supported controlled coordination policies. On the other hand they cannot straightforwardly be exported in a black-box component setting. When we deal with group-ware applications built by assembling third-party (*COTS: Commercial-Off-The-Shelf*) components which are truly black-box, the issue is not only in specifying and analyzing a coordination policy rather in being able to enforce it out of a set of already implemented (local) behaviors. In a black-box components setting, one of the challenges is related to the ability to predict possible coordination policies of the components interaction behavior by only exploiting a limited knowledge of the single components computational behavior.

In this paper we propose to apply a software architecture based approach [8, 9] to the development of COTS-based group-ware applications. It is based on our framework [8, 9] for correct and automatic synthesis of software connectors. The framework, given a set of COTS components, allows for COTS components assembly and interaction with respect to specified behavioral properties. In the context of this paper we only consider behavioral properties that model coordination policies. Our approach puts together COTS-systems by assuming a well defined architectural style [9] in such a way that it is possible to detect and to fix coordination anomalies. We assume that a specification of the desired assembled system is available and that a precise definition of the coordination policies to enforce exists. The coordination policies are specified in terms of behavioral properties of the composed system. With these assumptions we are able to develop a

Proceedings of the Twelfth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE'03)
1080-1383/03 $17.00 © 2003 IEEE

framework that automatically derives the assembly code for a set of components so that, if possible, a coordination policy-satisfying system is obtained. The assembly code implements an explicit software connector which mediates all interactions among the system components as a new component to be inserted in the composed system. The connector can then be analyzed and modified in such a way that the specified coordination policies are enforced.

The paper is organized as follows. Sections 2 and 3 introduce background notions and briefly summarize the method concerning the synthesis of connectors that are conflicts-free and coordination policy-satisfying. In Section 4 we apply the approach to an instance of a typical CSCW application, that is a collaborative writing (CW) system we have designed. Section 5 discusses future work and concludes.

2 Background

In this section we provide the background needed to understand the approach described in Section 3.

2.1 The reference architectural style

The architectural style we use, called *Connector Based Architecture* (CBA), consists of components and connectors which define a notion of top and bottom. The top (bottom) of a component may be connected to the bottom (top) of a single connector. Components can only communicate via connectors. It is disallowed the direct connection between connectors. Components communicate synchronously by passing two type of messages: notifications and requests. A notification is sent downward, while a request is sent up. Connectors are responsible for the routing of messages and they exhibit a strictly sequential input-output behavior[1]. The CBA style is a generic layered style. The example treated in Section 4 is related to a single-layer system. In [9] we show how to cope with multi-layered systems.

2.2 Configuration formalization

In order to describe components and system behaviors we use CCS [17] (*Calculus of Communicating Systems*) notation. Our framework allows the automatic derivation of these CCS descriptions from *"HMSC (High level Message Sequence Charts)"* and *"bMSC (basic Message Sequence Charts)"* [1] specifications of the system. This derivation step is done by applying a suitable version of an existent translation algorithm from bMSCs and HMSCs to LTSs (*Labelled Transitions Systems*) [20]. These kinds of specifications are common practice in real-scale contexts thus CCS can be merely regarded as an internal to the framework specification language. Since HMSC and bMSC specifications model finite-state behaviors of a system we will use finite-state CCS.

[1] Each input action is strictly followed by the corresponding output action.

3 Approach description

We are assuming that a specification of the system to be assembled is provided. This specification is given in terms of bMSCs and HMSCs descriptions. As said in Section 2.2, from bMSCs and HMSCs description we automatically derive a description of each component's behavior as finite-state CCS term (i.e. LTS). Moreover we assume that a specification of the coordination policies to be enforced exists. This specification is given in LTL (*Linear-time Temporal Logic*) [4]. In the following, we first discuss our method proceeding in three steps as illustrated in Figure 1 and then, in Section 4, we apply the method to our working example.

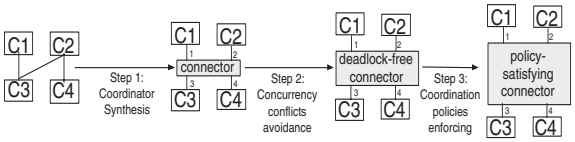

Figure 1. 3 step method

The first step builds a connector (i.e. the coordinator) following the CBA style constraints. The second step performs the deadlocks detection and recovery process. This step restricts the behavior of the synthesized coordinator in order to avoid concurrency conflicts. Finally, the third step performs the check of the specified coordination policies against the deadlock-free connector and then synthesizes a coordination policy-satisfying connector. From the latter we can derive the code implementing the coordinator component which is by construction correct with respect to the coordination policies specification. We refer to [8, 9] for a formal description of the whole approach.

