# Proving Deadlock Freedom in Component-Based Programming [*]

Paola Inverardi[1] and Sebastian Uchitel[2]

[1] Dip. di Matematica, Universit'a dell' Aquila, I-67010, L'Aquila, Italy. email: `inverard@univaq.it`

[2] Dep. of Computing, Imperial College, Queen's Gate 180, London, SW7 2BZ, UK. email: `su2@doc.ic.ac.uk`

**Abstract.** Emerging technologies such as commercial off-the-shelf products (COTS) and component integration frameworks such as CORBA and COM are changing the way software is produced. 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, they face a critical problem: Component integration.

So far existing techniques for detecting dynamic integration errors are based on behavioural analysis of the composed system and have serious space complexity problems. In this work we propose a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting. Our goal is to prevent and detect these errors in component based programming settings in a component-wise fashion. We aim for effective methods that can scale to real size applications even at the price of incompleteness as opposed to many existing methods that although theoretically complete might fail in practice.

## 1 Introduction

In recent years important changes have taken place in the way we produce software artefacts. On one side, software production is becoming 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 cheap system development [14]. Although these trends may seem independent, they actually have been bound together with the wide spreading of component integration technologies such as CORBA and COM. Distributed applications are being designed as sets of autonomous, decoupled components, allowing rapid development based on integration of COTS

[*] P. Inverardi was artially supported by the Italian MURST national Project SALADIN. S. Uchitel was partially supported by ARTE Project, PIC 11-00000-01856, ANPCyT and TW72, UBACyT.

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

Integration technologies and development techniques assume rather simple architectural contexts, usually distributed, with simple interaction capabilities. Nevertheless they face critical problems that pose a challenging research issues. For example, consider this quote from a recent US Defence 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 sub-typing theories play an important role in preventing and detecting some integration errors, interaction properties remain problematic. Component assembling can result in architectural mismatches when trying to integrate components with incompatible interaction behaviour (e.g. [5]), resulting in system deadlocks, livelocks or failing to satisfy desired general functional and non-functional system properties. So far existing techniques for detecting dynamic integration errors are based on behavioural analysis (e.g. [6, 4]) of the composed system model. The analysis is carried on at system level, possibly in a compositional fashion [6] and has serious problems with state explosion. Our goal is to prevent and detect these errors in component based programming settings in a component-wise fashion. We aim for effective methods that can scale to real size applications even at the price of incompleteness as opposed to many existing methods that although theoretically complete might fail in practice.

Our approach exploits the standardization and simplicity of the interaction mechanisms present in the component-based frameworks. We overcome the state explosion problem in deadlock verification for a significant number of cases. Our approach is based on enriching component semantics with additional information and performing analysis at a component level without building the system model. We start off with a set of components to be integrated, a composition mechanism, in this case full synchronization, and a property to be verified, namely deadlock freedom. We represent each component with an ACtual behaviour (AC) graph. An ASsumption (AS) graph for proving deadlock freedom is derived from each AC graph. Our checking algorithm processes all AC and AS graphs trying to verify if the AC graphs provide the requirements modelled 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,

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

our algorithm is not complete (there are deadlock free systems that the algorithm fails to recognize), which is the price we must pay for tractability.

Summarizing, the contributions of this work are a broader notion of component semantics based on assumptions and a method for proving deadlock freedom in a component-based setting that is very efficient in terms of space-complexity. While the space complexity of our approach is polynomial, existing approaches have exponential orders of magnitude.

In the next section we discuss related work. In Section 3 we informally introduce the characteristics of a simple component/configuration language based on CCS and recall the definition of Labelled Transition Systems which are our basic model. In Section 4 we illustrate a simple case study that is used in Section 5 to present our approach. We discuss the methods completeness and complexity and conclude with final comments and future work.

## 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 minimization. The first class verifies properties of individual components and properties of the global system are deduced from these (e.g. [12]). However as stated in [12] when verifying properties of 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 using fixed size AS graphs.

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, therefore are subject to state explosion in worst cases. Neither efficient data representations such as Binary Decision Diagrams [3] nor most recent results like [1] have solved the space complexity problem.

