Modular Consistency Analysis of Component-Based Designs

Yan Jin, Charles Lakos and Robert Esser
School of Computer Science, University of Adelaide
Adelaide, SA 5005, Australia
{yan, charles, esser}@cs.adelaide.edu.au

In this article, we present a practical analysis approach that makes use of the modular nature of component-based designs to alleviate the state space explosion problem, a well-known obstacle to system verification. The key is to specify interaction protocols for components using a lightweight formal language and then to utilise these protocols as contracts for independent analysis of the components and their interactions. These protocols are often missing or informally documented. It is demonstrated how consistency properties of closed and open component-based designs can be verified using this divide-and-conquer approach.

ACM Classification: F.3.1 (Logics and Meanings of Programs – Specifying and Verifying and Reasoning about Programs), D.2.4 (Software Engineering – Software/Program Verification), D.2.11 (Software Engineering – Software Architectures), D.2.12 (Software Engineering – Interoperability), F.1.1 (Computation by Abstract Devices – Models of Computation)

1. INTRODUCTION

In recent years, component-based software development has become more popular for the production of large-scale software applications. By building systems from independently developed components, a promising means of achieving software reuse, rapid development and complexity management is provided. However, as noted by Bérard et al (2001):

“system complexity, and hence the likely number of design errors, grows exponentially with the number of interacting system components”.

This is a consequence of the well-known state space explosion problem and it largely limits the applicability of exhaustive analysis. To overcome this problem, various reduction techniques have been proposed in the literature. Among them, modular analysis (or compositional verification) (Long, 1993; Grumberg and Long, 1994; Graf et al, 1996; Henzinger et al, 1998; McMillan, 1998; Alur and Henzinger, 1999; Cheung and Kramer, 1999; Henzinger et al, 2000) is a powerful divide-and-conquer technique for decomposing the analysis task of a system into subtasks of individual components. The key to this is to consider each component in conjunction with assumptions about the context of the component, and to consider the composition of components in conjunction with their interface behaviour.

Copyright© 2004, Australian Computer Society Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that the JRPIT copyright notice is given and that reference is made to the publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Australian Computer Society Inc.

Manuscript received: 2 July 2003
Communicating Editor: Sidney A. Morris
In component-based systems, however, this key information is often missing or only informally described. Currently, the interface specifications of components tend to be rather restricted, capturing only the signatures, i.e. the names, data types and direction of information flow, but excluding information about the communication protocols of components. This is because software engineers lack a formal means for precisely specifying the interfaces behind which components encapsulate their services. As a result, components cannot be analysed independently due to the lack of information about the environments in which they are embedded, and the composition of components cannot be analysed due to the lack of rigorous specifications of the interface behaviour of components.

In this article, we present a formal technique which focuses on communication protocols while abstracting away from the data values being communicated. The protocol of a component describes the services it provides, the way it reacts to its inputs and what it expects from its environment. It does not, however, disclose the implementation detail of the component. We use interface automata (IAs), a formal lightweight language proposed by de Alfaro and Henzinger (2001), as the notation for describing protocols.

With the contextual assumptions captured by an interface automaton (IA), each component can be checked for conformance with the IA in isolation from the system. This ensures that a component is able to abide by the interaction protocol given by the IA, provided its environment behaves as expected. Furthermore, the composition of components can be analysed utilising the interface behaviour of components specified by the IAs while disregarding the internal activities of components. Using this divide-and-conquer approach, the state space explosion problem can be alleviated. This approach has been implemented in the context of the Moses tool suite (Esser and Janneck, 2001) and has also been applied to a non-trivial case study: the production cell (Jin et al., 2003c).

This article constitutes a more detailed presentation of earlier publications (Jin et al., 2003a; Jin et al., 2003b). It reformulates that work in the regular framework of Arnold and Nivat; it uses the notion of communication ports for passing information between processes (or components); it takes the more realistic approach of constraining the type of messages that are transferred via each port; and it extends the material on verifying properties of component networks.

The rest of this article is structured as follows. In Section 2, this approach is compared with related work. In Section 3, the underlying concepts such as general reactive systems and networks are introduced. In Section 4, discrete-event components and interface automata are defined followed by the conformance relation between them. In Section 5, networks of discrete-event components are clarified and a modular analysis method for determining their consistency is presented. Finally, conclusions are drawn in Section 6. The proofs of key theorems are found in the appendix.

It should be emphasized that throughout this article the word analysis is used in the sense of verifying system properties rather than in any sense of requirements elicitation.

2. RELATED WORK
Interface automata were first introduced by de Alfaro and Henzinger (2001). The authors established a simple and well-defined semantics for them and defined their composition by two-party synchronisation. Also, alternating simulation was proposed to determine a refinement relationship between interface automata. This relationship takes an optimistic view of the environment by assuming that it is always helpful, only supplying inputs expected by an automaton. This optimistic view allows more possible implementations than a pessimistic approach where the environment can behave as it pleases.

We take this optimistic view and adapt alternating simulation to define the conformance of components with interface automata, taking into account data values used in components. Also, a
computationally efficient method of checking the relation is presented, which does not require the construction of the Cartesian product of their state spaces as in de Alfaro and Henzinger (2001) – only the reachable states in the product need to be constructed. Additionally, in contrast to the simple composition scheme in de Alfaro and Henzinger (2001), we allow interface automata to be composed in terms of synchronisation vectors, a more general composition mechanism introduced by Arnold and Nivat (Arnold, 1994).

Our method of checking alternating simulation is inspired by Rajamani and Rehof (2002), where a similar relation was proposed to check the conformance (or refinement) relationship between CCS models. However, this relation is more restricted than ours, as it requires both specification and implementation models to have no mixed states (where both input and output transitions can occur), while in our approach this is not required. Furthermore, because of the presence of blocking outputs, models in their approach are different in nature from ours where a component is in full control of its outputs.

There are some other approaches which also utilise the environmental assumptions of components for verification. In Inverardi and Uchitel (2001) and Inverardi et al (2000) the assumptions and the actual behaviour of components are derived from component specifications. The system property is then determined by pairwise matching between the assumptions of a component and the actual behaviour of another component. However, the proposed method is incomplete and limited to one-to-one communication or synchronisation.

The approach in Uchitel and Yankelevich (2000) requires a separate model of the environment, which captures the environmental assumptions made by a component. In contrast to interface automata used in our approach, the model only specifies the component assumptions about the temporal relation between the input events. Thus it cannot be used as an abstraction of the component for the system analysis. As a result, the global state space, though constrained by the assumption models, still needs to be built. This would easily lead to the state space explosion.

3. PRELIMINARIES
This section introduces general definitions for reactive transition systems and networks composed of reactive transition systems. These definitions are set in the framework of Arnold and Nivat. Subsequent sections will specialise these definitions for discrete-event components and interface automata.

3.1 Reactive Transition Systems
Definition 1. A reactive transition system (RTS) is defined as \( L = (s^0, S, \Sigma, \Delta) \), where

- \( S \) is a set of states and \( s^0 \in S \) is the initial state;
- \( \Sigma \) is a set of events, consisting of three mutually disjoint sets of input events \( \Sigma^I \), output events \( \Sigma^O \), and internal events \( \Sigma^I \);
- \( \Delta \subseteq S \times \Sigma \times S \) is a set of steps.

RTSs are similar to labelled transition systems (LTSs) (Arnold, 1994) which have been used to give the operational semantics of many modelling languages, e.g. CSP (Roscoe, 1997) and CCS (Milner, 1989). RTSs differ from LTSs in having an explicit distinction between input, output and internal events (called labels in LTSs). The distinction reflects the fact that, in an asynchronous distributed application, a system has control over its internal and output events, but no control over
Modular Consistency Analysis of Component-Based Designs

its input events. Instead, when an input event occurs is under the control of the environment. That is, a system decides when to produce an output, while the environment decides when to provide an input. Hence we let $\Sigma^{ctrl} = \Sigma^O \cup \Sigma^H$ be a set of controllable events of $L$ and $\Sigma^{obs} = \Sigma^I \cup \Sigma^O$ be a set of observable events.

A RTS $L$ is finite if both $S$ and $\Sigma$ are finite. Also, $L$ is deterministic if $\forall e \in \Sigma, s, s', s'' \in S, (s, e, s') \in \Delta \land (s, e, s'') \in \Delta$ implies $s' = s''$. Otherwise $L$ is nondeterministic.

**Definition 2.** A trace $\sigma$ of a RTS $L$ from $s_1 \in S$ is an event sequence $e_1 e_2 \ldots e_m$ such that $\exists s_2, \ldots, s_{m+1} \in S, \forall j: 1 \leq j \leq m, (s_j, e_j, s_{j+1}) \in \Delta$. State $s_{m+1}$ is called reachable from $s_1$ (via $\sigma$), or reachable in $L$ if $s_1 = s^0$. Trace $\sigma$ is said to be internal if $\forall j: 1 \leq j \leq m, e_j \in \Sigma^H$, or to be empty if $m = 0$. An empty trace is denoted as $\lambda$. Given a set of events $E$, the event restriction $\sigma \mid E$ is an event sequence obtained by removing from $\sigma$ all events not in $E$.

In the following, we write $s \xrightarrow{e} s'$ as a shorthand for $(s, e, s') \in \Delta$, and $s \xrightarrow{e} s'$ for $e \in \Sigma^{obs}$ if $\exists s'' \in S, s'' \xrightarrow{e} s'$ and $s''$ is reachable via a (possibly empty) internal trace of $L$ from $s$.

