On the Analysis of Dynamic Properties in Component-Based Programming

Paola Inverardi[1] and Alexander L. Wolf[2]

[1] Dip. di Matematica, Universit'a dell' Aquila, I-67010, L'Aquila, Italy
[2] Dep. of Computer Science, University of Colorado, Boulder, Colorado USA

In recent years important changes have taken place in the way we produce software artifacts. The emerging market of commercial off-the-shelf (COTS) components and the increasing spread of component integration technologies such as CORBA, Java/RMI, and COM are determining a completely new way of building distributed systems. Although integration technologies and development techniques assume rather simple architectural contexts (usually distributed, with simple interaction capabilities), they face a critical problem that poses a challenging research issue: understanding if system components correctly integrate.
Component assembly can result in architectural mismatches when trying to integrate components with incompatible interaction behavior, leading to system deadlocks, livelocks, or failing to satisfy desired general functional and non-functional system properties. The approach we present in this paper is focused on preventing and detecting dynamic integration errors in a component-based development setting. We describe a method for deadlock detection that takes a novel approach based on component assumptions and provides a conservative checking algorithm with a state-space complexity significantly lower than comparable approaches. Although the focus in the present paper is on deadlock freedom of fully synchronized components, we believe the method can be generalized to support other composition mechanisms such as asynchronous communication and other properties such as general liveness and safety properties.

1 Introduction

In recent years important changes have taken place in the way we produce software artifacts. On one side, software production is becoming more and more involved with distributed applications running on heterogeneous networks. On the other, emerging technologies such as commercial off-the-shelf (COTS) products are becoming a market reality for rapid and cheaper system development [17]. Although these trends may seem independent, they actually have been bound together with the widespread use of component integration technologies such as CORBA, Java/RMI, and DCOM. Distributed applications are being designed as sets of autonomous, decoupled components, allowing rapid development based

on integration of COTS and simplifying architectural changes required to cope with the dynamics of the underlying environment.

Although integration technologies and development techniques assume rather simple architectural contexts, usually distributed, with simple interaction capabilities, they face a critical problem that poses a challenging research issue: understanding whether system components integrate correctly.

There is a growing interest on this topic, both in industrial and military contexts. For example, consider this quote from a recent US Defense Department briefing:

> *"A major theme of this year's demonstrations is the ability to build software systems by composing components, and do it reliably and predictably. We want to use the right components to do the job. We want to put them together so the system doesn't deadlock."*[1]

While for type integration and interface checking, type and subtyping theories play an important role in preventing and detecting some integration errors, interaction properties remain problematic. Component assembly can result in architectural mismatches when trying to integrate components with incompatible interaction behavior [1, 5], resulting in system deadlocks, livelocks, or failing to satisfy desired general functional and non-funtional system properties.

The work we present in this paper represents one step in our general goal of preventing and detecting dynamic integration errors in component-based development. Within this setting, our aim is twofold. First we want to analyze systems in a component-wise manner, that is, we wish to use information providable at a component level to verify system properties. This makes sense in a component-based world: components are bought as is, so if we understand what information component makers should provide, better systems (i.e., ones that have the desired global system properties) can be constructed. Second, we want to be able to verify dynamic properties. This means providing tractable methods that can manage real scale applications even at the cost of completeness. This is what at present type checking provides for a whole class of static correctness properties.

So far, existing techniques for detecting dynamic integration errors are based on behavioral analysis [3, 11] of the composed system model. The analysis is carried on at the system level, possibly in a compositional fashion [6], and has serious problems with state explosion. Our approach [9, 8] is based on enriching component semantics with additional information and performing analysis at a component level without building the system model. Additional information is provided by means of *assumptions*, which are the requirements a component puts on its environment in order to guarantee a certain property in a specific composition context.

The method we formulate starts off with a set of components to be integrated, a composition mechanism (e.g., full sychronization), and a property to be verified (e.g., deadlock freedom). We represent each component with an actual behavior graph (AC). An assumption graph (AS) for proving deadlock freedom is derived

[1] http://www.dyncorp-is.com/darpa/meetings/edcs99jun/

from each AC graph. Our checking algorithm processes all AC and AS graphs trying to verify if the AC graphs provide the requirements modeled by all the AS graphs. The algorithm works by finding pairs of AC and AS graphs that match through a suitable partial equivalence relation. According to the match found, arcs of the AS graph that have been provided for (covered arcs) are marked, and root nodes of both AC and AS graphs are updated. The algorithm repeats this process until all arcs of all AS graphs have been covered or no matching pair of graphs can be found. The former implies deadlock freedom of the system while the latter means that the algorithm cannot prove system deadlock freedom. Consequently, our algorithm is not complete (i.e., there are deadlock free systems that the algorithm fails to recognize), which is the price we must pay for tractability.