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. For an extensive treatment of this aspect refer to [9].

There have been other attempts at proving deadlock freedom statically. Interesting results in this direction can be found in [10] where a type system is proposed that ensures (certain kinds of) deadlock freedom through static checking. The 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 behavioural 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. In [2] component behaviour is decomposed into interface descriptions, which then can be used to prove deadlock freedom. The method is more incomplete than ours as it requires components not to exhibit non-deterministic behaviour that can be resolved by the influence of their environment. In other words, a non-deterministic choice involving a component input is not allowed. Our approach allows this kind of non-determinism, furthermore, the example used in this paper exhibits many of these non-determinisms.

The method presented in this paper originates from the work in [9]. However it differs in a number of ways:

- The present method is more complete. This is mainly because of the partial equivalence relation used in this approach: We do not require the whole behaviour of a component to be used when providing a portion of another component's assumption. Relaxing this requirement allows more systems to be checked. Detailed examples can be found in [8].
- The component/configuration language is well founded and simpler.
- There is a clear distinction between assumption generation and assumption checking.
- There is a clear distinction between notions of equivalence and partial matching, therefore it is possible to adapt the definitions for other notions of equivalence, allowing for example to switch from synchronous to asynchronous communication.

## 3 A Basic Component-Configuration Language

Our model for component-based systems describes components in terms of their input and output actions using labelled transition systems (LTS) (Definition 1). Input and outputs are considered to be blocking actions, thus we shall work with the (synchronous) parallel composition of LTS. System description using LTS and parallel composition is widely used in research (e.g. [6, 4]) and we have chosen CCS [11] as our specification language mainly for its simplicity and firm foundations. Thus, in this work component behaviour shall be described as CCS processes and system configuration shall be specified using the parallel composition and restriction operators. As LTSs of all examples used in this paper are shown, knowledge of CCS is not critical to follow the main ideas of this paper.

**Definition 1 (Labelled Transition Systems).** *A component $C$ is modelled by a labelled transition system $< S, L, \rightarrow, s >$, where $S$ is a set of states; $s$ is the initial component state; $L$ is a set of labels representing the channels through which the component can communicate; $\rightarrow \subseteq (S \times Act \times S)$ is a transition relation that describes the behaviour of the component.*

## 4 The Compressing Proxy Problem

In this section we briefly present the Compressing Proxy example. For a more detailed explanation refer to [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).

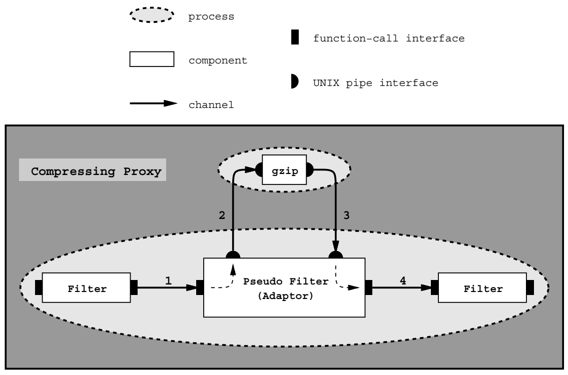

**Fig. 1.** The Compressing Proxy.

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 behaviour 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 model all four system component behaviours, gzip, Adaptor, Upstream and Downstream CERN filters as the CCS processes in Table 4. The Upstream CERN filter is very simple, it can continuously perform output actions through its *upstream* interface point ($u$). Similarly, the Downstream Filter can perform input actions through its *downstream* port ($d$). The gzip interface consists of a port for inputting the source file ($s$), one for outputting the compressed file ($z$), and two other ports to model *end of source file* ($es$) and *end of compressed file* ($ez$). Finally the adaptor interacts with all the other components, and therefore its interface is the union of all the other component interfaces. The complete system configuration is given by (UF | GZ | AD | DN) $\backslash\{u, s, es, z, ez, d\}$ and the following CCS processes:

| **Upstream Filter (UF)** | **Downstream Filter (DF)** |
|---|---|
| $\mathrm{UF} \overset{def}{=} \overline{u}.\mathrm{UF}$ | $\mathrm{DF} \overset{def}{=} d.\mathrm{DF}$ |
| **GZip (GZ)** | **Adaptor (AD)** |
| $\mathrm{GZ} \overset{def}{=} s.\mathrm{In}$ | $\mathrm{AD} \overset{def}{=} u.\overline{s}.\mathrm{ToGZ}$ |
| $\mathrm{In} \overset{def}{=} s.\mathrm{In} + es.\overline{z}.\mathrm{Out} + \tau.\overline{z}.\mathrm{Out}$ | $\mathrm{ToGZ} \overset{def}{=} \overline{s}.\mathrm{ToGZ} + \overline{es}.z.\mathrm{FromGZ}$ |
| $\mathrm{Out} \overset{def}{=} \overline{z}.\mathrm{Out} + \overline{ez}.\mathrm{GZ} + \tau.\mathrm{GZ}$ | $\mathrm{FromGZ} \overset{def}{=} z.\mathrm{FromGZ} + ez.\overline{d}.\mathrm{AD}$ |

## 5 Property Checking Using Assumptions

We represent component behaviour (and component assumptions later on) with directed, rooted graphs that simply extend labelled transition systems to allow multiple root nodes:

**Definition 2 (Graphs).** *A (directed rooted) graph $G$ is a tuple of the form $(N_G, L_G, A_G, R_G)$ where $N_G$ is a set of nodes, $L_G$ is a set of labels with $\tau \in L_G$, $A_G \subseteq N_G \times L_G \times N_G$ is a set of arcs and $R_G \subseteq N_G$ is a nonempty set of root nodes.*

- *We shall write $g \xrightarrow{l} h$, if there is an arc $(g, l, h) \in A_G$. We shall also write $g \rightarrow h$, meaning that $g \xrightarrow{l} h$ for some $l \in L_G$.*
- *If $t = l_1 \cdots l_n \in {L_G}^*$, then we write $g \overset{t}{\longrightarrow}{}^* h$, if $g \xrightarrow{l_1} \cdots \xrightarrow{l_n} h$. We shall also write $g \longrightarrow^* h$, meaning that $g \overset{t}{\longrightarrow}{}^* h$ for some $t \in {L_G}^*$.*
- *We shall write $g \overset{l}{\Rightarrow} h$, if $g \overset{t}{\longrightarrow}{}^* h$ for some $t \in \tau^*.l.\tau^*$.*

We define the notion of Actual Behaviour (AC) Graph for modelling component behaviour. The term actual emphasizes the difference between component behaviour and the intended, or assumed, behaviour 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.

**Definition 3 (AC Graphs).** *Let $< S, L, \rightarrow, s >$ be a labelled transition system for component $C$. We call a graph of the form $(S, L, \rightarrow, \{s\})$ the ACtual behaviour graph (AC graph) of a component $C$. We shall usually denote nodes in $AC$ with $\nu$.*

In the rest of the section we will show 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 [6], behaviour of all components can be finitely represented. In addition, in order to simplify presentation, we add two constraints:

- Components can perform each computation infinitely often. In other words, all nodes of a components AC graph are reachable from any other node. This condition simplifies the presentation of the partial equivalence relation. It can easily be dropped by introducing a proper treatment of *ending* nodes in the definitions below.
- There are no "shared" actions. This means that every communication is only used by two components. Again, this condition simplifies the presentation of the checking algorithm and can be easily dropped at the expense of more checks.

It is worth noticing that dropping the first condition has no implication on the complexity results of this presentation, while for the second only time complexity changes. In Section 6 we discuss more deeply the implications of both constraints.

## 5.1 Deriving Assumptions for Deadlock Freedom

We wish to derive from the behaviour of a component 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, as we shall show, is efficiency, while the drawback is incompleteness.