Given a state $s \in S$, the sets of enabled input and output events at $s$ are defined by $en^I(s) = \{ e \in \Sigma^I \mid \exists s' \in S, s \xrightarrow{e} s' \}$ and $en^O(s) = \{ e \in \Sigma^O \mid \exists s' \in S, s \xrightarrow{e} s' \}$, respectively. An input event $e \in \Sigma^I$ is called refused at $s$ if $e \notin en^I(s)$. A RTS $L$ is called input-universal if $\forall s \in S, en^I(s) = \Sigma^I$. In our approach, it is important to develop two derivatives for RTSs: mirrors and input-universal versions. Mirrors will provide the ideal environments for RTSs, while input-universal versions will make RTSs able to accept all input events by incorporating trap steps.

**Definition 3.** Given a RTS $L$, the mirror of $L$ is a RTS $M = (s^0_L, S_L, \Sigma_M, \Delta_L)$, where $\Sigma_M = \Sigma_L^O$ and $\Sigma_M^O = \Sigma_L^I$.

The mirror of $L$ is identical to $L$ but has the input and output events of $L$ interchanged.

**Definition 4.** Consider a RTS $L$. Let $\perp \notin S_L$ be a single error state, $\Delta_{trap} = \{ (s, e, \perp) \mid s \in S_L, e \in \Sigma_L^I \land en^I(s) \} \land \Delta_{idle} = \{ (\perp, e, \perp) \mid e \in \Sigma_L^I \} \land \Delta_{idle}$ a set of trap steps, and $\Delta_{idle} = \{ (\perp, e, \perp) \mid e \in \Sigma_L^I \} \land \Delta_{idle}$ a set of idle steps. Then the input-universal version of $L$ is a RTS $U$ such that

$$U = \begin{cases} (s^0_L, S_L \cup \{ \perp \}, \Sigma_L, \Delta_L \cup \Delta_{trap} \cup \Delta_{idle}) & \text{if } \Delta_{trap} \neq \emptyset, \\ (s^0_L, S_L \cup \{ \perp \}, \Sigma_L, \Delta_L \cup \Delta_{idle}) & \text{otherwise.} \end{cases}$$

The input-universal version is a RTS that includes not only the existing steps but also new trap steps. A trap step, taken when an unspecified input event is received, will cause the error state $\perp$ to be entered. Clearly, for every state-event pair $(s, e)$ with $s \in S$ and $e$ a refused input event at $s$, a step is added to $\Delta_{trap}$ which emanates from $s$, receives $e$ and enters $\perp$. On the other hand, we have ensured that the input-universal version of an (already) input-universal RTS is the same RTS.

### 3.2 RTS Networks

We define the composition of RTSs in terms of synchronisation vectors introduced by Arnold and Nivat (Arnold, 1994). Their approach to synchronisation was quite general in that any non-empty set of processes may synchronise on the events named in so-called synchronisation vectors. In this way, not only peer-to-peer but also multicast and broadcast communication among processes can be
Modular Consistency Analysis of Component-Based Designs

described. Our approach differs from Arnold and Nivat in differentiating input and output events, since this is an important issue for distributed component-based systems. Components should be able to accept inputs at any time, but may be selective in when they produce outputs. Also, we insist that each synchronisation vector should have exactly one output event and an arbitrary number of synchronised input events. To define RTS networks, we introduce a preliminary definition to help clarify the nature of synchronisation vectors.

Definition 5. Consider mutually disjoint sets \( E_0, E_1, \ldots, E_n \), a relation \( R \subseteq \prod_{0 \leq i \leq n} E_i \), and a set \( E \subseteq \bigcup_{0 \leq i \leq n} E_i \). Let projections \( \pi_i : R \rightarrow E_i \) for \( 0 \leq i \leq n \) and sets of keys \( \kappa_r = \{ \pi_i(r) \in E \mid 0 \leq i \leq n \} \) for \( r \in R \). Then \( R \) is said to be indexed by \( E \) if there exists:

- Exactly one key per tuple: \( |\kappa_r| = 1 \) for all \( r \in R \);
- Exactly one occurrence per key: \( \forall e \in E, \exists r \in R, e \in \kappa_r \land \forall r' \in R \setminus \{ r \}, e \notin \kappa_{r'} \).

We will use such a relation to capture the synchronisation patterns between RTSs. These patterns may have multiple input events but exactly one output event, which will then individually provide a key and cumulatively provide the index for the relation. Also, every output event is involved in exactly one synchronisation pattern. In defining RTS networks, we assume a special symbol \( \epsilon \) which is not an event of any RTS.

Definition 6. A RTS network is a tuple \( N = (\Sigma, W, R) \), where

- \( \Sigma \) is a set of external events of the network, consisting of two disjoint sets of input events \( \Sigma^I \) and output events \( \Sigma^O \). We let \( \Sigma^d = \Sigma \cup \{ \epsilon \} \);
- \( W \) is a finite set of RTSs. We let \( \Sigma^d_l = \Sigma^o_l \cup \{ \epsilon \} \) for all \( l \in W \);
- \( R \subseteq \Sigma^d \times \Pi_{l \in W} \Sigma^d_l \) is a relation indexed by \( \Sigma^I \cup \bigcup_{l \in W} \Sigma^O_l \).

A RTS network is called closed if \( \Sigma = \emptyset \), or open otherwise. We often write \( N = (W, R) \) for a closed RTS network \( N \). The relation \( R \) is called a set of synchronisation vectors (in the terminology of the Arnold and Nivat model). A synchronisation vector in the set describes a particular synchronisation pattern between the component RTSs. More specifically, the RTSs with events present in the vector are synchronised, while the other RTSs marked by \( \epsilon \) remain unchanged. The symbol \( \epsilon \) represents the irrelevancy of a RTS to a particular synchronisation. Further, the RTS corresponding to the network events is referred to as the environment, denoted by “env”. Its output events denote the network input events and its input events denote the network output events, i.e. \( \Sigma^o_{\text{env}} = \Sigma^I \) and \( \Sigma^o_{\text{env}} = \Sigma^O \). Note that env is an unknown RTS and thus we only use it as a syntactic term to facilitate further explanation.

In order to give RTS networks a sound interleaving semantics, we require exactly one output event in each synchronisation vector, because output is nonblocking in asynchronous applications. Also, we require that at most one synchronisation vector can be matched for a particular output event. Hence the synchronisation vectors \( R \) are indexed by the output events (of the environment or the component RTSs). In each synchronisation vector, the unique output event is called the produced event and the input events in the vector are the consumed events of the synchronisation. Also, the RTS that produces the produced event is called the producer, the RTSs that receive the consumed events are the consumers, and the rest are the idlers of the synchronisation. A formal specification is given as follows.
Before giving the semantics of general RTS networks, we first consider networks of input-universal RTSs.

**Definition 7.** Consider a RTS network \( N \) and a synchronisation vector \( r \in R \). Let \( W' = \{ \text{env} \} \cup W, l \in W', \) and projections \( \pi_l : R \to \Sigma_l \) for all \( l \). Then the producer of \( r \), denoted by \( \rho_r \), is the unique RTS \( l \) such that \( \pi_l(r) \in \Sigma_l^{\rho_r} \). \( \pi_l(r) \) is called the produced event of \( r \). Also, the set of consumers of \( r \), denoted by \( \eta_r \), consists of all RTSs \( l \) such that \( \pi_l(r) \in \Sigma_l^l \). For any consumer \( l \), the event \( \pi_l(r) \) is called a consumed event of \( r \). In addition, an idler of \( r \) is a RTS \( l \) such that \( \pi_l(r) = \epsilon \). The set of idlers is denoted by \( \varphi_l \).

According to the definition, such a RTS network defines a RTS \( N = (\Sigma, W, R) \) such that all \( l \in W \) are input-universal.

**Definition 8.** Consider a RTS network \( N = (\Sigma, W, R) \) such that all \( l \in W \) are input-universal. The synchronised product of \( N \) is a RTS \( L = (s, \Sigma, \Sigma, \Delta) \), where

- \( s = \prod_{l \in W} s_l \) and \( \Sigma \subseteq \prod_{l \in W} S_l \) is the smallest set such that \( s \in S \) and \( s_l(s, e, s') \in \Delta \) implies \( s' \in S \). We assume projections \( \pi_l : S \to S_l \) and let \( s_l = \pi_l(s) \) and \( s'_l = \pi_l(s') \) for \( l \in W, s, s' \in S \);
- \( \Sigma_l^l = \Sigma_l \Sigma_l^\rho_l, \Sigma^\rho = \Sigma_N^\rho, \Sigma \subseteq \bigcup_{l \in W} \Sigma_l^{\text{out}} \);
- \( \Delta \) consists of input steps

\[
\{(s, e, s') \mid e \in \Sigma_l^l, \exists r \in R, \rho_r = \text{env} \land e = \pi_l(r) \land \forall g \in \eta_r, (s_g, \pi_l(r), s'_g) \in \Delta_l \land \forall h \in \varphi_r, s_h = s_h\}, \tag{1}
\]

output steps

\[
\{(s, e, s') \mid e \in \Sigma_l^l, \exists r \in R, l \in W, l = \rho_r \land e = \pi_l(r) \land (s_l, \pi_l(r), s'_l) \in \Delta_l \land \forall g \in \eta_r, (s_g, \pi_l(r), s'_g) \in \Delta_l \land \forall h \in \varphi_r, s_h = s_h\}, \tag{2}
\]

and internal steps

\[
\{(s, e, s') \mid \exists l \in W, e \in \Sigma_l^\rho \land (s_l, e, s'_l) \in \Delta_l \land \forall g \in W \setminus \{l\}, s'_g = s_g\} \tag{3}
\]

\[
\cup \{(s, e, s') \mid \exists r \in R, l \in W, \pi_l(r) = \epsilon \land l = \rho_r \land e \in \Sigma_l^\rho \land (s_l, \pi_l(r), s'_l) \in \Delta_l \land \forall g \in \eta_r, (s_g, \pi_l(r), s'_g) \in \Delta_l \land \forall h \in \varphi_r, s_h = s_h\}, \tag{4}
\]

