Enhancing Architectural Mismatch Detection with Assumptions *

Sebastián Uchitel, Daniel Yankelevich
Dpto. de Computación, Universidad de Buenos Aires
{suchitel,dany}@dc.uba.ar

Abstract

Detecting software architecture inconsistencies is a critical issue in software design. Software systems are described in terms of components, component behavior and interaction and mismatch detection is explored through techniques based on behavior analysis. Integration problems, however, are not only caused by behavioral mismatch: components make assumptions about their environment to guarantee functional and non-functional properties. If the actual deployment environment of each component does not satisfy its assumptions, component and system properties may not hold. In this work we propose to extend the idea of architectural mismatch to include the notion of assumption. We concentrate on a subset of possible assumptions and show how software architects can benefit from using them. We also present a discussion on how architecture description languages (ADLs) can be extended to include assumptions.

1. Introduction

Detecting software architecture inconsistencies, such as incompatible components, is a critical issue in software design, and there has been significant work in the Software Architecture (SA) community in this direction. Software systems are described in terms of components, component behavior and interaction [20] and mismatch detection is explored through techniques based on behavior analysis [16, 8, 2]. *Connectors* and *connections* have a vital role in mismatch detection because they model how components are being combined in a software system. Issues as if *connectors* should be explicitly represented in architectures or not, whether they are first class entities [2] or what information should they convey [20] have been discussed extensively. However, in these discussions, it is assumed that connections are control transfer points, parameter passing or shared data [20]. As Parnas pointed out in 1972, *such a definition of connection is a highly dangerous oversimplification (sic). The connections between components are the assumptions which the components make about each other* [18].

Many examples can be given to show that component assumptions can be critical at an architectural level analysis. For example,

- Transaction servers or security supervisors usually assume that all accesses to a database server are done through them. It is not enough to check if components that interact with them follow the expected communication protocol, because the violation of their assumptions may make their functionality useless. For instance bypassing a transaction server may prevent it from providing consistent transactions or 100% recovery on failures.

- X-Windows components make important assumptions on the number and distribution of simultaneously active windows and although too many active windows do not cause a change in how X-Windows services are required (e.i. interaction is not affected), performance may drop to an unacceptable level. As presented in [10] the idea of implementing a spreadsheet using an X-Windows component by implementing each cell as a window is sound in terms of interfaces and interaction mechanisms, however taking into account the assumptions of an X-Window component, the approach makes no sense.

- Real time components make many assumptions on the network reliability and latency and can only guarantee some properties if these assumptions are kept.

- Coordinating agents make assumptions on version numbers in mobile systems in order to guarantee quality of certain distributed services.

*partially supported by ANPCyT, project ARTE BID-PICT-1856 and by UBACyT, project ARTE TW72

The point is that all these properties, secure transactions, reasonable performance, compliance to real time deadlines and quality of service, are granted because they have been proved or tested up to certain hypothesis. Therefore, these hypothesis or in other words assumptions, are important and must be kept.

In this paper we propose to extend the idea of architectural mismatch. We show that it is important to analyze not only if components interact together correctly but also if all component assumptions are not violated by their environments. We concentrate on a specific subset of possible assumptions and show in Section 3 by means of two examples how software architects can benefit from using assumptions to enhance architectural descriptions and detect mismatches that would go undetected if only interaction information is used. In Section 4 instead of proposing yet another architecture description language (ADL), we discuss how existing ADLs can be extended to include the ideas presented in this paper. Finally in 5 with conclusions and future work.

2. Related work

The term *assumption* has been used by the software engineering community in different ways. Our perspective of this term is borrowed from [18] where connections are thought of as assumptions that components make about each other, instead of control or information transfer points. Our view is that assumptions extend the idea of connection as a control or information transfer point, that interaction and assumptions can be combined to enhance architectural mismatch detection.

The idea of using assumptions as behavior abstractions can be found in work by Allen [1] on the architecture description language (ADL) Wright [20]. In Wright components interact with their environment through ports that relate to roles, which are connector interaction points. The architect defines the component, connector, port and role semantics using a subset of the language CSP [7] taking into account that a specification is consistent only if the port/role specification is an abstraction of the components/connectors behavior. Although ports and roles were introduced to support connector reuse, ports and roles can be seen as the interaction a component or connector assumes there will be through that particular port or role. Once connector-role and component-port consistency has been checked, some interaction mismatches can be detected analyzing abstracted behavior only. This approach differs from our work, from our perspective component assumptions are not abstractions of their own behavior, abstractions add information to behavioral specification. In Section 4 we exemplify the problems that arise trying to force our view of assumptions in a Wright setting.