As components are combined together composing them in parallel and restricting all actions, 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 in the following way:

**Definition 4 (AS Graphs).** *Let $(N, L, A, R)$ be an AC graph of a component $C$, then the corresponding ASsumption (AS) graph is $(N, L, A', R)$ where $A' = \{(\nu, \overline{a}, \nu') \mid (\nu, a, \nu') \in A\}$. We shall usually denote nodes in AS graphs with $\mu$.*

The AC and AS graphs for the components of the Compressing Proxy appear in Figure 2.

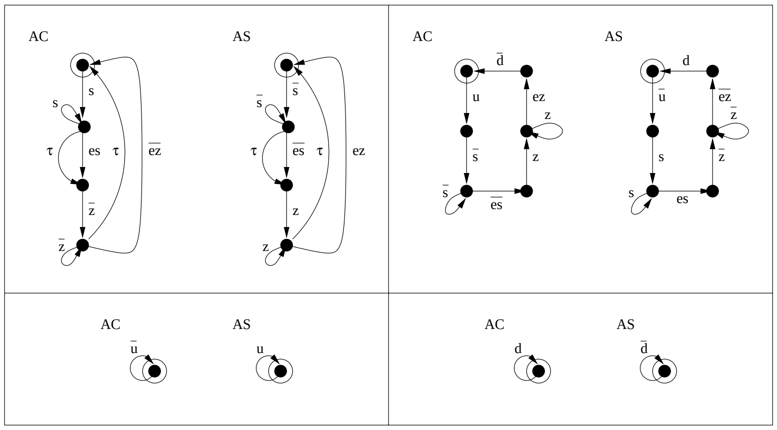

**Fig. 2.** Graphs for components GZ, AD, UF, SF (starting top-left)

## 5.2 Checking Assumptions

Once component assumptions have been derived, we wish to verify if the environment satisfies these assumptions. The environment corresponds to 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 with the following notion of equivalence:

**Definition 5 (Equivalence).** *Let $G$ and $H$ be graphs with the same alphabet, i.e. $L_G = L_H$. We define $\approx$ to be the union of all relations $\rho$ where $R_G \times R_H \subseteq \rho$ and if $(g, h) \in \rho$ the following two conditions hold:*

- $g \xrightarrow{l}_G g' \Rightarrow (\exists\, h' : h \overset{l}{\Rightarrow}_H h' \wedge (g', h') \in \rho) \wedge (\forall\, h' : h \overset{l}{\Rightarrow}_H h' \Rightarrow (g', h') \in \rho)$, *and*
- $h \xrightarrow{l}_H h' \Rightarrow (\exists\, g' : g \overset{l}{\Rightarrow}_G g' \wedge (g', h') \in \rho) \wedge (\forall\, g' : g \overset{l}{\Rightarrow}_G g' \Rightarrow (g', h') \in \rho)$.

The idea behind the definition of equivalence 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. Note that our notion of equivalence is more restrictive than the notion of weak bisimilarity [11] since we need to assure that a given behaviour *must* be provided by all the branches that provide the matched portion. This is what, in the above clauses, the for-all conditions express.

We verify the equivalences between AS graphs and environments without constructing the whole environment behaviour. The main idea is to allow a portion of component behaviour 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 (Section 5.2). 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. The checking algorithm of Section 5.2 iteratively finds partial equivalences and marks the assumptions accordingly until all assumptions have been satisfied.

**Partial Equivalence** A partial equivalence between an AC and 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 set of nodes called stopping nodes is reached. Stopping nodes represent the points where the actual behaviour will stop providing the assumption's requirements, hence there should be another AC graph capable of doing so from then on. We now formally introduce these notions. The notion of stopping nodes is needed to guarantee that the graph portions included in the partial equivalence correspond to behaviours starting from root nodes onwards. Throughout the following definitions, given a relation $\rho$ and an element $a$, we shall write $a \in \rho$ if there is an element $b$ such that $(a, b) \in \rho$ or $(b, a) \in \rho$.