According to the definition, such a RTS network defines a RTS (later called the composite RTS). A state of the RTS is a vector of states of all its component RTSs (later called the components), and its initial state is a vector of initial states of the components. An input step of the composite RTS is a step receiving a network input event. Since there must exist a synchronisation vector where env is the producer and the event is the one produced (a requirement for \( R \)), the input step also synchronises the corresponding input steps of the consumers (Formula 1). Further, the composite RTS will perform an output step when the producer of a synchronous vector generates the produced event, provided that env is a consumer of the vector (Formula 2). The step involves the corresponding output step of the producer synchronised with the corresponding input steps of the consumers including the environment. Finally, an internal step taken by any component is an internal step of the composite RTS (Formula 3). This step clearly has no impact on other components or the environment. Also, an output step taken by a component becomes internal to the composite RTS, when the corresponding synchronisation vector involves no network events (Formula 4). Similarly, the step can also synchronise the corresponding input steps of the consumers.
One can see that the composite RTS involves not only synchronised communications among the components but also interleavings of their internal activities. The latter are the main contributors to the state space explosion when a network is directly analysed and thus our approach seeks to minimise their effect. We now generalise the synchronised product to an arbitrary RTS network by the following definition.

**Definition 9.** Consider a RTS network \( N = (\Sigma, W, R) \). Let \( N' = (\Sigma, W', R) \) be a derived RTS network such that \( W' = \{ U_l \mid l \in W \} \), where \( U_l \) is the input-universal version of \( l \). Then the synchronised product of \( N \) is defined by the synchronised product of \( N' \) (Definition 8).

The above definition acknowledges the fact that in asynchronous systems the reception of an unspecified input event often indicates a design error. Immediately from Definitions 8 and 9, we can have the following proposition.

**Proposition 1.** The synchronised product of any RTS network is input-universal.

To relate the behaviour of a RTS network to that of its components, we define trace projections on components.

**Definition 10.** Consider a RTS network \( N = (\Sigma, W, R) \) and a RTS \( l \in W \). Let \( L_N \) be the synchronised product of \( N \), \( \sigma \) a trace of \( L_N \) from \( s^0_N \), and

\[
\psi = \{(o, i) \mid \exists r \in R, l' \in \{\text{env}\} \cup W, o \in \Sigma^O_{l'}, \pi_{l'}(r) = o \land i \in \Sigma^I_{l'}, \pi_l(r) = i\}
\]

a function mapping an output event of any other RTS, which can cause the RTS \( l \) to take an input step, into the corresponding input event of \( l \). Then the trace projection \( \pi_l(\sigma) \) of \( \sigma \) on \( l \) is defined by

\[
\pi_l(\sigma) = \begin{cases} 
\lambda & \text{if } \sigma = \lambda, \\
e \cdot \pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land e \in \Sigma^e_{l}, \\
i \cdot \pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land i \in \Sigma^i_{l} \land (e, i) \in \psi, \\
\pi_l(\sigma') & \text{if } \sigma = e \cdot \sigma' \land e \notin \Sigma^e_{l} \land \exists (e, i) \in \psi.
\end{cases}
\]

The trace projection \( \pi_l(\sigma) \) on \( l \) is an event sequence consisting of the events that \( l \) takes while the network \( N \) follows trace \( \sigma \) from its initial state \( s^0_N \). It is computed recursively by considering each event \( e \) in \( \sigma \) in turn. If \( e \) is an internal or output event of \( l \), then \( e \) is appended to the trace projection; if \( e \) is an output event of other RTSs which corresponds to a synchronisation vector \( r \) with \( \pi_l(r) \) an input event of \( l \), then \( \pi_l(r) \) is appended to the trace projection; otherwise \( e \) is ignored. Note we have ensured that \( \pi_l(\sigma) \) is a trace of \( l \) from \( s^0_l \).

### 4. INDEPENDENT ANALYSIS OF COMPONENTS

In this section, general RTSs are specialised as discrete-event components and as interface automata. Also, the conformance relation between these is studied and a practical conformance checking method is presented.

#### 4.1 Discrete-Event Components

As RTSs are often too elementary for practical use, we extend these systems in two ways. First of all, we redefine an event to consist of two parts including a kind and a value, and associate each part
with different importance depending on the context. Kinds are used to classify events while values represent data being communicated (also called event parameters). This allows us to introduce a level of abstraction to component behaviour and vary the level of abstraction by changing the way events are partitioned. In addition, we extend RTSs with input/output ports, each port identifying a particular kind of events. Then it can be assumed that a component (or a reactive system) communicates with others only through its ports. That is, a component always receives data (or messages) fed to its input ports and produces data (or messages) via its output ports. In this way, event kinds (or ports) play a critical role in defining component composition but have little impact on the computation of individual components, while event values are of great importance to component computation but less relevant to component composition. This facilitates the system modelling in that the ports can form a component’s view of the rest of the system and decouple the outside world from the component. It thus largely reduces the interdependency between components. This approach also allows the designer to compose components by simply connecting ports, and thus relieves them from the labor of relating individual events. In defining components, we assume a countable universe of transmitted values $\mathcal{V}$.

**Definition 11.** A discrete-event component (DEC) is defined as $C = (\alpha, \theta, s^0, S, \Sigma, \Delta)$, where

- $\alpha$ is a finite set of ports, consisting of two disjoint sets of input ports $\alpha^I$ and output ports $\alpha^O$;
- $\theta : \alpha \rightarrow 2^\mathcal{V}$ is a total function, mapping each port to a subset of values from the universe;
- $\Sigma = \Sigma^I \cup \Sigma^O \cup \Sigma^H$ is a set of events, where $\Sigma^f = \{(f,v) \mid f \in \alpha^I, v \in \theta(f)\}$, $\Sigma^O \subseteq \{(f,v) \mid f \in \alpha^O, v \in \theta(f)\}$ and $\Sigma^H \cap (\alpha \times \mathcal{V}) = \emptyset$;
- The tuple $(s^0, S, \Sigma, \Delta)$ forms an input-universal RTS.

A DEC is called *closed* if $\alpha = \emptyset$, or *open* otherwise. $\theta$ is called the **typing** function, associating each port with a type (or kind) of values that can be transmitted via the port. An input/output event of a DEC is regarded as an occurrence of message transfer at an input/output port of the DEC, while internal events of each DEC are considered to be unique to that DEC. Also, like Lynch (1996, Chapter 8), we require DECs to be input-universal RTSs. This acknowledges the fact that components are often developed to work properly in unknown environments in bottom-up design. This is also a requirement for independent deployment of components. In the following, we abbreviate $\langle f,v \rangle$ to $f.v$ for $\langle f,v \rangle \in \Sigma^I \cup \Sigma^O$.

Figure 1 models a DEC in a compact form, where black bullets, arcs and triangles represent states, steps and ports, respectively. In particular, the initial state is pointed to by an arrow with no source. Input ports are listed at the left, e.g. “$a$” and “$b$”, and output ports at the right, e.g. “$c$”. Following the port names, types of transmitted values are specified. In this case, all values must be integers. In addition, the text attached to an arc denotes a parameterized event. For instance, $a.x?$ denotes an event accepting an integer from $a$ represented by $x$, while $c.(x+y)!$ indicates an event producing an integer via $c$, which is the sum of $x$ and $y$. Note that the actual size of the DEC’s state space depends on the ranges of $x$ and $y$.

This figure describes an adder which accepts two parameters respectively from the two input ports and then reports their sum via the output port. The adder expects the environment to behave in a certain way in order to function properly. For instance, it expects to be fed with only one parameter of each kind before producing the sum. However, it is often required for a component to be robust and able to work in all kinds of environments. Hence the example component also
performs some tasks such as error reporting or recovery (denoted by grey arcs) if the environment does not behave as it expects, e.g. providing two values of kind $a$ or $b$ in a row. Since the way these tasks are specified is not the focus of this article, we have omitted the detail.

It should be noted that even though robustness is required of DECs, the system designer often does not want the “grey” tasks to be executed, since it prevents the DECs from providing their normal functionality to the system. Also, in practice the portion of a DEC providing the normal functionality is usually intensively tested or verified, whereas the other portion is not and thus could contain potential bugs. Once encountered, these bugs may cause problems, sometimes even disastrous consequences. These concerns highlight one of the main goals of this article, which is to ensure that the branches irrelevant to the functionality needed in a component-based design, e.g. the grey tasks, will never be executed.