Other ADLs [19, 17, 5, 14, 21, 15] are also oriented towards describing the notion of information exchange of control flow through connectors and do not approach the idea of component assumptions. In Section 4 we discuss the relation between existing ADLs and assumptions more extensively.

Cheung and Kramer [12] mention how assumptions can be introduced into hierarchical composition of components by means of user-specified interfaces. Although interfaces are not limited to interaction through one specific port as in the Wright approach, they still are tied to the concept of architectural behavior: User-specified interfaces can be included at a certain level of the composition hierarchy as long as they include all possible behaviors of their *peers*. Assumptions are not general system properties as their approach suggests, assumptions are an important part of a component description therefore from a hierarchical point of view the behavior and assumptions of a component should be on the same level. This would not be possible with the user-specified interface approach as the assumptions of a component need not be an abstraction of the component behavior.

The term assumption is used differently in [9]. The main idea is to use assumptions to verify specific properties such as deadlock freedom in a tractable manner. Assumptions are derived from actual behaviors taking into account the property to be proved, and then checked without constructing the complete system model. Although the general idea of assumptions is the same, in this approach assumptions are given explicitly by users to guarantee a set of ad hoc properties while in [9] assumptions are generated from behaviors using the knowledge of the property to be verified.

The object orientated community has been working for some time on design patterns [4]. Design patterns describe interaction mechanisms for solving specific problems. Each problem can be considered a description of the environment for which the pattern is suitable. These restrictions on the environment of the pattern resemble our idea of assumptions but applied to a set of components with a specific interaction mechanism.

Within the MetaObject Protocol (MOP) community, the context in which a software component will be used is given a fundamental role [11]. The main idea is that a component is designed and implemented with certain assumptions over its context and intended use. Many times these do not match the clients intended use, therefor, the MOP community proposes that objects should provide ways for their clients to modify their behaviors to better suit their needs. The bottom line is that context and intended use (e.i. assumptions on components environment) are important for system construction. In a way, the MOP approach can be seen as the dual of ours: they are interested in implementations that allow modifying component assumptions while we are interested in describing assumptions relevant to software ar-

chitecture.

3. Examples

To motivate discussion, we present two examples, a truck tracking system and a gas station system. We will use in this section two types of diagrams. For representing the behavior of components in terms of their interaction we will use labeled transition systems where states represent component states, arcs represent potential state changes of a component and arc labels for the actions that a component must perform when changing state. For representing a system architecture we will use a hierachy of subsytems as in [13]. Subsystems, represented as boxes with rounded corners, are formed by the parallel composition of their descendants, which can be subsystems or simply components (represented with rectangles). For a formal definition of labeled transition systems and parallel composition refer to Appendix A.

3.1. A truck tracking system example

A freight transportation and logistics company needs a system that can keep track of its truck fleet, reporting on positions of individual trucks and general statistics. The approach it takes in the construction of such a system is a commercial off-the-shelf (COTS) one. The company decides to buy a geographical information server that is capable of receiving information from remote mobile entities, process it and respond to queries (e.i. truck nearest to an unscheduled delivery point). In addition a control component is to be developed for administration purposes and to process certain data according to particular company requirements.

The server controls the status of a truck distribution fleet by providing a service named *Register*. Trucks *Register* periodically to the server and receive an acknowledge each time. One of the tracking service provided by the server allows clients to receive a stream of *Positions* (in terms of street names), a client issues *Start* and *Stop* commands to access this service. In addition, this server, as many servers do, requires maintenance. Periodically, a maintenance service *Purge* must be launched so all old *Register* data can be deleted. If the server is not Purged often enough, performance, and therefore accuracy, of the *Positions* stream degrades quickly. Once the server performs the administration tasks launched by *Purge* it returns a status report *Done*. There are many reasons a server of these characteristics may not include automatic administration features, for example, the resource usage of administration tasks may lower performance temporarily, thus the decision of when to launch them could be done more effectively by a user. Figure 1 describes the behavior of the server in terms of three processes, one for each service provided by the server. These processes are composed to describe the overall server behavior. In Figure 2 we show the complete server behavior as the composition of its three service processes.