**Definition 6 (Stopping Nodes).** *Let $G$ and $H$ be graphs and $\rho$ a relation in $N_G \times N_H$. We say that the set $S_{G,\rho} = \{g \mid g \in N_G \wedge g \in \rho \wedge g \notin R_G \wedge (g \xrightarrow{l} g' \Rightarrow (g' \notin \rho \vee g' \in R_G))\}$ is the set of stopping nodes of $\rho$ in $G$ . We omit the symmetric definition for $S_{H,\rho}$.*

Informally a stopping node is a node in the relation $\rho$ that is not a root node and for which no other nodes in $\rho$ are reachable.

**Definition 7 (Partial Equivalence).** *Let $G_{ac}$ be an actual behaviour graph with nodes $\nu_i$, $G_{as}$ be an assumption graph with nodes $\mu_j$. We define $\simeq$ to be the union of all relations $\rho$ such that $R_{G_{ac}} \times R_{G_{as}} \subseteq \rho$ and all the following hold:*

1. *if $\nu \in \rho$ then $\nu \in R_{G_{ac}}$ or there is a node $\nu' \in \rho$ such that $\nu' \rightarrow \nu$.*
2. *if $(\nu, \mu) \in \rho$, $\nu \notin S_{G_{ac},\rho}$ and $\mu \notin S_{G_{as},\rho}$ then*
   *(a) $\nu \xrightarrow{l} \nu' \Rightarrow (\exists\, \mu' : \mu \overset{l}{\Rightarrow} \mu' \wedge (\nu', \mu') \in \rho) \wedge (\forall\, \mu' : \mu' \overset{l}{\Rightarrow} \mu \Rightarrow (\nu', \mu') \in \rho)$, and*
   *(b) $\mu \xrightarrow{l} \mu' \Rightarrow (\exists\, \nu' : \nu \overset{l}{\Rightarrow} \nu' \wedge (\nu', \mu') \in \rho) \wedge (\forall\, \nu' : \nu' \overset{l}{\Rightarrow} \nu \Rightarrow (\nu', \mu') \in \rho)$.*
3. *if $(\nu, \mu) \in \rho$, $\nu \in S_{G_{ac},\rho}$ or $\mu \in S_{G_{as},\rho}$ then*
   *(a) if $\nu \in R_{G_{ac}}$ then $(\nu' \xrightarrow{l} \nu \Rightarrow \exists\, \mu' : \mu' \overset{l}{\Rightarrow} \mu)$, and*
   *(b) if $\mu \in R_{G_{as}}$ then $(\mu' \xrightarrow{l} \mu \Rightarrow \exists\, \nu' : \nu' \overset{l}{\Rightarrow} \nu)$.*

4. *if $(\nu, \mu) \in \rho$ and $\nu \in S_{G_{ac},\rho}$ then $\mu \in S_{G_{as},\rho}$ or $\mu \in R_{G_{as}}$.*
5. *if $(\nu, \mu) \in \rho$ and $\mu \in S_{G_{as},\rho}$ then $\nu \in S_{G_{ac},\rho}$ or $\nu \in R_{G_{ac}}$.*

*We say that $G_{ac}$ and $G_{as}$ are partially equivalent if $G_{ac} \simeq G_{as}$.*

The definition obviously resembles the definition of equivalence of Def. 5. We succinctly explain the additions: A partial equivalence relation does not necessarily cover all nodes of each graph, thus Rule 1 is needed to enforce that the nodes that are included in the relation are connected. By Rule 2, all nodes that are related and do not belong to the boundary where the actual behaviour stops providing the assumption requirement, are required to be equivalent (in the same sense as in Definition 5). Rules 3, 4 and 5 involve stopping nodes, the main idea is that equivalence is not required from these nodes onwards. However, stopping nodes must be related to stopping nodes, or if there is a loop (i.e. a path that returns to a root node) in one graph, a stopping node might be related to a root node (Rules 4 and 5). Rule 3 deals with spurious pairs of stopping and root nodes, by requiring the stopping node to represent the end of looping paths in the other graph. In short, all rules but 3 are intended for the *partial* part of the definition while Rule 3 is intended for the *equivalence* part.

**Checking Algorithm** The checking algorithm is very simple, it iteratively finds partial equivalences 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.