4.2 Interface Automata

Assembling events by kinds makes it possible to specify the interaction protocols expected by the designer of components in terms of temporal relations between different kinds of events, while abstracting away from the specific values. The protocols are very useful for guiding the development or selection of individual components and for further study of a system design, and thus should be formally specified. We employ a restricted version of interface automata (de Alfaro and Henzinger, 2001) to serve this purpose. We constrain interface automata to be deterministic and to have no internal events. We believe that this version is sufficiently expressive for our purpose and, as shown later, this restriction leads to an efficient conformance checking method.

**Definition 12.** An interface automaton (IA) is defined as a finite deterministic RTS $A = (s^0, S, \Sigma, \Delta)$, where $\perp \notin S$ and $\Sigma^H = \emptyset$. We let $U^ia$ be a universal set of IAs.

Let the events of IAs correspond to the ports of DECs. Then the information conveyed by an IA is twofold. On the one hand, it restricts the kinds of output events that a component under consideration can produce. On the other hand, it states the component’s assumption that the environment never provides input events of an unspecified kind. As an example, suppose we get an IA shown in Figure 2 when decomposing a system. Then an assumption is captured that the environment always provides an event of kind $a$ followed by an event of kind $b$ and then waits for an event of kind $c$ before providing any further events. Also, it is guaranteed that the component under development or selection must not produce any output before getting both $a$ and $b$.

Comparing with Figure 1, we can learn that the DEC can be an implementation of this IA in that the DEC does not break the guarantee expressed by the IA and can be used in an application where the environment does not break the assumption captured by the IA. We shall formally study this...
relationship in a general context in the next section. Moreover, we emphasize that IAs are inter-
mediate products of top-down design capturing the interface and the protocol but not the
implementation, while DECs are building blocks for bottom-up design.

While abstracting away implementation details such as the data being communicated, IAs can
describe the interaction protocols expected of components at a high level of abstraction. We shall
show that the use of IAs enables modular analysis of component-based designs. It is worth noting that
the abstraction is a relative issue in that the level of abstraction heavily depends on the way events
are partitioned. The level of abstraction becomes greater as we classify more events into each kind.

To facilitate further study, we define the most abstract implementation for an IA, which includes
the IA events as ports and is able to produce all possible data values accompanying an enabled IA
output event.

**Definition 13.** Consider an IA $A$ and a total typing function $\theta: \Sigma_A \to 2^\gamma$. Let $U$ be the input-
universal version of $A$, then the most abstract implementation (MAI) of $A$ (with respect to $\theta$) is a
DEC $J = (\alpha_f, \theta, s_f, S_f, \Sigma_f, \Delta_f)$, where

- $\alpha_f^I = \Sigma_A^I$ and $\alpha_f^O = \Sigma_A^O$;
- $\Sigma_f^I = \{ (f, v) \mid f \in \Sigma_A^I, v \in \theta(f) \}$, $\Sigma_f^O = \{ (f, v) \mid f \in \Sigma_A^O, v \in \theta(f) \}$ and $\Sigma_f^H = \emptyset$;
- $\Delta_f = \{ (s_f, f, v, s') \mid (s, f, s') \in \Delta_U, v \in \theta(f) \}$.

The MAI of an IA is an input-universal DEC. Similar to the input-universal version of the IA,
after getting an unspecified input event with an arbitrary value, the MAI goes to the error state,
where it can accept all input events but never produce outputs or move out of the error state.

### 4.3 Conformance of Discrete-Event Components

The employment of IAs for guiding component development or selection leads to an important
aspect of this approach, viz. the conformance of DECs with IAs. The intention is that an IA can
safely be implemented by a conforming DEC without compromising the system safety properties
that hold for the IA, particularly in a network of IAs.

The conformance cannot be defined by traditional refinement relations, e.g. trace containment
and simulation, because they only allow the implementation to have less input and output behaviour
than the specification, whereas DECs are able to handle more inputs than IAs. Instead we adopt
alternating simulation (de Alfaro and Henzinger, 2001), a relation based on an optimistic view of
the environment. More specifically, the implementation is always assumed to run in an environment
where the assumption of the specification is respected. In this way, the implementation can offer
less outputs (since it will be less likely to violate the system safety) and accept more inputs (since
the environment will not offer them). In the following, we extend the relation to accommodate the
implementation with data values, as required by our definition of DECs.

**Definition 14.** Consider an IA $A$ and a DEC $C$ such that $\Sigma_A^I \subseteq \alpha_C^I$ and $\Sigma_A^O \supseteq \alpha_C^O$. $C$ conforms to
$A$, written $C \preceq A$, if there exists an alternating simulation relation $\Delta_C \subseteq \Delta_A \times \Delta_A$ such that $s_0^C \preceq s_0^A$ and for $q \preceq s$, the following conditions hold:

1. $\forall e \in \Sigma_A^H$, $\exists (q, e, q') \in \Delta_C$ implies $q' \preceq s$;
2. $\forall f, v \in \Sigma_A^O$, $\exists (q, f, v, q') \in \Delta_C$ implies $\exists (s, f, s') \in \Delta_A$ where $q' \preceq s'$;
3. $\forall f \in \Sigma_A^I$, $\exists (s, f, s') \in \Delta_A$ implies $\forall v \in \theta_C(f)$, $(q, f, v, q') \in \Delta_C$ where $q' \preceq s'$. 
For a DEC state \( q \) to simulate an IA state \( s \), first of all, the resultant DEC state must simulate the previous IA state \( s \) after the DEC takes an internal step from \( q \) (Condition 1). Also, the DEC must not produce an output event that the IA cannot produce (Condition 2). Note that Conditions 1–2 implies that \( f \in \text{en}_{A}^{O}(s) \) for all \( f.v \in \text{en}_{C}^{O}(q) \). Further, the resultant DEC state must simulate the resultant IA state after the DEC takes an event \( f.v \) (for any value \( v \)) from \( q \) and the IA takes the event \( f \) (Conditions 2 and 3). Note also that the input-universal nature of DECs implies that \( f.v \in \text{en}_{C}(q) \) for all \( f \in \text{en}_{A}(s) \), \( v \in \theta_{C}(f) \). Therefore, this relation encodes an input and output duality that the DEC at state \( q \) allows more input events but produces less output events than the IA at state \( s \). Clearly, the most abstract implementation of an IA conforms to the IA (with respect to arbitrary \( \theta \)).

### 4.4 Practical Conformance Checking

To check the conformance of a DEC to a given IA, instead of building the Cartesian product of their states as proposed by de Alfaro and Henzinger (2001), we employ a two-step method which has significant benefits for tractability. First of all, we calculate the local state space of the DEC utilising the context assumptions of the IA, and then determine the conformance by checking the state space for the absence of error states.

The local state space of the DEC is defined as the synchronised product of a closed RTS network of two components – one component is the DEC to be checked, while the other component is the most abstract implementation of the mirror of the IA. The second component thus represents the minimally helpful environment of the first which still provides all expected inputs to the first. Formally, it is defined as follows.

**Definition 15.** Consider a DEC \( C \) and an IA \( A \) such that \( \Sigma_{A}^{f} \subseteq \alpha_{C}^{f} \) and \( \Sigma_{A}^{g} \supseteq \alpha_{C}^{g} \). Let \( \theta = \theta_{C} \cup \{(f, \emptyset) \mid f \in \Sigma_{A} \setminus \alpha_{C}^{f} \} \), \( J \) be the MAI of the mirror of \( A \) with respect to \( \theta \) (Definition 13), \( W = \{C, J\} \), \( R = \{(f.v, f.v) \mid f.v \in \Sigma_{A}^{f} \cup \Sigma_{A}^{g}\} \), and \( N = \{W, R\} \) be a derived closed RTS network. Then the synchronised product of \( N \) is called the local state space of \( C \) with respect to \( A \).

The following theorem presents the checking method and gives both the sufficient and necessary conditions for the conformance.

**Theorem 2.** Let \( C, A, \theta, J \) be as in Definition 15 and \( L_{\otimes} = \{s_{\otimes}^{0}, s_{\otimes}, \Sigma_{\otimes}, \Delta_{\otimes}\} \) be the local state space of \( C \) with respect to \( A \). Then \( C \) conforms to \( A \) if and only if \( \forall s \in S_{\otimes}, \pi_{J}(s) \neq \bot \).

On the basis of the theorem, we can now be sure that Figure 1 conforms to Figure 2 and can be placed into a system design as an implementation of the latter. If we restrict ourselves to a finite set of transmitted values, the conformance can be automatically checked using our Moses tool (Esser and Janneck, 2001).

### 5. MODULAR ANALYSIS OF DEC NETWORKS

A typical component-based design process combines top-down and bottom-up design. IAs are obtained during system decomposition, together with the synchronisation patterns between them. These IAs are then used for developing or selecting suitable DECs (or for further decomposition). These obtained DECs are next composed to form a concrete component-based design, viz. a closed DEC network, where the synchronisation patterns of IAs are reused as the interconnections between DECs. Hence the IAs capture the interaction protocols expected by the designer of the DECs.

In this section, we lay the foundation for this approach by defining networks of both DECs and IAs and by giving consistency properties for both closed and open DEC networks. Modular
approaches to verifying these networks are presented next, which utilise the more abstract IA networks to alleviate the state space explosion. In addition, a modular method for checking the conformance of open DEC networks with IAs is also presented, which maximizes the benefit of the above approaches for hierarchical component-based designs.