4 Application: a semi-synchronous CW system

Collaborative writing is one discipline of the CSCW research area. Collaborative writing is defined in [14] as: *'the process in which authors with different expertise and responsibilities interact during the invention and revision of a common document'*. In this section we apply our approach in order to build from a set of suitable COTS components a CW system [7, 14]. Based on a detailed analysis [10] of many CW systems [12, 11, 5, 15] we can identify the COTS computational components that provide the main features of

a CW system. These four types of COTS components are showed in Figure 2.

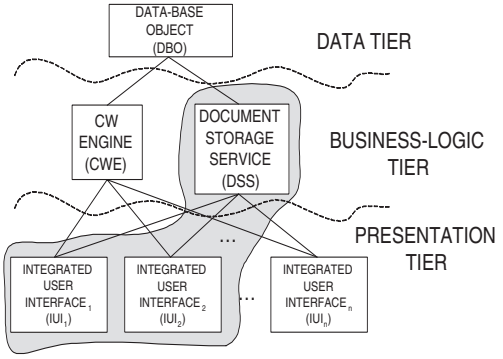

Figure 2. Architecture of the CW system

Our CW system is a three-tier application. According to our approach, it is composed through coordinators components automatically synthesized in order to satisfy a specified coordination policy. In the following, we briefly describe our CW system and apply our approach to a subsystem of it (see the grey area in Figure 2). A detailed description of the CW system and of the complete application of the approach to it can be found in [10].

In order to build our CW system we have identified the following four COTS components. 1) **DBO**: This component is a data-base. The data-base stores all group awareness information useful to support a group activity. 2) **CWE**: This component is a CW engine; actually it provides all services useful to perform a group activity in the CW context. It is a handler of all group awareness information stored in *DBO* and of the typical CW activities [10]. 3) **DSS**: A document is a set of document's partitions. This component is an abstraction of the physical container of the shared documents that are logically partitioned according to their structure. In an asynchronous working mode we use version-controlled documents. In a synchronous working mode it is shared among the users and we have to use pessimistic concurrency control. Referring to the version-controlled hierarchical documents [15], a local copy of a document is an alternative and the globally shared document is the last document's version. When a user wants to work in asynchronous mode, the *DSS* expects that all other users work in asynchronous mode too. In this way the *DSS* can maintain a consistent version of the globally shared document and it evolves in a new consistent version only after the merging of all users alternatives. 4) **IUI**: This component is an integrated environment of tools for editing, navigation, display of awareness communication among the group members and import and export of data from external applications. It is composed of a CW user interface supporting all CW operations, editors for many data types, communication tools such as e-mail and chat.

Let us suppose that the designer of the composed CW system provides a behavioral specification in terms of bMSCs and HMSCs see Figures 3 and 4. The continued lines in the bMSCs are method calls; the hatched lines are the corresponding responses. In our example we consider only two instances of IUI_i: IUI_1 and IUI_2. Moreover we only provide the system's behavioral specification for the part of the CW system identified in the grey area of Figure2. This subsystem is composed by components IUI_1, IUI_2 and *DSS*.

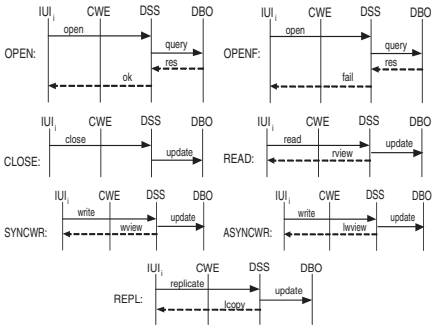

Figure 3. bMSCs of OPEN, OPENF, CLOSE, READ, SYNCWR, ASYNCWR and REPL scenarios

In Figure 3 we show the bMSCs representing the *'open work session'*, the *'close work session'*, the *'data displaying'*, the *'data synchronous updating'*, the *'data asynchronous updating'* and the *'data replication for asynchronous writing'* scenario.

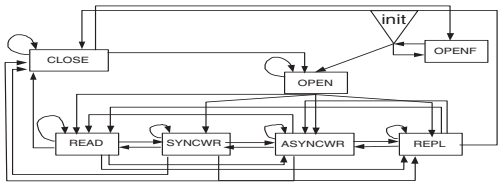

Figure 4. HMSC of the CW sub-system

In Figure 4 we show the HMSC specification for the composed CW sub-system. Let us suppose that the designer of the CW system wants that the composed system satisfies a particular coordination policy. The policy is specified in form of the following behavioral LTL property: $P_1 = G(F(write_1)) \wedge G(F(write_2))$. P_1 is a general progress property related to a writing scenario. It means that who requires to commit an update on the shared document will

eventually always be able to perform this update. From the HMSC and bMSCs specification we can automatically derive the behavioral models (i.e. AC-Graphs [10, 9]) for each component in our CW sub-system (see Figure 5).