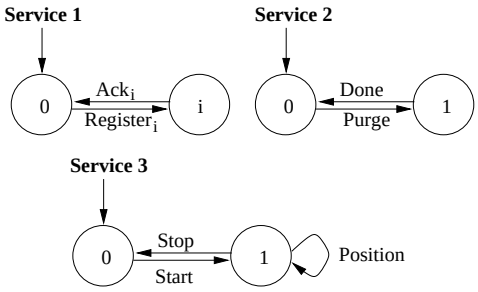

Figure 1. Behavior of the server processes

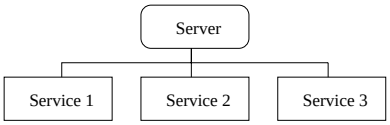

Figure 2. Behavior of the server component

It is clear that our model of the server is so far incomplete. If this model were the only information provided by the COTS manufacturer, it would be insufficient for assembling properly a system based on the server component. The reason is that there is a strong requirement for user intervention for maintenance purposes. This requirement clearly is not an interaction issue as it does not impose a strict behavior of server environment, rather, it expresses an assumption that the server makes over its environment on how it should be used. If the environment does not fulfill this assumption, the server will eventually degrade. How can this additional information be included in the system model? The manufacturer of the server ought to be able to guarantee a certain level of efficiency based on the amount of register information the server processes, therefore an assumption that the server will be purged before a certain number of *Register* events can be easily modeled by an LTS. Figure 3 models this assumption choosing (arbitrarily) five as the amount of *Register* events for which performance of the server cannot be guaranteed. Note that state 5 is a trap o error state that models the fact that the assumption has not been complied to. In Figure 4 we describe the complete description of the Server component as a composition of its behavior (boxes with solid lines) and assumptions (boxes with dashed lines).

Suppose a Control component was built implementing the company's business rules but not taking into account server assumptions (Figure 5). The final system model (Figure 6) including trucks and server and control components should provide the necesary information to detect a mismatch between how the server is used and how the server

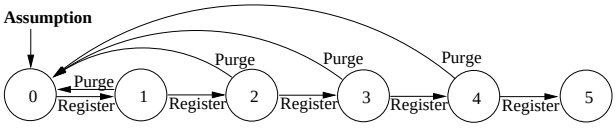

Figure 3. Assumptions of the Server component

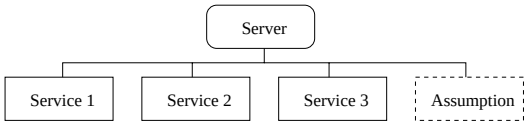

Figure 4. Complete server description including behavior and assumptions

was expected to be used. In this case, as server assumptions were explicitly included in the model, any deadlock detection method or reachability analysis (for example [3, 13]) can reveal the mismatch: the system can evolve into state 5 of the server assumption therefore the server assumptions do not hold meaning that system performance cannot be guaranteed.

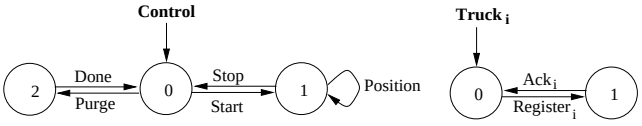

Figure 5. Behaviors of the control and truck component

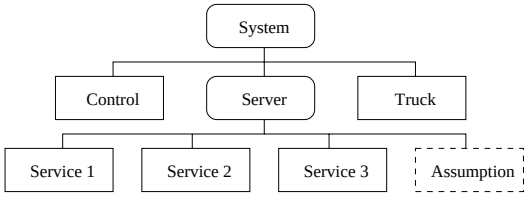

Figure 6. Truck tracking system

A solution to this problem (taking into account that the COTS server is not modifiable) is building a wrapper around the server that automates all maintenance operations hiding the original server from components while preserving its services. It has a very simple structure with one process for truck registering and another for performing maintenance operations when needed (Figures 7 and 8).

By applying simple changes to other components (Figure 9) a new system (Figure 10) may be constructed that guarantees assumptions made by the Server component.