The checking algorithm that is presented below intends to clarify how the checking of component assumptions is done and to provide a basis for correctness, completeness and complexity. By no means is the algorithm, optimum. Many heuristics could be built into it in order to increase time and space efficiency, however, at this stage of our research we are interested in orders of complexity.

**Definition 8 (Covered Arcs).** *Let $G_{ac}$ be an actual behaviour graph, $G_{as}$ be an assumption graph and $\simeq$ a partial equivalence relation such that $G_{ac} \simeq G_{as}$, then we say that an arc $(\mu, l, \mu') \in A_{G_{as}}$ is covered if $\mu, \mu' \in \simeq$ and $\mu \notin S_{G_{as},\simeq}$.*

**Definition 9 (Checking Algorithm).** *Let $\Gamma_{ac} = \{G_{ac_1}, G_{ac_2}, \ldots, G_{ac_n}\}$ be a set of AC graphs and $\Gamma_{as} = \{G_{as_1}, G_{as_2}, \ldots, G_{as_n}\}$ the set of corresponding AS graphs.*

1. *Let $G'_{ac_i} = G_{ac_i}$ for every $G_{ac_i} \in \Gamma_{ac}$.*
2. *If $\Gamma_{as}$ is empty then*
   - *If $G'_{ac_i} \approx G_{ac_i}$ for every $G_{ac_i} \in \Gamma_{ac}$, return true.*
   - *Otherwise return false.*
3. *Try to find an AC graph $G_{ac_i}$ in $\Gamma_{ac}$, an AS graph $G_{as_j}$ in $\Gamma_{as}$, and a partial equivalence between them ($G_{ac_i} \simeq G_{as_j}$). If it is not found, return false.*

4. *Relabel with $\tau$ every arc in $G_{as_j}$ that is covered by $\simeq$. If all arcs in $G_{as_j}$ are labeled $\tau$ remove it from $\Gamma_{as}$.*
5. *If $S_{as} \cup S_{ac} \neq \emptyset$ then let $R_{G_{as_i}} = \{\mu \mid \mu \in S_{as} \vee \exists \nu \in S_{ac} : \nu \simeq \mu\}$ and let $R_{G_{ac_i}} = \{\nu \mid \nu \in S_{ac} \vee \exists \mu \in S_{as} : \nu \simeq \mu\}$.*
6. *Go to step 2.*

Note that the algorithm returns true only if all assumptions have been satisfied and the system configuration is in a setting equivalent to its initial state (Step 4).

We now apply the algorithm to the Compressing Proxy example. We represent partial equivalences with dotted lines for related nodes and crosses for stopping nodes. In figure 3, 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 4.

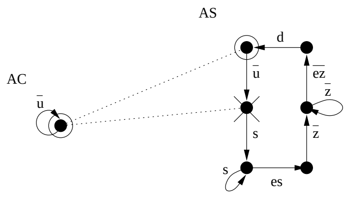

**Fig. 3.** Successful match of Upstream Filter AC and Adaptor AS graphs

Figure 4 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 labelled *es*. Hence the algorithm, after all possible attempts, terminates at Step 2 returning false; meaning that the proposed configuration is presumably not deadlock free.

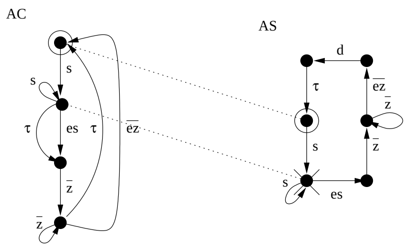

**Fig. 4.** Unsuccessful match of gzip AC and Adaptor AS graphs

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

As mentioned in Section 4, the adaptor must be modified to prevent system deadlock. We propose a new Adaptor component and show that the algorithm proves the new system is deadlock free.

**New Adaptor (NAD)**

NAD $\overset{def}{=}$ $u.\overline{s}.$ToGZ
ToGZ $\overset{def}{=}$ $\overline{s}.$ToGZ + $\overline{es}.z.$FromGZ + $\tau.z.$FromGZ
FromGZ $\overset{def}{=}$ $z.$FromGZ + $ez.\overline{d}.$NAD + $\tau.\overline{s}.$ToGZ

In figure 5, the partial equivalence that covers the $\overline{es}$ edge allows the New Adaptor AS graph to be updated, and Figure 6 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.