5.1 DEC Networks

Similar to RTS networks, DEC networks are composed of components interacting by means of synchronisation vectors. Differently, designing these networks can be much simpler as these vectors can be specified by the interconnection between input/output ports of the component DECs without worrying about the individual values. In defining the networks, we still assume a special symbol $\varepsilon$ which is not a port of any component DEC.

**Definition 16.** A DEC network is defined by $D = (\alpha, \theta, P, \gamma)$, where

- $\alpha$ is a finite set of external ports of the network, consisting of two disjoint sets of input ports $\alpha^I$ and output ports $\alpha^O$. We let $\alpha^F = \alpha \cup \{\varepsilon\}$;
- $\theta : \alpha \rightarrow 2^V$ is a total typing function, mapping each port to a set of values;
- $P$ is a finite set of DECs. We let $\alpha^F_p = \alpha_p \cup \{\varepsilon\}$ for all $p \in P$;
- $\gamma \subseteq \alpha^F \times \prod_{p \in P} \alpha^F_p$ is a relation indexed by $\alpha^I \cup \bigcup_{p \in P} \alpha^O_p$.

Similar to RTS networks, we call a DEC network *closed* if $\alpha = \emptyset$, or *open* otherwise. We write $D = (P, \gamma)$ for a closed network $D$. Also, we call $\gamma$ a set of interconnections. In addition, we redefine $\rho$, $\eta$ and $\varphi$ for an interconnection $f \in \gamma$ in order to formulate the well-formedness rule for DEC networks.

**Definition 17.** Consider a DEC network $D$ and an interconnection $f \in \gamma$. Let $\alpha_{env} = \alpha$, $\alpha_{env}^I = \alpha^I_D$, $\alpha_{env}^O = \alpha^I_D$, $P' = \{\text{env}\} \cup P$, $l \in P'$, and projections $\pi_l : \gamma \rightarrow \alpha_l$ for all $l$. Then the producer of $f$, denoted by $\rho_f$, is the unique DEC $l$ such that $\pi_l(f) \in \alpha_l^I : \pi_l(f) = \alpha_l^I$. The producer of $f$. Also, the set of consumers of $f$, denoted by $\eta_f$, consists of all DECs $l$ such that $\pi_l(f) \in \alpha_l^I$. For any consumer $l$, $\pi_l(f)$ is called a consumer port of $f$. In addition, an idler of $f$ is a DEC $l$ such that $\pi_l(f) = \varepsilon$. The set of idlers is denoted by $\varphi_f$.

**Definition 18.** A DEC network $D$ is called well-formed if $\theta_{\rho_f} (\pi_{\rho_f}(f)) \leq \theta_l (\pi_l(f))$ holds for all $f \in \gamma$, $l \in \eta_f$.

The well-formedness rule requires that the values that can be possibly transmitted are all valid (or meaningful) to all the receiving DECs. In other words, only ports of matching kinds can be connected in a DEC network. This thus ensures data type compatibility between ports. In the following, we shall only consider well-formed DEC networks.

An example DEC network is visualized as a process network in Figure 5, where rectangles denote DECs and triangles denote DEC ports. This network is closed with no external ports declared. The DEC set $P$ contains the DECs in Figures 1, 3 and 4, where “*” matches any unspecified input events. The interconnections $\gamma$ are depicted by arcs, where $\varepsilon$ is interpreted as the non-involvement of a DEC in an arc. We know that the adder calculates the sum of two given parameters. The doubler calculates the double of a given number, utilising the function provided by
the adder. When receiving a number from port “d”, it feeds the adder with the same number at ports “a” and “b”, gets the sum at port “c”, and then reports back the result via port “e”. The user simply uses the service provided by the doubler to calculate the doubles of randomly chosen integers. As such, we have omitted the minutiae for error reporting or recovery. Next, we give the semantics of DEC networks in terms of RTS networks.

![A doubler DEC model](image)

**Figure 3: A doubler DEC model**

![A user DEC model](image)

**Figure 4: A user DEC model**

![A DEC network](image)

**Figure 5: A DEC network**

### Definition 19
Consider a DEC network $D = (\alpha, \theta, P, \gamma)$. Let $P^\prime = \{\text{env}\} \cup P$ and $N = (\Sigma, P, R)$ be a derived RTS network such that

- $\Sigma = \Sigma^I \cup \Sigma^O$, $\Sigma^I = \{(f, v) | f \in \alpha^I, v \in \theta(f)\}$ and $\Sigma^O \subseteq \{(f, v) | f \in \alpha^O, v \in \theta(f)\}$;
- $\alpha_{\text{env}} = \alpha$, $\alpha_{\text{env}}^I = \alpha_{\text{env}}^O = \alpha$, $\Sigma_{\text{env}} = \Sigma$, $\Sigma_{\text{env}}^I = \Sigma^O$, $\Sigma_{\text{env}}^O = \Sigma^I$ and $\theta_{\text{env}} = \theta$;
- $\Sigma^g = \Sigma \cup \{\epsilon\}$, $\Sigma_{\text{env}}^g = \Sigma^g$, and for all $p \in P$, $\Sigma_p^g = \Sigma_p^{\text{obs}} \cup \{\epsilon\}$;
- polymorphic projections $\pi_1$ with $\pi_1 : \gamma \rightarrow \alpha_1$ and $\pi_1 : (\Sigma^g \times \Pi P \Sigma_p^g) \rightarrow \Sigma_1^g$ for $l \in P^\prime$;
- $R = \{r \in \Sigma^g \times \Pi \text{p} P \Sigma_p^g | \exists \epsilon \in \gamma, v \in \theta_{\text{pt}}(\pi_{\text{pt}}(f)), \forall l \in P^\prime, (\pi_1(f) = \epsilon \land \pi_1(r) = \epsilon) \lor (\pi_1(f) \neq \epsilon \land \pi_1(r) = \pi_1(f).v)\}$.

Then the synchronised product of $D$ is defined by the synchronised product of $N$.

Note in the definition of $R$ that the data being communicated, viz. $v$, ranges over the value set defined for the producer port. Also, it is left unchanged during the transmission from the producer DEC to the consumer DECs. In other words, a DEC network only relays or broadcasts the data but never modifies it. Note also that if all component DECs do not contain the error state $\perp$, then the synchronised product of the network does not contain $\perp$ in its state space either, since DECs are input-universal. In addition, immediately from Definitions 9, 19 and Proposition 1, we can have the following proposition.

### Proposition 3
Given a DEC network $D = (\alpha, \theta, P, \gamma)$. Let $L = (s^0, S, \Sigma, \Delta)$ be the synchronised product of $D$. Then $(\alpha, \theta, s^0, S, \Sigma, \Delta)$ defines a DEC (called the composite DEC of $D$).

Usually, there are two possible ways to build a DEC network. The first is to include all DECs in a flat network and define the interconnections between them. The second is to assemble some of the DECs and the interconnections between them into an open DEC network. The latter can then be used as a component for building a larger network. The interpretation of DEC networks in terms of...
DECs in Proposition 3 enable such a hierarchical design. In other words, a DEC network can be composed of components which are in turn DEC networks. Semantically, a hierarchical network is identical to a flattened network which combines all the components and interconnections of the constituent networks, as defined below.

**Definition 20.** Consider a DEC network \( D_1 = (\alpha_1, \theta_1, P_1, \gamma_1) \) and an open DEC network \( D_2 = (\alpha_2, \theta_2, P_2, \gamma_2) \) such that \( D_2 \in P_1 \), the flattened network \( D \) of \( D_1 \) by \( D_2 \) is defined by \( D = (\alpha_1, \theta_1, P, \gamma) \), where \( P' = P_1 \setminus \{ D_2 \} \), \( P = P_1 \cup P_2 \), \( F = \alpha_1^P \times \bigwedge_{p \in P} \alpha_2^p \), and

\[
\gamma = \{ \mathbf{f} \in F \mid \exists \mathbf{f}_1 \in \gamma_1, \mathbf{f}_2 \in \gamma_2, \pi_{D_1}(\mathbf{f}_1) = \pi_{env}(\mathbf{f}_2) \land \pi_{D_2}(\mathbf{f}_1) \neq \epsilon \land \forall p \in P_1', \pi_p(\mathbf{f}) = \pi_p(\mathbf{f}_1) \land \forall p \in P_2, \pi_p(\mathbf{f}) = \pi_p(\mathbf{f}_2) \}
\]

(5)

\[
\cup \{ \mathbf{f} \in F \mid \exists \mathbf{f}_1 \in \gamma_1, \pi_{D_2}(\mathbf{f}_1) = \epsilon \land \forall p \in P_1', \pi_p(\mathbf{f}) = \pi_p(\mathbf{f}_1) \land \forall p \in P_2, \pi_p(\mathbf{f}) = \epsilon \}
\]

(6)

\[
\cup \{ \mathbf{f} \in F \mid \exists \mathbf{f}_2 \in \gamma_2, \pi_{env}(\mathbf{f}_2) = \epsilon \land \forall p \in P_1', \pi_p(\mathbf{f}) = \epsilon \land \forall p \in P_2, \pi_p(\mathbf{f}) = \pi_p(\mathbf{f}_2) \}
\]

(7)

In the definition, the components of both \( D_1 \) and \( D_2 \) become components of \( D \) and the interconnections of both \( D_1 \) and \( D_2 \) are unified into the interconnections of \( D \). The unification is based on shared ports of \( D_2 \). For example, suppose \( \mathbf{f}_1 \) is an interconnection in \( D_1 \) and \( \mathbf{f}_2 \) is in \( D_2 \) such that \( \pi_{D_1}(\mathbf{f}_1) = \pi_{D_2}(\mathbf{f}_1) \) and \( \pi_{env}(\mathbf{f}_1) = \pi_{env}(\mathbf{f}_2) \). We unify \( \mathbf{f}_1 \) and \( \mathbf{f}_2 \) into one interconnection in \( D \) such that both \( \pi_{D_2}(\mathbf{f}_1) \) and \( \pi_{env}(\mathbf{f}_2) \) are removed (Formula 5). On the other hand, interconnections in both \( D_1 \) and \( D_2 \) which do not involve ports of \( D_2 \) are expanded and retained in \( D \) (Formulae 6–7). Flattening a hierarchical design recursively, we can ultimately get a DEC network with no networks as subcomponents.