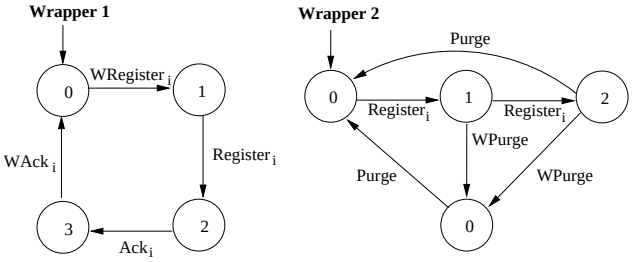

Figure 7. Behavior of the wrapper processes

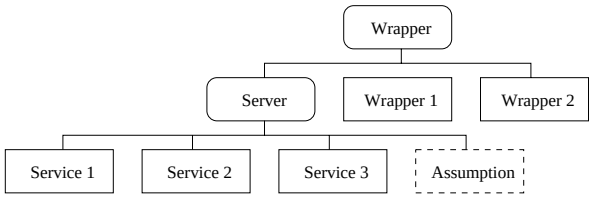

Figure 8. Wrapper component

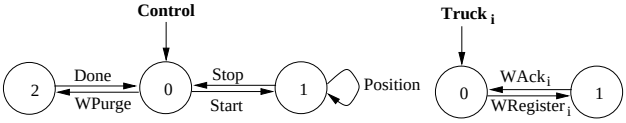

Figure 9. Behaviors of modified control and truck components

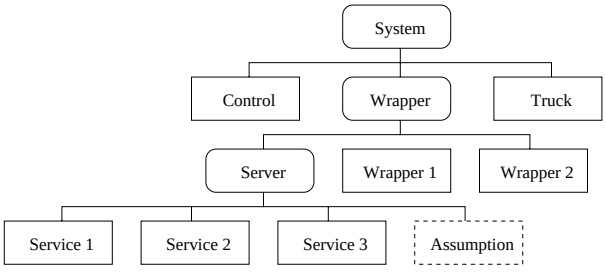

Figure 10. Truck tracking system with assumptions

The issue here is that a although the initial system configuration was correct in terms of component interaction, a mismatch was occurring between the intended use of a component and the actual use it was given by its environment. Failure in satisfying the component assumption means that the component cannot guarantee a certain level of performance, therefore the consequence of not using explicit as-

sumption would have been not detecting in advance a unsatisfactory system configuration. In the next example we show how assumptions can be used to guarantee correct functional use of a component based on a well-known example.

3.2. A gas station example

We present another example based on a case presented originally by [6] and further studied by [12]. A gas station is modeled by an operator, two customers, a pump and a customer request queue. Figures 12 and 11 show the LTS for the behaviors of all components. The operator accepts money from clients ($Prepay_i$) and according to the request queue activates ($Activate_i$) the pump. Once the operator receives the charge information ($Charge_i$) from the pump the change ($Change_i$) is given to the customer and the request queue is updated. The pump must be activated before receiving commands for starting and stopping to pump gas ($Start_i$, $Finish_i$) and finally informing the amount to be charged.

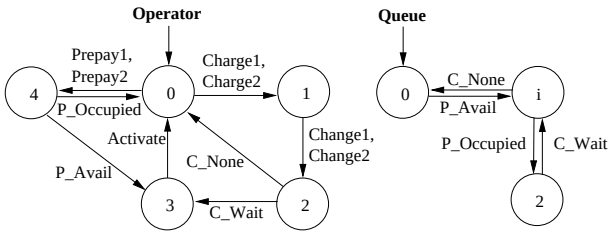

Figure 11. Behavior of the operator and the queue components

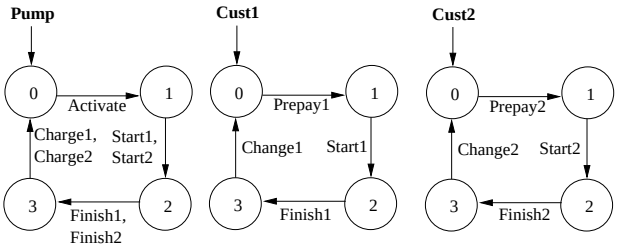

Figure 12. Behavior of the pump and customer components

The complete system behavior (Figure 13) has a problem [12]: The operator may receive charge information of Customer_1 from the pump and use for giving change to Customer_2. The origin of this malfunction is in the interaction between the operator and the pump. However, there is no interaction mismatch, components coordinate correctly and the system does not deadlock. The problem is that the operator is not working correctly in terms of the assumptions that the pump has. The pump can serve one customer at a time, before serving a customer it requires to be activated. Although the pump behavior does not restrict *Start*, *Finish* and *Charge* commands to be of the same customer after activation, it assumes that it will be. As these assumptions have not been explicitly stated in the system model of Figure 13 the fact that an important system property does not hold may not be detected.