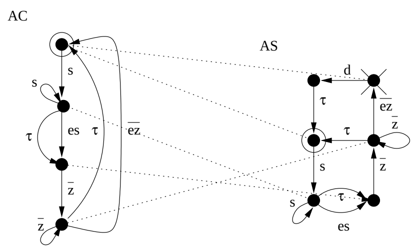

**Fig. 5.** Successful match of gzip AC and New Adaptor AS graphs

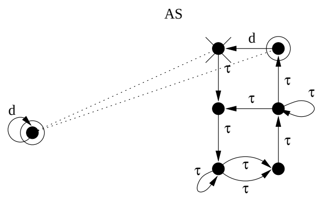

**Fig. 6.** Matching Downstream Filter AC and New Adaptor AS graphs

## 6 Method Assessment

Up to now, we have presented a method for checking deadlock freedom that trades off completeness for efficiency. We now show some evidence to substantiate the positive characteristics of our approach compared to existing ones. Because our work is in an initial stage, we cannot give, yet, empirical evidence, however a theoretical assessment on the completeness and complexity of our approach reveals that an implementation will yield good results.

**Complexity** The algorithm presented 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 behaviour graph and an assumed behaviour graph of its context.

In standard approaches, 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, worst case still requires the whole state space to be analysed, 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 $O(N^3K^4 log(K))$, 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 its complexity would be upper bounded by $O(K^2 log(K))$ [13]. 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 $O(N^2(K^2 log(K)))$. Finally, considering the worst case in which each partial match only covers a single arc of the $NK^2$ possibilities, we get $O(K^2N^3(K^2 log(K)))$ which reduces to $O(N^3K^4 log(K))$.

**Completeness** Completeness is an important issue in our approach. We have mentioned that our method is not complete and in this section we discuss how incomplete it is and possible improvements. With respect to correctness, the proof is sketched in [8].

There are two different sources of incompleteness in our approach. Firstly, because at the beginning of Section 5 we constrained the systems for which the method can be used. Secondly, because our checking algorithm may not be able to conclude deadlock freedom for some deadlock free systems.

The first restriction of Section 5 that requires components to be able to perform each computation an infinite number of times does not affect the completeness of our approach. The goal of this restriction is to simplify the presentation of definitions. Dropping the constraint requires defining the concept of recursive arcs as arcs that can be taken infinite times and modifying the matching scheme: recursive arcs of AS graphs can only be matched with recursive AC graphs while non-recursive arcs must match with non-recursive arcs. Definitions of this kind can be found in [9].

The restriction on shared channels is more serious. If a channel can be used by more than two components, there is potential *global* non-determinism in the overall system behaviour. A component may have the possibility of synchronizing with one of several components leading to a non-deterministic choice on the partner to which synchronize with. In terms of our approach, this means that one cannot commit to which AC graph will provide the AS graph requirements. Because 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. This non-determinism introduced by shared channels is similar to the non-determinism that makes our algorithm incomplete; therefore we will discuss a solution to both problems farther on in the section.

Having discussed the restriction imposed on components, the incompleteness of the checking algorithm remains. Our approach is intrinsically incomplete. We attempt to prove a global property such as deadlock freedom in terms of local properties of each component. That is we say that the system is deadlock free if no system component can ever block. Obviously this is a strong sufficient condition but by no mean a necessary one.

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. The algorithm reaches a state in which it cannot do further matches between AC and AS graphs. One reason for this is that the definition of partial match may be just too restrictive, leaving out some matches that might be correct. Compared to the previous version of this approach [9] the notion of partial match represents an improvement since it does not require matching between AC and AS graphs to use the entire AC graph to prove deadlock freedom. Examples that can be verified with the partial matching introduced in this paper and not with the previous versions of the approach can be found in [8].