Summarizing, the contributions of our approach are a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component based setting that is vey efficient in terms of space complexity. While the space complexity of our approach is polynomial, existing approaches have exponential orders of magnitude. In this paper we give an informal description of the steps of our approach and we illustrate the method on a simple example. Complete presentations of the approach in the scope of two different specification contexts, namely CHAM and CCS, can be found elsewhere [8, 9].

2 Related Work

In order to obtain efficient verification mechanisms in terms of space complexity, there has been much effort to avoid the state explosion problem. There are two approaches: compositional verification and minimizations. The first class verifies properties of the individual components, and properties of the global system are deduced from these [12, 14, 18]. However, as pointed out by Grumberg and Long [14], when verifying properties of the components it may also be necessary to make assumptions about the environment, and the size of these assumptions is not fixed. Our approach shares the same motivation but it verifies properties of the component context, represented as fixed size AS graphs, in order to ensure a global system property.

The compositional minimization approach is based on constructing a minimal semantically equivalent representation of the global system. This is managed by successive refinements and use of constraints and interface specifications [6, 7]. However, these approaches still construct some kind of global system representation, and therefore are subject to state explosion in the worst case.

Binary Decision Diagrams [2] are used in many implementations for coding system states. Although it has been proved to be an efficient approach in many cases, it still suffers from space complexity problems.

From the perspective of property checking in large software systems, work in the area of module interconnection and software architecture languages can be mentioned, however the focus is not on efficient property verification of dynamic

properties nor is the specific setting of component-based programming taken into account [9].

There are other attempts at proving partial deadlock freedom statically. Kobayashi and Sumii [10, 16] propose a type system that ensures certain kinds of deadlock freedom through static checking. Their approach is based on including the order of channel use in the type information and requiring the designer to annotate communication channels as reliable or unreliable. As in our work, they use behavioral information to enhance the type system, however part of the additional information must be provided by users and is related to channels rather than components. In our approach, additional information is derived from the property to be proved and the communication context. Besides, the derived information extends component semantics, thus integrating well with the current direction that software development has taken, based on component integration technologies and commercial off-the-shelf products.

3 Property Checking Using Assumptions

We represent component behavior (and component assumptions later on) using directed, rooted graphs. We define the notion of *actual behavior* (AC) graph for modeling component behavior. The term "actual" emphasizes the difference between component behavior and the intended, or assumed, behavior of the environment. AC graphs model components in an intuitive way. Each node represents a state of the component and the root node represents its initial state. Each arc represents the possible transition into a new state where the transition label is the action performed by the component.

In this section we present the various steps upon which our approach is based, i.e., how component assumptions can be derived and used for proving deadlock freedom in a system composed of a finite number of components that communicate synchronously. Following a common hypothesis in automated checking of properties of complex systems [11], behavior of all components can be finitely represented.

3.1 Deriving Assumptions for Deadlock Freedom

We wish to derive from a component behavior the requirements on its environment that guarantee deadlock freedom. A system is in deadlock when it cannot perform any computation, thus in our setting, deadlock means that all components are blocked waiting for an action from the environment that is not possible. Our approach is to verify that no components under any circumstance will block. This conservative approach suffices to prove deadlock freedom exclusively from component assumptions. The payback is efficiency, while the drawback is incompleteness.

Let us consider a context in which components are combined together, composing them in parallel and forcing them to synchronize whenever possible, where

synchronization is obtained when they offer at the same time complementary actions [8]. In this context, a component will not block if its environment can always provide the actions it requires for changing state. Thus, we can define the notion of component assumption in the context of parallel composition and deadlock freedom as a sort of complementary graph of the AC graph, that is, a graph that is structurally identical to the AC graph but whose labels are complementary with respect to the corresponding labels in the AC graph. We call this graph the *assumption graph* (AS).

3.2 Checking Assumptions

Once component assumptions have been derived, we wish to verify if these assumptions are satisfied by the environment, which, intuitively, is the rest of the components in the given context. This satisfaction relation reduces to proving if the component environment is equivalent to the component assumption by means of a suitable notion of equivalence. The idea behind the definition of equivalence we use is that the graphs can always imitate each other. If a graph performs an action l, the other graph can also perform l and, no matter what internal choices it may make, it will be able to continue imitating the other graph. However, our notion of equivalence is more restrictive than the notion of weak bisimilarity [13], since we need to assure that a given behavior *must* be provided by all the branches that provide the matched portion.