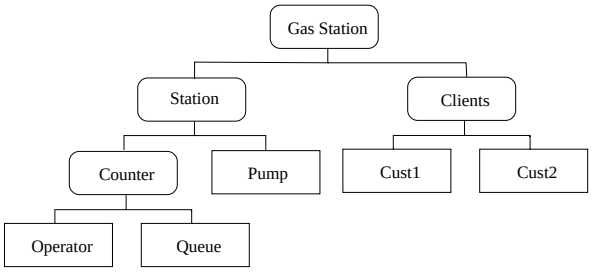

Figure 13. Gas station system

The assumptions the pump makes on how customers are to be managed can be modeled as the composition of two individual assumptions (Figure 14). The first one models the pumps assumption that customers are charged immediately after they have finished pumping gas, the second models the supposition that customers do not use a different customer number for starting and stopping the pump. If these assumptions are kept, problems such as *correct change* [12] would not arise. Expliciting these assumptions as in Figure 15 allows for mismatch detection using standard techniques ([3, 13]). Analysis reveals that the new system model can reach state 4 of the first pump assumption, thus the pump has not been included in an adequate environment and correct system behavior cannot be guaranteed.

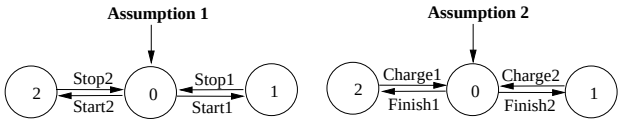

Figure 14. Assumptions of the pump component

4. Assumptions in architecture description languages

In previous sections, we have shown why component behavior is not the only cause for architectural mismatch and

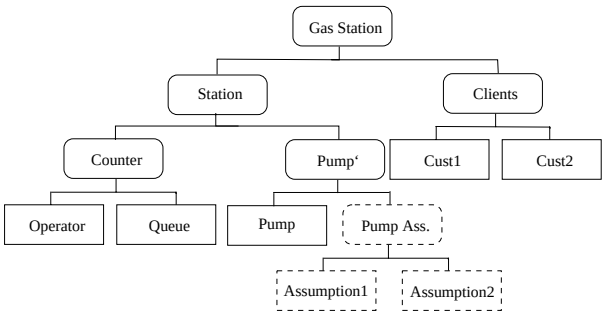

Figure 15. Complete gas station description

therefore can provide important information at the architectural design stage. We now discuss some issues on how component assumptions may be included in current architecture description languages (ADLs).

The first point is if component assumptions are already expressible in current ADLs. A first thought might be the following. Assumptions seem to predicate over interaction and ADLs already model component interaction, aren't assumptions the same thing as the behavioral descriptions found in ADLs? No, for several reasons:

- Component behavior is implemented in terms of ports, function calls, service points, and protocols in real programming languages. Oppositely, assumptions are not explicitly written into component code, they are component requirements, therefore they are validated and verified, not implemented.
- A component assumption predicates on its environment behavior, not on its own behavior. Component behavior predicates about interaction between the component and "neighboring" components while component assumptions may be made on components that do not interact directly with itself. The alphabet for describing behavior is different from the one for assumptions.
- Behavioral descriptions model the capabilities of a component. In order to use the component, the environment must interact through the modeled interaction point following the defined protocols. Assumptions can be, in some sense, not so strong depending on the property they guarantee. The architect could decide explicitly not to satisfy an assumption and this decision may not invalid the use of the component. In our example, if users do not maintain the server as assumed, the server might not stop working, but performance could drop dramatically. This could even be irrelevant to a user not concerned about performance issues.

In architecture description languages concepts such as connectors (Wright [20], Unicon [19], C2 [17], Aesop [5]), connections (Rapide [14], MetaH [21]) or bindings (Darwin [15]) are used to represent interaction between components: high level protocols, interaction points, ports, etc. These concepts are based on the notion of information exchange and interaction, thus architecture description languages presently cannot express assumptions. In particular cases, connectors can technically be used to express some assumptions that a component does. But this is neither the goal nor the idea of connectors. To illustrate this point we will comment why Wright cannot be used for expressing assumptions. We choose because it is a well known ADL that captures many of the concepts in the software architecture field, however our discussion could be based on some of the other ADLs. In our example server assumptions describe the expected environment behavior in terms of three different interaction points. If one were to model these assumptions in Wright, it would be necessary to use a port specification. However as our assumption refers to all three server services, all services would need to be provided through the same port. Although this restriction on the way services must be modeled is rather unintuitive, Wright also requires port specifications to be an abstraction of the component behavior. So it would also be necessary to include the components assumption in the components behavior description. Thus both behavior and assumptions would be mixed which is not what we want. We would end up with a complicated behavior description, in which different architectural aspects are mixed up. In particular, the Wright specification would be expressing that the server requires a certain interaction pattern in order to interact correctly, while the assumption intended to be a requirement needed to assure performance aspects.