### 5.2 IA Networks

In the section, we introduce the concepts of interface automaton networks and define the consistency for them, which, roughly speaking, refers to the compatibility of constituent IAs in the networks.

**Definition 21.** An IA network is a closed RTS network \( N = (W, R) \), where \( W \) is a finite set of IAs.

**Definition 22.** Consider an IA network \( N = (W, R) \). Let \( L_N = (s^0, S, \Sigma, \Delta) \) be the synchronised product of \( N \). Then \( N \) is consistent if \( \pi_a(s) \neq \perp \) for all \( s \in S, a \in W \).

An IA network is a restricted form of RTS networks. Its consistency ensures that the network is free from unspecified reception. More specifically, when a synchronisation occurs as a result of an output produced by a constituent IA, for every consumer IA of the corresponding synchronisation vector, the consumed event must be specified at its current state. In other words, for every consumer, there always exists a state to enter in its state space after consuming the event. That is to say, the error state is not reachable by the input-universal version of any constituent IA in the synchronised product of the IA network.

### 5.3 Verification of Closed DEC Networks

#### 5.3.1. Consistency Property

The IAs and IA networks of Section 5.2 capture the interaction protocols expected by the designer of the DECs. The consistency of a closed DEC network is defined in terms of the freedom from unexpected reception. In other words, every possible trace (or event sequence) of a DEC in the network corresponds to a trace of its associated IA, disregarding the internal events and data values accompanying the input/output events of the DEC.
To present a formal definition of the consistency property, we need to formally specify the association between the specification IAs and the implementation DECs as well as the trace projection from the behaviour of the DEC network and that of the IAs. The former is specified in Definition 23 as a sketching function $\beta$, which maps each component DEC to an IA that has been used to guide the development or selection of the DEC. The latter is defined by Definition 24.

**Definition 23.** A closed DEC network $D = (P, \gamma)$ is said to be sketched by a total function $\beta: P \rightarrow UEFA$ if for all $(p, a) \in \beta$, $p$ conforms to $a$.

Deriving from the trace projection of DECs (cf. Definition 10), we obtain a mapping from the behaviour of the network to that of the IAs associated with the DECs, as defined below.

**Definition 24.** Consider a closed DEC network $D$ sketched by $\beta$. Let $\sigma$ be a trace of $D$ from $s_D^0$, $(p, a) \in \beta$, $\pi_p(\sigma) |_{\Sigma_p^{obs}}$ be the observable event sequence of the trace projection of $\sigma$ on $p$ (Definitions 2, 10). Then the trace projection $\pi_p(\sigma) |_{\Sigma_a}$ of $\sigma$ on $a$ is the sequence of ports involved in $\pi_p(\sigma) |_{\Sigma_p^{obs}}$.

We can now define the freedom of unexpected reception for traces and thereafter the consistency of closed DEC networks.

**Definition 25.** Let $D$, $\beta$, $\sigma$ be as in Definition 24. Then a trace $\sigma$ is free from unexpected reception (with respect to $\beta$) if $\pi_p(\sigma) |_{\Sigma_a}$ is a trace of a from $s_a^0$ for all $(p, a) \in \beta$. $D$ is called consistent (with respect to $\beta$) if all traces of $D$ from $s_D^0$ are free from unexpected reception.

In other words, a closed DEC network is consistent if the environment of every DEC is always helpful and never provides input events of an unexpected kind to the DEC. The environment of a DEC (in the network) refers to the open DEC network composed of all the other DECs. Also, given an IA associated with the DEC, we let an IA event refused at a state of the IA represent a kind (or type) of input events unexpected at a corresponding state of the DEC. Often receiving such an event will force the DEC to execute some error report or recovery tasks and thus prevent it from providing the normal functionality to the system.

From the definition, we can prove an important property for traces free from unexpected reception, given by the following proposition.

**Proposition 4.** Consider a closed DEC network $D$ sketched by $\beta$ and a trace $\sigma$ of $D$ from $s_D^0$, which is free from unexpected reception with respect to $\beta$. Let $q \in S_D$ be a state reachable via $\sigma$, $(p, a) \in \beta$, and $s_a \in S_a$ be a state reachable via $\pi_p(\sigma) |_{\Sigma_a}$ from $s_a^0$. Then $s_a$ is the unique state reachable in $a$ via $\pi_p(\sigma) |_{\Sigma_a}$ and $\pi_p(q) \leq s_a$.

### 5.3.2 Verifying Consistency

In order to alleviate the state space explosion problem, a DEC network is not analysed directly in this approach. Instead, we utilise the interface automata, the interaction protocol specifications of the DECs, to build an IA network. In the IA network, the IA events are related in the same way as the component ports are interconnected in the DEC network. The following theorem shows that, to verify the consistency of a closed DEC network, it is sufficient to prove both the conformance of every DEC with its corresponding IA and the consistency of the IA network.
Theorem 5. Consider a closed DEC network $D = (P, \gamma)$ sketched by $\beta$. Let $N = (\text{image}(\beta), \gamma)$ be a derived IA network. Then $D$ is consistent with respect to $\beta$ if $N$ is consistent.

Suppose IAs in Figures 2, 6 and 7 have guided us to select DECs in Figures 1, 3 and 4, respectively, to compose the DEC network in Figure 5. To prove the consistency of this network, we can independently prove the conformance of the DECs to their corresponding IAs and the consistency of an IA network consisting of the three IAs. It is not hard to construct the synchronised product and prove the consistency of the IA network. The product is a RTS shown in Figure 8, where all events are internal and no error state exists.

Since IAs capture the interface behaviour of DECs, the IA network describes a superset of the interaction patterns between DECs, which involves neither data values nor internal computations of DECs. As a consequence, IA networks generally have smaller state spaces than DEC networks. Hence it is much cheaper to determine the consistency of IA networks. Further, because conformance checking involves only one single DEC at a time, the state space that needs to be handled is also much smaller. Therefore, using this divide-and-conquer approach, the potential state space explosion problem for this verification task can be alleviated.

5.4 Verification of Open DEC Networks

In order to enable the modular analysis at each level of the hierarchy, we present modular methods for analysing open DEC networks in this section. The properties under consideration include both consistency and conformance.

5.4.1 Consistency Property

To define the consistency for open DEC networks, we let “env” be a syntactical term referring to the environment of a network and associate env with an IA in defining the sketching function. The IA captures the interaction protocol expected of any DEC (or DEC network) later acting as the environment of the network.

**Definition 26.** An open DEC network $D = (\alpha, \theta, P, \gamma)$ is said to be sketched by a total function $\beta: (P \cup \text{env}) \rightarrow U^a$ if (1) for $(\text{env}, a) \in \beta$, $\Sigma_a^\alpha \subseteq \alpha^D$ and $\Sigma^I_a \supseteq \alpha^D$; and (2) for all $(p, a) \in \beta$ such that $p \neq \text{env}$, $p$ conforms to $a$.

The consistency of an open DEC network is then defined in terms of the consistency of a derived closed network which contains an additional DEC that can be arbitrary but still conforms to the IA associated with env.

**Definition 27.** Consider an open DEC network $D = (\alpha, \theta, P, \gamma)$ sketched by $\beta$. $D$ is called consistent (with respect to $\beta$) if for any DEC $C \notin P$ such that $C$ conforms to $\beta(\text{env})$, the closed DEC network $D' = \{P', \gamma\}$ is consistent with respect to $\beta'$, where $P' = P \cup \{C\}$ and $\beta' = \{(C, \beta(\text{env}))\} \cup \{(p, a) \in \beta \mid p \neq \text{env}\}$. 
The following proposition shows that the consistency of hierarchical DEC networks, where open networks are composed, can be determined by checking the consistency at each level of the hierarchy. This thus provides a foundation for component-based design in that an IA and its mirror can be used as a pair to guide the independent development of components and the composition.

**Proposition 6.** Consider a DEC network \( D_1 = (\alpha_1, \theta_1, P_1, \gamma_1) \) and an open DEC network \( D_2 = (\alpha_2, \theta_2, P_2, \gamma_2) \) such that \( D_2 \subseteq P_1 \), \( D_1 \) and \( D_2 \) are consistent with respect to \( \beta_1 \) and \( \beta_2 \), respectively, and \( \beta_2(env) \) is the mirror of \( \beta_1(D_2) \). The flattened network \( D \) of \( D_1 \) by \( D_2 \) is consistent with respect to \( \beta \), where \( \beta = \{(p, a) \in \beta_1 \mid p \neq D_2\} \cup \{(p, a) \in \beta_2 \mid p \neq env\} \).