However, the main reason for the incompleteness of our approach is connected to non-determinism. When there is a non-deterministic choice in component behaviour, when a component can interact with one of two different components, there can be not a unique matching 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 (Note that this is still less restrictive than the approach in [2]). In order to solve this problem we are working in the direction of changing the main algorithm in the following way: When there is a choice in the AS graph and it is not possible to match all choices simultaneously, the choices must be selected in turns. For each selection, the algorithm proceeds matching. If the checking algorithm succeeds matching all AS graphs, it tries the rest of the choices. If checking succeeds for all choices, then it succeeds for the complete AS graph. This improvement on completeness does not have a drastic impact on complexity. In terms of time complexity, there is an important increase; however, in terms of space complexity, the modification presents no significant changes. No new graphs must be represented, the algorithm has to register the non-deterministic points and go back to them. Thus, space complexity will not jump to an exponential order, maintaining the advantages the approach has with respect to other methods.

Summarizing, the approach we present is incomplete but we think there is room for significant improvements. On the other hand, 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 complexity of the solution from exponential order to a polynomial one.

## 7 Conclusions and Future Work

In this work we have presented a broader notion of component semantics based on assumptions and a derived space-efficient 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 behaviour analysis, 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. We think it can be a useful tool to be included in a verification tool-set together with complete but not always applicable ones.

Our approach heavily relies on the component-based setting. This is a very interesting context in which experimenting new verification techniques. In fact, on one side components by definition force standardization and therefore simplifications of the integration frameworks. On the other side there is room for suggestions on the kind of information that a component should explicitly carry on with it in order to be integrated in all suitable contexts. That is the notion of component semantics is conceived in broader terms than in traditional programming.

Dynamic properties are difficult to be proven and, as we discussed in Section 2, most of the proposed approaches to overcome state explosion are based on characterizing local properties and then try to ensure that these properties can be lifted up to the global system level. Our contribution is actually in this line, but with the aim of fixing once and for all the kind of information that a component has to carry with it independently of the contexts it will eventually be used. To this respect, we believe that assumptions are a good way to extend component semantics in order to verify properties more efficiently.

Ongoing and future work goes in several directions. Firstly, we are working on the validation of the framework through experimental results. We are currently working on an implementation of the algorithm, and considering other coordination contexts like non-fully synchronized or asynchronous ones. In particular we are trying to cast our approach in given architectural styles and experiment with commercial component base frameworks as COM. Second, we wish to extend the approach to deal with other properties such as general liveness and safety properties. To this respect we are thinking of general safety properties expressed with property automata that may be decomposed into component assumptions or specific component assumptions such as particular access protocols for shared resources. Lastly, we are working on an extension of the algorithm in order to improve completeness of our approach.

## Acknowledgements

We would like to thank Alexander Wolf and Daniel Yankelevich for discussions and previous work on the subject of the paper.

## References

1. R. Alur, L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Automatic modular verification. In *Proceedings of CONCUR '99: Concurrency Theory*, 1999.
2. F. Arbab, F.S. de Boer, and M. M. Bonsangue. A logical interface description language for components. In COORDINATION'00, vol. 1906 of *LNCS*. Springer, 2000.
3. 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.
4. 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.
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*, 6(1):7–35, 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, Italia, October 1999. http://www.doc.ic.ac.uk/ su2/pubs/techrep99.pdf
9. P. Inverardi, D. Yankelevich, and A. Wolf. Static checking of systems behaviors using derived component assumptions. *ACM TOSEM*, 2000.
10. N. Kobayashi. A partially deadlock-free typed process calculus. *ACM Transactions on Programming Languages and Systems*, 20(2):436–482, March 1998.
11. R. Milner. *Communication and Concurrency.* Prentice Hall, New York, 1989.
12. O.Grumberg and D.E.Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, May 1994.
13. R. Paige and R.E. Tarjan. Three partition refinement algorithms. *SIAM Journal on Computing*, 16(6):973–989, 1987.
14. Clemens Szyperski. *Component Software. Beyond Object Oriented Programming.* Addison Wesley, Harlow, England, 1998.