How can assumptions be included in a SA description? The starting point is that as behavior and assumptions refer to different problems it is natural for the architect to provide them separately when modeling a component. Two main aspects need to be defined, how to model assumptions and how to combine behavior and assumption descriptions.

As there are many kinds of assumptions that a component modeling them can be a difficult task. Component assumptions can range from non-functional properties such as network latency to component version numbers. Even considering assumptions on interaction, which is on what ADLs focus on, there are many possibilities. A component might expect a certain usage pattern of its services as we have seen in section 3.1. These patterns could involve time constraints such as when a database server might have been designed to support up to a certain amount of register updates per minute. A component might even assume the probability of a service being used. The picture gets more complicated as assumptions are not necessarily limited to the components interactions with its environment, assump-

tions on *intra*-environment interaction can occur too: Take an agent coordinating the work of several components, it might make assumptions on the kind and moment of interactions that occur between components being coordinated.

Although the spectrum of assumptions is too big to manage in one formalism, ADLs can be extended to model some assumptions. ADLs model component interaction and system structure, therefore it is reasonable to extend them to express assumptions on the interaction capabilities of the environment. The following are some guidelines that should be used:

- ADLs model component behavior in an operational manner, thus a feasible and simple approach would be give assumptions in a similar manner.
- Probably the same kind of formalism could be used to describe behavior and assumptions. In our examples labeled transition systems were used, however other formalisms such as process algebra (as in Wright) could be used.
- Assumption specification must be independent from component behavioral description as they represent different ideas, component assumptions are not implemented, behavior is.
- Component interface should remain untouched by assumptions.
- Components are, at an architectural level, the basic construction unit, therefore their complete specification (behavior and assumptions) should be regarded as unit too in the overall system model. Components modeled together with their assumptions must be composable in the same way components are combined in ADLs presently.
- Component assumptions are not implemented thus assumptions *communicating* or *interacting* make no sense: Assumptions must not add component or system behaviors. In other words combining components behavior with its assumptions must, at most, restrict the overall component behavior.
- Mismatch detection can be done by making assumption specifications *listen* or *monitor* architecture interaction and keep track of the properties they express. In this way standard reachability techiniques and tools could be used.

5. Conclusions and future work

In this work we have proposed applying a broader notion of connection to software architectures. The connections between components also include the assumptions that the components make about each other. We have shown through some simple examples that explicitly specifying component assumptions is important, it can help detect software architecture mismatches that otherwise go unnoticed. We have shown that existing ADLs cannot, as they are now, express assumptions and finally instead of present yet another ADL we show how existing ones can be easily extended to include some kinds of assumptions.

The contribution of this work from the software engineering perspective is that using assumptions in an architectural description allows expressing requirements that are not in the scope of behavioral descriptions. Even for requirements that relate closely to behavioral description of a component, assumptions provide conceptual clarity, separating concerns and not interfering with components interface.

This work represents only a first step in adding assumptions to software architecture descriptions. An interesting aspect to work on is to understand what kinds of assumptions are relevant at a software architecture level description, for example assumptions on structure, on regularity, distribution, bursts of service calls, assumptions on data, etc. A second aspect is how to include these other kinds of assumptions in existing ADLs and lastly modify or create new ADLs that can manage assumptions. Finally, the relation between assumptions and non-functional properties, as shown in our example, should be further analyzed.

A. Labeled Transition Systems

We model process behavior by labeled transition systems (LTS). An LTS of a process contains the set of states that the process can reach and a set of labeled transitions between states that model all possible evolutions of the process state. Transition labels are the actions performed by a process that are relevant for a particular behavioral description. The internal actions of a process that determine state changes are represented by transitions labeled with the symbol τ. The set of non internal actions of a process P is called *alphabet* and noted $\alpha(P)$. Formally an LTS of a component is a quadruple (S, A, δ, s) where S is a set of states, $A = \alpha(P) \cup \{\tau\}$ is a set of labels, $\delta \subseteq S \times A \times S$ is a transition relation and $s \in S$ is the initial state.