5.4.2 Verifying Consistency

Similar to the consistency checking of closed networks, the consistency of open DEC networks can also be determined by the consistency of derived IA networks. These IA networks consist of the IAs associated with both env and the components, and share the same interconnections with the open DEC networks except that each component is substituted by its corresponding IA and env is by \( \beta(env) \).

**Theorem 7.** Consider an open DEC network \( D = (\alpha, \theta, P, \gamma) \) sketched by \( \beta \). Let \( N = (image(\beta), \gamma) \) be a derived IA network. Then \( D \) is consistent with respect to \( \beta \) if \( N \) is consistent.

Suppose we have an open DEC network consisting of the adder in Figure 1 and the doubler in Figure 3, and a function \( \beta \) mapping env to Figure 7 and the two DECs to the IAs in Figures 2 and 6. Then this theorem allows us to check the consistency of the open network, even though the user DEC is unknown.

5.4.3 Verifying Conformance

As stated previously, an open DEC network forms a DEC. To check the conformance of such a network to an IA, we can certainly use the checking method proposed in Section 4.4. However, this may lead to the state space explosion. Instead, we demonstrate using the following theorem that the conformance can be deduced from the derived IA network as well.

**Theorem 8.** Let \( D, \beta, N \) be as in Theorem 7 and \( A \) be the mirror of \( \beta(env) \), then \( D \) conforms to \( A \) if \( N \) is consistent.

Applying the theorem into the open DEC network consisting of the doubler and the adder, we know this network conforms to the mirror of Figure 7.

6. CONCLUSION

We have presented a practical method for verifying the consistency of component-based designs. This employs a divide-and-conquer approach where each components is individually tested for conformance with its interaction protocol captured by an interface automaton, and the network of interface automata (which matches the network of components) is checked for consistency. This divide-and-conquer approach, together with abstraction from the data values transmitted between components, can lead to a significant reduction in the state space to be explored. Furthermore, it was shown that the consistency analysis can be carried out at each level of the hierarchy. This further minimises the state space that needs to be built in each analysis task.

We have provided only a sufficient condition for determining the consistency of DEC networks, which refers to the consistency of IA networks. This will be appropriate for many practical
situations since the consistency of IA networks will be ensured when synchronisation patterns between components are designed during system decomposition, and thus before the development or selection of DECs is carried out. In some cases, however, the use of pre-existing DECs and their abstraction into IAs may mean that the proposed verification process will produce false negatives, i.e. the above technique may incorrectly report inconsistency. In this case, a process of progressive refinement will need to be adopted, as advocated elsewhere (Clarke et al., 2000).

The above technique can also be extended to the analysis of safety properties which require more detail than can be derived merely from the IA network. To check these properties, we can make use of the component local state spaces to deduce a superset of the state combinations of relevant DECs with the help of the derived IA network, and check this set for the violation of the properties. In this way, the preservation of such properties can be ensured. Preliminary investigations of this technique are reported in (Jin et al., 2003c).

The techniques presented in this paper have been implemented in the Moses tool, including the extensions for more general safety properties. The implementation has been applied by the authors in (Jin et al., 2003c) to the production cell case study (Lewerentz and Lindner, 1995). Approximately three orders of magnitude improvement in the size of the state space were achieved. The implementation of the extended safety property checking meant that the important safety requirements posed in an earlier paper (Heiner and Deussen, 1995) could be verified.

REFERENCES


Modular Consistency Analysis of Component-Based Designs


7. PROOFS

Proof of Theorem 2

Sufficiency. Let a relation $\phi = \{ (q, s) \in S_\Sigma \mid q \in S_C, s \in S_A \}$, then we prove $\phi$ is an alternating simulation relation between $C$ and $A$ by induction. First, $(s^0_C, s^0_A) \in \phi$. Next, suppose $(q, s) \in \phi$, then

- For $e \in \Sigma^H$, if $\exists q \xrightarrow{e} C q'$, then $(q', s) \in S_\Sigma$ and $(q', s) \in \phi$;

- For $f \cdot v \in \Sigma^C$, we know $f \in \Sigma^A$, $v \in \theta(f)$ and thus $f \cdot v \in \Sigma^J_I$. If $\exists q \xrightarrow{f \cdot v} C q'$, then $\exists s' \in S_J, (q, s) \xrightarrow{f \cdot v} (q', s')$ (because $J$ is input-universal). Since $(q', s') \in S_\Sigma$, from the condition of the theorem, we have $s' \neq \bot$. Hence $s' \in S_A$ and $(q', s') \in \phi$;

- For $f \in \Sigma^I_A$, we know $f \in C$. If $\exists s' \xrightarrow{f} A s'$, we have $\forall v \in \theta_C(f), f \cdot v \in \Sigma^A \land s \xrightarrow{f \cdot v} J s'$.

Since $f \cdot v \in \Sigma^C$ and $C$ is input-universal, $\exists q' \in S_C, (q, s) \xrightarrow{f \cdot v} (q', s')$. Hence $(q', s') \in \phi$;

Therefore, $\phi$ is an alternating simulation relation and $C$ conforms to $A$.

Necessity. Let $s \leq$ be an alternating simulation relation between $C$ and $A$, $\sigma$ be a trace of $\Sigma^C$ from $s^0_C$, and $(q, s) \in S_\Sigma$ be a state reachable via $\sigma$. Then we prove $s \neq \bot$ and $q \leq s$ by induction over the length of $\sigma$. First, when $\sigma = \lambda$, we know $(q, s) = (s^0_C, s^0_A) = s^0_C$, Hence $s \neq \bot \land q \leq s$. Next, suppose $s \neq \bot \land q \leq s$ holds for any $\sigma$. Since $S_J = S_A \cup \{ \bot \}$, we know $s \in S_A$.

- For $e \in \Sigma^H$, if $\exists (q, s) \xrightarrow{e} (q', s')$, then $s' = s$ (thus $s' \neq \bot$) and $q \xrightarrow{e} C q'$. Since $q \leq s$, we can get $q' \leq s$ (Def. 14, Cond. 1).

- For $f \cdot v \in \Sigma^C$, if $\exists (q, s) \xrightarrow{f \cdot v} (q', s')$, then $q \xrightarrow{f \cdot v} C q'$. Since $q \leq s$ and $A$ is deterministic, $s' \in S_A$, $s \xrightarrow{f} A s'$ and $q' \leq s'$ (Def. 14, Cond. 2). Thus $s' \neq \bot$.

- For $f \cdot v \in \Sigma^C$, if $\exists (q, s) \xrightarrow{f \cdot v} (q', s')$, then $s \xrightarrow{f \cdot v} J s'$. Since $f \in \Sigma^A$, $s \xrightarrow{f} A s'$ and $s' \neq \bot$.

From Def. 14 (Cond. 3), we know $q' \leq s'$.

Therefore, that $C$ conforms to $A$ implies $\forall (q, s) \in S_\Sigma, s \neq \bot$.

Proof of Proposition 4

Proof. Let $\xi_{p} = \pi_p(\sigma) |_{\Sigma^p}$ and $\xi_{\alpha} = \pi_p(\sigma) |_{\Sigma^\alpha}$ for $(p, \alpha) \in \beta$. Then $s_{\alpha}$ is the only state reachable via $\xi_{\alpha}$ in $a$ because $\Sigma^H_a = \emptyset$ and $a$ is deterministic. In addition, we know $|\xi_p| = |\xi_{\alpha}|$ and that every event $f$ in $\xi_{\alpha}$ corresponds an event $f \cdot v$ in $\xi_p$ from Def. 24. Therefore, it follows by induction that $\pi_p(q) \leq s_{\alpha}$.

Modular Consistency Analysis of Component-Based Designs

Proof of Theorem 5

Proof. We prove this theorem by induction over the length of a trace $\sigma$ of $D$ from $s^0_D$. Let $W = \text{image}(\beta)$, $q \in S_D$ be a state reachable via $\sigma$, and $\xi_a = \pi_p(\sigma) \mid_{\Sigma_a}$ be the trace projection of $\sigma$ on $a$ for $(p, a) \in \beta$ (cf. Def. 24). Then at each step of the induction, we prove that (i) $\sigma$ is free from unexpected reception; (ii) $\exists s \in S_N$ such that $\forall a \in W, \pi_a(s)$ is the reachable state in $a$ via $\xi_a$.

Firstly, when $\sigma = \lambda$, we know $q = s^0_D$. Clearly, (i) holds. Then $s = s^0_N$ satisfies (ii). Next, suppose (i)–(ii) hold on any trace $\sigma$. For any step $q \xrightarrow{e} D q'$, let a trace $\sigma' = \sigma \cdot e$, then we shall prove (i)–(ii) hold on $\sigma'$. Let $s \in S_N$ be the state satisfying (ii) for $\sigma$ and $\xi_a'$ be defined over $\sigma'$. Then from Prop. 4 we can have $\forall (p, a) \in \beta, \pi_p(q) \preceq \pi_a(s)$.

- if $\exists p \in P$ such that $e \in \Sigma_p^H$, then $\xi_a' = \xi_a$ for all $a \in W$ and thus (i) holds. Clearly, $s$ is the state for which (ii) holds on $\sigma'$.

- if $\exists p \in P$ such that $e \in \Sigma_p^P$, we know $\pi_p(q) \xrightarrow{e \beta} \pi_p(q')$. Let $e = f \cdot v$, then $\exists s' \in S_a$, $\pi_a(s) \xrightarrow{f \cdot v} s' \wedge \pi_p(q') \preceq s'$ (because $\pi_p(q) \preceq \pi_a(s)$). Thus $\xi_a' = \xi_a \cdot f$ is a trace of $a$.