We verify the equivalences between AS graphs and environments without constructing the whole environment behavior. The main idea is to allow a portion of a component behavior to provide a portion of another component assumption. For this we need to provide a notion of *partial equivalence* that preserves equivalence in a conservative way. Once a partial equivalence has been established, the assumption graph has been satisfied to some extent and therefore some marking mechanism is necessary in order to record it.

Partial Equivalence A partial equivalence between an AC and an AS graph allows the equivalence relation to be defined up to a certain point in the graphs. The AC and AS graphs are not required to be completely equivalent; their root nodes must be equivalent, the nodes reachable from root nodes too, and so on until a set of nodes called *stopping nodes* is reached. Stopping nodes represent the points where the actual behavior will stop providing the assumption's requirements, hence there should be another AC graph capable of doing so from then on.

Checking Algorithm and an Example Application The checking algorithm is very simple. It iteratively finds partial equivalencies between AC and AS graphs, marks all the fulfilled assumptions, and changes the roots of both graphs. Iteration stops when all assumptions are completely marked. An important point is that partial equivalences guarantee that the matched portions of

assumptions cannot be matched in any other way, therefore the order in which partial matches are applied does not affect the correctness of the algorithm.

We now apply the algorithm to the Compressing Proxy example [4, 9]. To improve the performance of UNIX-based World Wide Web browsers over slow networks, one could create an HTTP (Hyper Text Transfer Protocol) server that compresses and uncompresses data that it sends across the network. This is the purpose of the Compressing Proxy, which weds the **gzip** compression/decompression program to the standard HTTP server available from CERN.

The main difficulty that arises in the Compressing Proxy system is the correct integration of existing components. The CERN HTTP server consists of *filters* strung together in series executing in one single process, while the **gzip** program runs in a separate UNIX process. Therefore, an adaptor must be created to coordinate these components correctly (see Figure 1).

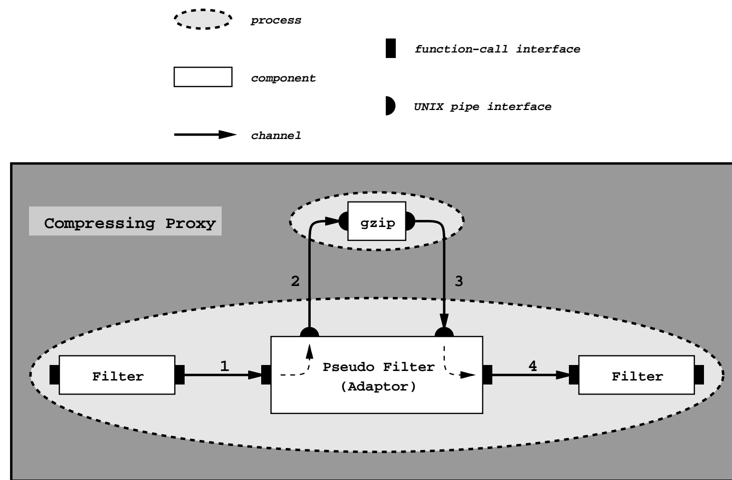

Fig. 1. The Compressing Proxy.

However, the correct construction of the adaptor requires a deep understanding of the other components. Suppose the adaptor simply passes data on to **gzip** whenever it receives data from the upstream filter. Once the stream is closed by the upstream filter (i.e., there are no more data to be compressed), the adaptor reads the compressed data from **gzip** and pushes the data toward the downstream filter.

At a component level, this behavior makes sense. But at a global system level we can experience deadlock. In particular, **gzip** uses a one-pass compression algorithm and may attempt to write a portion of the compressed data (perhaps because an internal buffer is full) before the adaptor is ready, thus blocking.

With **gzip** blocked, the adaptor also becomes blocked when it attempts to pass on more of the data to **gzip**, leaving the system in deadlock.

A way to avoid deadlock in this situation is to have the adaptor handle the data incrementally and use non-blocking reads and writes. This would allow the adaptor to read some data from **gzip** when its attempt to write data to **gzip** is blocked.

We represent partial equivalences with dotted lines for related nodes and crosses for stopping nodes. In Figure 2, the upstream filter matches successfully with the adaptor. Once the successful match has been made, both graphs are modified. The new state of the adaptor can be seen in Figure 3.

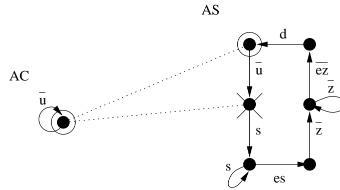

Fig. 2. Successful Match of Upstream Filter AC Graph against Adaptor AS Graph.