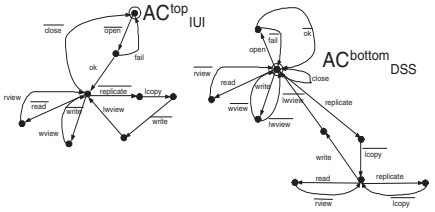

Figure 5. AC-Graphs of components IUI_1, IUI_2 and *DSS*

AC-Graphs model components behavior in terms of interactions with the external environment. The double-circled states are the initial states. The overlined actions are output actions, the others are input actions. From the AC-Graphs of IUI_1, IUI_2 and *DSS* we derive the corresponding EX-Graphs [10, 9] (see Figure 6).

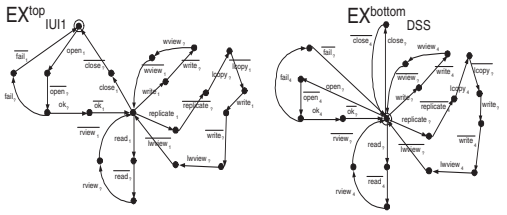

Figure 6. EX-Graphs of components IUI_1, IUI_2 and *DSS*

Each EX-Graph represents a partial view (i.e. the single component's view) of the connector behavior. The EX-Graph for component C_i (i.e. EX_i) is the behavior that C_i expects from the connector. Thus EX_i has either transitions labelled with known actions or with unknown actions for C_i. Known actions are performed on the channel connecting C_i to the connector. This channel is known to C_i and identified by a number. Unknown actions are performed on channels connecting other components C_j $(j \neq i)$ to the connector, therefore unknown from the C_i perspective. These channels are identified by the question mark. Referring to Figure 6, the EX-Graph of IUI_2 is different from the EX-Graph of IUI_1 only in the identifier of the channel specified in known actions labels (2 instead of 1). We derive the connector global behavior through an EX-Graphs unification algorithm (refer to [9]). In this paper, for the sake of presentation, we only show a sub-graph of the connector global behavior graph (see Figure 7) and we reduce the analysis of the whole connector to the sub-graph $K_{1.1}$ of the connector global behavior graph. Refer to [10] for a complete visualization of the connector graph.

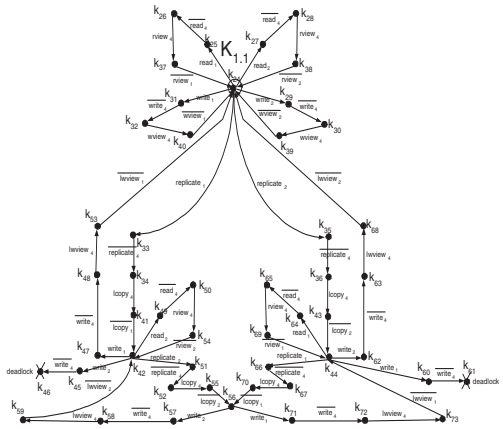

Figure 7. Sub-graph $K_{1.1}$ of the global connector graph

The sub-connector $K_{1.1}$ has two concurrency conflicts (i.e. deadlocks) represented by two finite branches. These deadlocks are related to the consistency maintenance in an asynchronous writing scenario. We recall that in order to maintain a consistent version of the shared document, the *DSS* expects that all users work in asynchronous mode every time another user chooses to work in asynchronous mode. The third-party components IUI_1 and IUI_2 do not respect this *DSS* assumption. Thus the composed system has concurrency conflicts. This puts in evidence a typical problem of when assembling COTS components. In order to synthesize the deadlock-free version of $K_{1.1}$ we simply prune the two finite branches. The deadlock-free $K_{1.1}$ forces IUI_1 and IUI_2 to respect the *DSS* assumption. Once obtained the deadlock-free version of the connector, the approach goes to the coordination policy enforcing step [10]. Our approach builds the Büchi automaton [4] of P_1 (i.e. B_{P_1}) and the Büchi automaton of the deadlock-free $K_{1.1}$ (i.e. $B_{K_{1.1}}$). In Figure 8 B_{P_1} is shown. The double-circled state is the accepting state of the automaton (i.e. p_2) and the state with an incoming arrow is the initial state (i.e. p_0).