An LTS of a process $P = (S, A, \delta, s)$ can evolve into an LTS of process $P' = (S, A, \delta, s')$ by an action $a \in \alpha(P)$ if and only if $(s, a, s') \in \delta$. We write this $(S, A, \delta, s) \xrightarrow{a} (S, A, \delta, s')$.

Processes can be composed determining new process behaviors. We will use a parallel composition operator ($\|$) like in CSP [7] where the composition of processes P and Q results in the synchronization of actions they have in common and interleaving of the others. Rules in Figure 16 give the operational semantics of the operator.

$$1a. \quad \frac{P \xrightarrow{a} P'}{P \| Q \xrightarrow{a} P' \| Q} \qquad a \notin \alpha(Q)$$

$$1b. \quad \frac{Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P \| Q'} \qquad a \notin \alpha(P)$$

$$2a. \quad \frac{P \xrightarrow{a} P', Q \xrightarrow{a} Q'}{P \| Q \xrightarrow{a} P' \| Q'} \qquad a \in \alpha(P) \cap \alpha(Q)$$

Figure 16. Rules for the parallel composition operator

As the parallel composition operator is commutative and associative we simplify notation by writing $(P \parallel Q) \parallel R$ as $P \parallel Q \parallel R$.

References

[1] R. Allen. *A Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

[2] R. Allen and G. Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering (ICSE'94)*, pages 71–80, Sorrento, Italy, May 1994.

[3] R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: a semantics based tool for the verification of concurrent systems. *ACM Transactions on Programming Languages and Systems*, 15(1):36–72, January 1993.

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

[5] D. Garlan, R. Allen, and J. Ockerbloom. Exploiting style in architectural design environments. In *Proceedings of SIGSOFT'94: Foundations of Software Engineering*, pages 175–188, New Orleans, December 1994.

[6] D. Helmbold and Luckham D. Debugging ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

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

[8] P. Inverardi and A. Wolf. Formal specification and analysis of software architectures using the chemical abstract machine model. *IEEE Transactions on Software Engineering*, 21(4):373–386, April 1995.

[9] P. Inverardi, D. Yankelevich, and A. Wolf. Checking assumptions in component dynamics at the architectural level. In *Second International Conference on Coordination Models and Languages (COORD '97), LNCS 1282*, pages 46–63, Berlin, 1997. Springer Verlag.

[10] G. Kiczales. Why are black boxes so hard to reuse? toward a new model of abstraction in the engineering of software. Invited Talk, OOPSLA '94, (http://www.uvc.com/kiczales/transcript.html), 1994.

[11] G. Kiczales, J. de Riviere, and D.G. Bobrow. *The Art of Metaobject Protocol*. MIT Press, Cambridge, MA, 1991.

[12] J. Kramer and J.C. Cheung. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *SIGSOFT95: 3rd International Symposium on the Foundations of Software Engineering*, pages 140–150, Washington D.C., October 1995.

[13] J. Kramer and S. C. Cheung. Checking safety properties using compositional reachability analysis. *ACM Transactions on Software Engineering and Methodology.*, 8(1):49–78, January 1999.

[14] D.C. Luckham, J.J. Kenney, L.M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[15] J. Magee, N. Dulay, S. Eisenbach, and J. Kramer. Specifying distributed software architectures. In *Proceedings of the Fifth European Software Engineering Conference (ESEC'95)*, Barcelona, September 1995.

[16] J. Magee, J. Kramer, and D. Giannakopoulou. Analysing the behaviour of distributed software architectures: A case study. In *Fifth IEEE Workshop on Future Trends of Distributed Computing Systems*, pages 240–247, October 1997.

[17] N. Medvidovic, R.N. Taylor, and E.J. Whitebead. Formal modeling of software architectures at multiple levels of abstraction. In *Proceedings of the California Software Symposium 1996*, pages 28–40, Los Angeles, CA, April 1996.

[18] D. Parnas. Information distribution aspects of design methodology. In *Proceedings of the 1971 IFIP Congress*, Amsterdam, November 1971. North Holland Publishing.

[19] M. Shaw, R. DeLine, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, pages 314–335, April 1995.

[20] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, New Jersey, 1996.

[21] S. Vestal. Metah programmer's manual, version 1.09. Technical report, Honeywell Technology Center, April 1996.