Figure 3 shows how a partial match can be established between the **gzip** AC graph and the adaptor AS graph. However, it is possible to see that there is no way of extending the relation in order to cover the edge labeled $\overline{es}$. Hence, the algorithm, after all possible attempts, terminates, indicating that the proposed configuration is presumably not deadlock free.

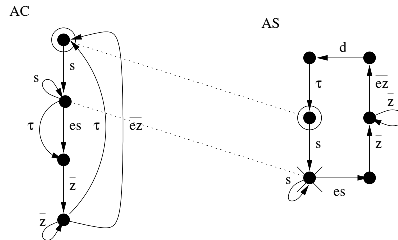

Fig. 3. Unsuccessful Match of **gzip** AC Graph against Adaptor AS Graph.

Notice that the mismatch occurs precisely where the deadlock in the system appears: **gzip** may attempt to output the compressed file ($\overline{z}$) while the adaptor is expecting to be synchronizing with a component, inputting an *end of source* (es) before the compressed file is output.

The adaptor must be modified to prevent system deadlock. In Figure 4, the partial equivalence that covers the $\overline{es}$ edge allows the modified adaptor's AS

graph to be updated, and Figure 5 finishes covering the AS graph completely. The algorithm goes on matching AC and AS graphs until all arcs of all AS graphs are covered. Thus, the checking algorithm finally returns true, meaning that the proposed system is deadlock free.

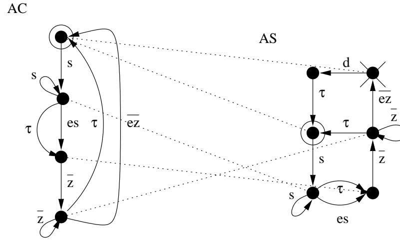

Fig. 4. Successful Match of **gzip** AC Graph against a Modified Adaptor AS Graph.

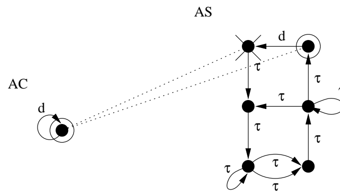

Fig. 5. Successful Match of Downstream Filter AC Graph against a Modified Adaptor AS Graph.

4 Method Assessment

Up to now we have informally introduced a method for checking deadlock freedom that trades off completeness for efficiency. We now briefly comment on the completeness and complexity of our approach. A detailed discussion of all the hypothesis and implications on the complexity, completeness, and correctness of our approach can be found elsewhere [8].

4.1 Complexity

The algorithm we sketched above offers a partial solution to the state explosion problem. In our approach, deadlock freedom is proven without building the entire finite-state model of the system. We only construct finite representations of each component individually: an actual behavior graph and an assumed behavior graph of its context.

In standard aproaches, using reachability analysis, the complete state space of the system is built. If we consider a concurrent system composed of N components of comparable size, whose finite state representation is of size $O(K)$, then the composed system state space is $O(K^N)$. Although there are many techniques for reducing the state space, such as automata minimization and "on the fly" algorithms, the worst case still requires the whole state space to be analyzed, leading to a time complexity of $O(K^N)$.

In our approach only two copies of each component are built, AC and AS graphs. Thus, following the same considerations as before, the state space complexity is radically improved to $O(KN)$. On the other hand, in terms of time complexity, the worst case of our algorithm is $O(N^3K^4log(K))$, which is comparable to the worst case of standard reachability. The time complexity results from the following: Establishing a partial equivalence relation between two graphs can be considered a variation of the standard bisimulation checking. Thus, the upper bound on its complexity would be $O(K^2log(K))$ [15]. However, the partial equivalence must be established for a pair of graphs. Thus, all possible pairs must be checked ($Comb(N,2)$), leading us to $O(N^2(K^2log(K)))$. Finally, considering the worst case in which each partial match only covers a single arc of the NK^2 possibilities. We get $O(K^2N^3(K^2log(K)))$, which reduces to $O(N^3K^4log(K))$.

4.2 Completeness

The approach presented in this paper may be considered incomplete in two different ways: firstly, some restrictions on the systems for which the method can be used are necessary, and secondly, because the checking algorithm may not be able to conclude deadlock freedom for some deadlock-free systems.