$B_{K_{1.1}}$ is equal to $K_{1.1}$ with all nodes marked as accepting states. Then, the method builds the automaton $B^{K_{1.1},P_1}_{intersection}$ which accepts the language $L(B_{KS_K}) \cap L(B_P)$. It represents the P_1-satisfying behaviors of the deadlock-free $K_{1.1}$. Finally in order to synthesize the *"fair"*[2] P_1-

[2] With respect to starvation property.

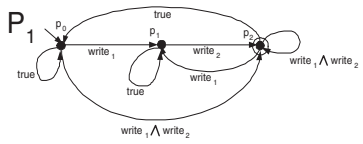

Figure 8. Büchi Automaton corresponding to P_1

satisfying deadlock-free connector, our method extracts the sub-automaton of $B^{K_{1.1},P_1}_{intersection}$ that contains as execution paths only *accepting cycles* [10] of $B^{K_{1.1},P_1}_{intersection}$. Informally, an accepting cycle is a cycle containing an accepting state. In Figure 9 we have shown this sub-automaton. It corresponds to the behavioral model for the P_1-satisfying starvation and deadlock-free $K_{1.1}$.

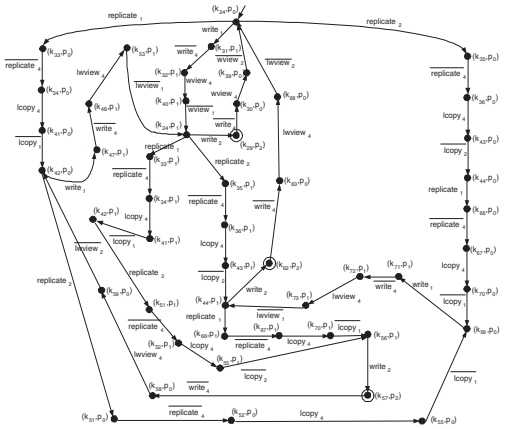

Figure 9. Accepting cycles of the Büchi Automaton accepting $L(B_{K_{1.1}}) \cap L(P_1)$

This behavioral model is enough to derive the conflicts-free policy-satisfying connector code that implements the connector methods related to the LTL policy specification. For all the others methods (i.e. the methods that have not influence on the coordination policy) the connector is a simple delegator. In the following we show the conflict-free policy-satisfying code implementing the method ***write*** of the connector component $K_{1.1}$ (see Figure 10). We refer to [10], for the complete implementation of $K_{1.1}$. The implementation refers to Microsoft COM (*Component Object Model*) components and uses C++ wiht ATL (*Active Template Library*) as programming environment. The method ***write*** of the inner *DSS* object gets a parameter of type ***S_DA***. ***S_DA*** is a document alternative struct. It contains information about the document update to be committed.

```
HRESULT write(S_DA da) {
   if(sLbl == 24) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 24;
      }
      else if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 24;
      }
   }
   else if(sLbl == 56) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 44;
      }
      else if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 42;
      }
   }
   else if(sLbl == 42) {
      if((chId == 1) && (pState == 0)) {
         return dssObj->write(da);
         pState = 1; sLbl = 24;
      }
   }
   else if(sLbl == 44) {
      if((chId == 2) && (pState == 1)) {
         return dssObj->write(da);
         pState = 0; sLbl = 24;
      }
   }

   return E_HANDLE;
}
```

Figure 10. Synthesized implementation of "write" method

This code is automatically synthesized by visiting the sub-automaton of Figure 9 and by exploiting the information stored in its states and transitions labels. The connector component $K_{1.1}$ is an aggregated server component that encapsulates an instance of the inner *DSS* component.

5 Conclusion and future work

In this paper we have described a connector-based architectural approach to component assembly. We have applied it in order to automatically synthesize coordinators components for COTS group-ware applications. Our approach focusses on detection of concurrent activities conflicts and on enforcing coordination policies on the assembly. A key role is played by the software architecture structure since it allows all the interactions among components to be explicitly routed through a synthesized connector. The enforced coordination policies are encapsulated in a connector component (i.e. the coordinator) which provides the coordinating code. The software architecture imposed on the composed system allows for easy replacement of a connector with another one in order to make the whole system flexible with respect to different coordination policies. The approach is oriented to coordination protocols enforcing besides coordination protocols analysis.

On the negative side, the approach, as it is, suffers the state-space explosion problem. Another possible limit is that the coordinator logic and its implementation is

completely centralized. Actually this is not a real limit because even if we centralize the coordinator logic we can derive a distributed implementation of it.