  - For any DEC $g \in P \setminus \{p\}$ such that $\exists f \in \gamma, \pi_g(f) \in \alpha_g^I$, let $f' = \pi_g(f)$, then we have $\pi_g(q) \xrightarrow{f', u} \pi_g(q')$. Let $b = \beta(g)$, then since $N$ is consistent, no error state exists in $L_N$ and thus $f' \in \text{en}_b^I(\pi_b(s))$. Since $\pi_g(q) \preceq \pi_b(s)$, we know $\exists s' \in S_b$ such that $\pi_b(s) \xrightarrow{f'} s' \wedge \pi_g(q') \preceq s'$. Hence $\xi_b' = \xi_b \cdot f'$ is a trace of $b$.

  - For any other DEC $g \neq p$, let $b = \beta(g)$ and $s' = \pi_b(s)$, then we know $\xi_b' = \xi_b$ is a trace of $b$.

To sum up, $\sigma'$ is free from unexpected reception and $s_a'$ is reachable via $\xi_a'$ in $a$ for all $a \in W$. Let $s'$ be the state such that $\pi_a(s') = s'_a$, then $s' \in S_N$ and $s'$ satisfies (ii).

Therefore, the theorem holds.

Proof of Proposition 6

Proof. Let $C, D'$ and $\beta'$ be as in Def. 27, then we need to prove that $D'$ is consistent with respect to $\beta'$. Clearly, $D'$ is sketched by $\beta'$. We then prove by induction that given a trace $\sigma$ of $D'$ from $s^0_{D'}$, for all $(p, a) \in \beta'$, $\xi_a = \pi_p(\sigma) \mid_{\Sigma_a}$ is always a trace of $a$. We denote this condition as (i).

First, when $\sigma = \lambda$, (i) clearly holds. Next, suppose (i) holds on any trace $\sigma$. Let $A = \beta_1(D_2), M = \beta_2(\text{env})$, and $s_a \in S_a$ be the state reachable via $\xi_a$ for all $a \in \text{image}(\beta) \cup \{A\}$. For any step $q \xrightarrow{e} D q'$, let a trace $\sigma' = \sigma \cdot e$, then we shall prove (i) holds on $\sigma'$.

- If $\exists p \in P \cup \{C\}$ such that $e \in \Sigma_p^H$, then (i) holds on $\sigma'$ for all $(p, a) \in \beta'$;

- If $\exists p \in P \cup \{C\} \setminus \{D_2\}$ such that $e \in \Sigma_p^P$, then (i) holds on $\sigma'$ for all $(p, a) \in \beta' \setminus \beta_2$, since $D_1$ is consistent. Let $e = f \cdot v$, $f \in \gamma_1$ (such that $\pi_p(f) = f$) and $f' = \pi_{D_2}(f)$, then

  - If $f' = e$, then (i) holds on $\sigma'$ for all $(p, a) \in \beta_2$ such that $p \neq \text{env}$;

  - If $f' \in \Sigma_A$, since $D_1$ is consistent, we know $f' \in \text{en}_A(s_A)$ and thus $f' \in \text{en}_M(s_A)$. Also, because $D_2$ is consistent with respect to $f' \cdot v$, (i) holds on $\sigma'$ for all $(p, a) \in \beta_2$ such that $p \neq \text{env}$.
Modular Consistency Analysis of Component-Based Designs

- If \( \exists p \in P_2 \) such that \( e \in \Sigma^O_p \), then (i) holds on \( \sigma' \) for all \( (p, a) \in \beta_2 \) such that \( p \neq \text{env} \), since \( D_2 \) is consistent. Also, we know \( \exists f \cdot v \in \Sigma^c_{D_2} \) such that \( (\Pi_{p \in P_2} \pi_p(q), f \cdot v, \Pi_{p \in P_2} \pi_q(q')) \in \Delta_{D_2} \). Because \( D_1 \) is consistent and \( D_2 \in P_1 \), (i) holds on \( \sigma' \) for all \( (p, a) \in \beta' \setminus \beta_2 \).

Therefore, the proposition holds.

Proof of Theorem 7

Proof. Let \( C, D' \) and \( \beta' \) be as in Def. 28. Then \( D' \) is clearly a closed DEC network sketched by \( \beta' \). Because \( N \) is consistent, we know \( D' \) is consistent due to Thm. 5. Therefore, \( D \) is consistent with respect to \( \beta \).

Proof of Theorem 8

Proof. We know from Thm. 7 that \( D \) is consistent with respect to \( \beta \). Let \( M = \beta(\text{env}) \) and \( \phi = \{ (q, \pi_M(s)) | q \in S_D, s \in S_N, \forall p \in P, \pi_p(q) \leq \pi_{\beta(p)}(s) \} \). Then we prove by induction that \( \phi \) is an alternating simulation relation between \( D \) and \( A \) (Note \( S_M = S_A \)). Firstly, \( (s^0_D, s^0_A) \in \phi \) because \( \pi_M(s^0_N) = s^0_M = s^0_A \). Next, suppose \( \exists s \in S_N \) such that \( (q, \pi_M(s)) \in \phi \).

- For \( e \in \Sigma^H_D \), if \( \exists q \xrightarrow{\xi_D} q' \), we prove \( (q', \pi_M(s)) \in \phi \). If \( \exists p \in P \) such that \( e \in \Sigma^H_p \), then \( (q', \pi_M(s)) \in \phi \). Otherwise, \( \exists p \in P \) such that \( e \in \Sigma^O_p \) and let \( e = f \cdot v \), then \( \exists f \in \gamma, \pi_{\text{env}}(f) = e \wedge \pi_p(f) = f \). Let \( a = \beta(p) \), then since \( p \leq a \), we know \( f \in \text{en}_a^O(\pi_a(s)) \). Because \( N \) and \( D \) are consistent, \( \exists s' \in S_N, s \xrightarrow{f} s' \wedge \pi_M(s) = \pi_M(s') \wedge \forall g \in P, \pi_g(q') \leq \pi_{\beta(g)}(s') \). Hence \( (q', \pi_M(s)) \in \phi \).

- For \( f \cdot v \in \Sigma^O_D \), if \( \exists q \xrightarrow{f \cdot v} q' \), then \( \exists p \in P, f \in \gamma, \pi_p(f) \in \alpha^O_p \wedge \pi_{\text{env}}(f) = f \). Let \( a = \beta(p) \) and \( f' = \pi_p(f) \), then since \( p \leq a \), we can get \( f' \in \text{en}_a^O(\pi_a(s)) \) and \( \exists s' \in S_N, s \xrightarrow{f'} s' \wedge \pi_M(s') \neq \bot \wedge \forall g \in P, \pi_g(q') \leq \pi_{\beta(g)}(s') \). Hence, \( (q', \pi_M(s')) \in \phi \).

- For \( f \in \Sigma^T_A \), if \( \exists \pi_M(s) \xrightarrow{f} s' \) then \( f \in \text{en}_M^T(s) \) and \( \pi_M(s) \xrightarrow{f} s' \). Since \( N \) is consistent, \( \exists s' \in S_N, \pi_M(s') = s' \wedge s \xrightarrow{f} s' \). Also, since \( D \) is input-universal, \( \forall v \in \theta(f), q \xrightarrow{f \cdot v} q' \). It is easy to prove that \( \forall p \in P, \pi_p(q') \leq \pi_{\beta(p)}(s') \), since \( D \) is consistent. Hence \( (q', \pi_M(s')) \in \phi \).

Hence, \( \phi \) is an alternating simulation between \( D \) and \( A \). From Def. 14, \( D \) conforms to \( A \).

BIOGRAPHICAL NOTES

Yan Jin is a PhD Student in the School of Computer Science at the University of Adelaide. His research focuses primarily on the formal specification and compositional verification of component-based systems, especially heterogeneous systems. In addition, he is actively researching on the formal semantics definition of visual modelling languages such as Petri nets and UML statecharts. Previously, Yan Jin graduated from Chongqing University, China, with a BEng in 1994 and a MEng in 1997, both in Computer Science, and worked as a network engineer for three years in the Shenzhen branch of the China Telecom Corporation.
Charles Lakos graduated from the University of Sydney with a BSc (Hons) in 1972 and a PhD in 1979, both in Computer Science. Charles has been interested in concurrency research since the late 1980s. His work has focussed primarily on the augmentation of the Petri Net formalism with object-oriented extensions, thus providing powerful structuring capabilities. Object-orientation has thus been an interest in both teaching and research for over 15 years. Recent work has considered the appropriate use of refinement in the Petri Net formalism, and this has had implications for the use of inheritance in UML statecharts. Charles was employed at the University of Tasmania from 1976 to 1998, and is now on the staff of the School of Computer Science at the University of Adelaide.

Robert Esser graduated from the University of Adelaide with a Bachelor of Engineering in 1981. He worked for a number of companies around the world predominantly in research and development over 14 years before gaining a Doctor of Technical Sciences degree at the Swiss Federal Institute of Technology, Zurich (ETH) in 1996. Subsequently he was appointed as a Motorola funded senior research fellow in the area of formal verification before gaining an academic position as senior lecturer in the School of Computer Science at the University of Adelaide. His interests include the modelling, simulation and verification of complex systems.