The first restriction on the system requires components to be able to perform each computation an infinite number of times, but does not affect the completeness of out approach. The goal of this restriction is only for the sake of simplicity of the formal presentation. The second restriction is more serious. We do not accept that more than two components have shared channels. If a communication channel can be used by more than two components, there is a potential nondeterminism in the overall system behavior. A component may have the possibility of synchronizing with one of several components leading to a nondeterministic choice. In terms of our approach, this means that one cannot commit to which AC graph will provide the AS graph requirements. As the matching process guarantees that the matched arcs of the AS graph will always be provided by the AC graph, no matching can be done. The nondeterminism introduced by shared channels is similar to the nondeterminism that makes our algorithm incomplete.

Having discussed the restriction imposed on components, the incompleteness of the checking algorithm remains. Our approach is intrinsically incomplete. First of all, we attempt to prove a global property such as deadlock in terms of local properties of each component. Second, we verify equivalences between the context and component assumptions in sucessive partial steps so as to not construct

a complete model of the component context. As a consequence, the characteristics of our setting lead to the following situation: Given a deadlock-free system, the algorithm may not be able to conclude that it is deadlock free. What happens is that the algorithm reaches a state in which it cannot do further matches between AC and AS graphs. However, the main reason for the incompleteness of our approach is nondeterminism. When there is a nondeterministic choice in a component's behavior, when a component can interact with one of two different components, there cannot be a unique mathing that guarantees how the system will evolve. In these situations the algorithm stops without obtaining AS graphs completely matched, and therefore not giving a conclusive answer. Incompleteness is the price that must be paid to make analysis tractable. Our method may apply only to a subset of problems, but it lowers the complexity of the solution from exponential order to a polynomial one.

5 Conclusions and Future Work

In this paper we have informally illustrated a preliminary space-efficient approach to proving dynamic properties of component-based systems. The approach assumes a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting. This method is based on deriving assumptions (component requirements on its environment in order to guarantee a certain property in a specific composition context) and checking that all assumptions are guaranteed through a partial matching mechanism. The method is considerably more efficient than methods based on system model behavior analysis, since its space complexity is polynomial, while existing approaches have exponential orders of magnitude. It is not complete, but it allows the treatment of systems whose synchronization patterns are not trivial.

Ongoing and future work is proceeding in two directions. First, to validate the proposed framework through experimental results, we are currently working on an implementation of the algorithm, and considering other coordination contexts, such as non-fully synchronized or asynchronous ones. Second, to extend the approach to deal with other properties, such as general liveness and safety properties, we are thinking of general safety properties expressed with property automata, such as in the Gas Pump example [11] that may be decomposed into component assumptions or specific component assumptions such as particular access protocols for shared resources.

We believe that assumptions are a good way to extend component semantics in order to verify properties more efficiently. The approach presented in this paper is an example of how this can be achieved.

Acknowledgements

We would like to thank Sebastián Uchitel and Daniel Yankelevich for discussions and common work on the subject of the paper.

References

1. B. Boehm and C. Abts. COTS Integration: Plug and Pray? *IEEE Computer*, 32(1), January 1999.
2. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L.J. Hwang. Symbolic model checking : 10^{20} and beyond. *Information and Computation*, 98:142–170, June 1992.
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. D. Compare, P. Inverardi, and A.L. Wolf. Uncovering Architectural Mismatch in Dynamic Behavior. *Science of Computer Programming*, 2(33):101–131, February 1999.
5. D. Garlan, R. Allen, and J. Ockerbloom. Architectural Mismatch: Why Reuse is so Hard. *IEEE Software*, 12(6), November 1995.
6. D. Giannakopoulou, J. Kramer, and S.C. Cheung. Analysing the Behaviour of Distributed Systems using Tracta. *Automated Software Engineering, special issue on Automated Analysis of Software*, 6(1):7–35, January 1999.
7. S. Graf, B. Steffen, and G. Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5), 1998.
8. P. Inverardi and S. Uchitel. Proving Deadlock Freedom in Component-Based Programming. Technical report, Universita' dell'Aquila, Italy, November 1999.
9. P. Inverardi, A.L. Wolf, and D. Yankelevich. Static Checking of System Behaviors Using Derived Component Assumptions. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear.
10. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, March 1998.
11. 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.
12. K. Laster and O. Grumberg. Modular model checking of software. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'98)*, Lisbon, March 1998.
13. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
14. O.Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
15. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
16. E. Sumii and N. Kobayashi. A generalized deadlock-free process calculus. In *3rd International Workshop on Hish-Level Concurrent Languages*, volume 16 of *Electronic Notes in Theoretical Computer Science.* Elsevier, September 1998.
17. Clemens Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.
18. G. Winskel. Compositional checking of validity on finite state processes. In *Workshop on Theories of Communication, CONCUR*, volume 458 of *LNCS*, 1990.