Future work goes in two directions: foundational and practical. Foundationally we are studying techniques and formal methods in order to reduce the state-space explosion problem. These techniques are related to automata-based model checking [4] for the implementation of on-the-fly connector synthesis. Moreover we are studying the applicability of partial order reduction techniques and of compositional reasoning techniques as, for example, the assume-guarantee paradigm. Practically, we are implementing a comprehensive framework to support the automation of the whole approach and the integration with other tools. The analysis of human cost and of the user interaction degree which have great influence on the framework's applicability has to be considered. To this respect it would be interesting to specify a coordination policy in a more user-friendly manner with the usage of a suitable extended HMSCs and bMSCs notation.

Acknowledgements

This work has been partially supported by Progetto MIUR SAHARA.

References

[1] Itu telecommunication standardisation sector, itu-t reccomendation z.120. message sequence charts. (msc'96). Geneva 1996.

[2] F. Arbab. Coordination of massively concurrent activities. In *141, Centrum voor Wiskunde en Informatica (CWI), http://citeseer.nj.nec.com/arbab95coordination.html*, page 31, 30 1995.

[3] M. Corts and P. Mishra. An implementation model for collaborative applications. In *Proceedings of CLEI'96 (Bogota, Colombia)*, volume 1, June 1998.

[4] O. G. Edmund M. Clarke, Jr. and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, London, England, 2001.

[5] C. A. Ellis and S. J. Gibbs. Concurrency control in groupware systems. In *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 399–407, 1989.

[6] R. Furuta and P. D. Stotts. Interpreted collaboration protocols and their use in groupware prototyping. In *Computer Supported Cooperative Work, http://citeseer.nj.nec.com/furuta94interpreted.html*, pages 121–131, 1994.

[7] J. Grudin. Computer-supported cooperative work: History and focus. *In IEEE Computer Journal*, 27(5):19–26, 1994.

[8] P. Inverardi and M. Tivoli. Automatic synthesis of deadlock free connectors for com/dcom applications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE, ACM Press*, Vienna, Sep 2001.

[9] P. Inverardi and M. Tivoli. Failures-free connector synthesis for correct components assembly. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/FCSforCCA.pdf, submitted for publication*, ITALY, February 2003.

[10] P. Inverardi, M. Tivoli, and A. Bucchiarone. Coordinators synthesis for cots group-ware systems: an example. *Technical Report, University of L'Aquila, Department of Computer Science, http://www.di.univaq.it/tivoli/cscw_techrep.pdf*, ITALY, March 2003.

[11] M. Koch. Design issues and model for a distributed multi-user editor. *Computer Supported Cooperative Work, International Journal*, 5(1), 1996.

[12] M. Koch and J. Kock. Using component technology for group editors - the iris group editor environment. In *In Proc. Workshop on Object Oriented Groupware Platforms*, pages 44–49, Sep 1997.

[13] Y. Laurillau and L. Nigay. Clover architecture for groupware. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work, New Orleans, Louisiana, USA, http://doi.acm.org/10.1145/587078.587112, ACM Press*, pages 236–245, 2002.

[14] M. Lay and M. Karis. Collaborative writing in industry: Investigations in theory and practice. *Baywood Publishing Company, Amityville*, 1991.

[15] B. G. Lee, K. H. Chang, and N. H. Narayanan. A model for semi-(a)synchronous collaborative editing. In *Proceedings of the Third European Conference on Computer Supported Cooperative Work, ECSCW 93*, pages 219–231, September 1993.

[16] D. Li and R. Muntz. Coca: Collaborative objects coordination architecture. In *Proceedings of ACM Conference on Computer Supported Cooperative Work, Seattle*, pages 179–188, 1998.

[17] R. Milner. *Communication and Concurrency*. Prentice Hall, New York, 1989.

[18] G. A. Papadopoulos and F. Arbab. Coordination models and languages. In *Centrum voor Wiskunde en Informatica, http://citeseer.nj.nec.com/article/papadopoulos98coordination.html*, 1998.

[19] A. B. Raposo, L. P. Magalhaes, and I. L. M. Ricarte. Coordinating activities in collaborative environments: A high level petri nets based approach. In *SCI 2000 - 4th World Muticonference on Systemics, Cybernetics and Informatics Proceedings. Orlando, USA. International Institute of Informatics and Sytemics (IIIS), 2000. Selected the best paper of session Management Information Systems III*, volume 1, pages 195–200, 2000.

[20] S. Uchitel, J. Kramer, and J. Magee. Detecting implied scenarios in message sequence chart specifications. In *ACM Proceedings of the joint 8th ESEC and 9th FSE*, Vienna, Sep 2001.

[21] A. I. Wang, R. Conradi, and C. Liu. A multi-agent architecture for cooperative software engineering. *SEKE97, Kaiserslautern, Germany*, June 1997.