An Introduction to Pervasive Interface Automata*

M. Calder**, P. Gray, A. Miller, and C. Unsworth

Computing Science, University of Glasgow, U.K.

Abstract. Pervasive systems are often context-dependent, component based systems in which components expose interfaces and offer one or more services. These systems may evolve in unpredictable ways, often through component replacement. We present pervasive interface automata as a formalism for modelling components and their composition. Pervasive interface automata are based on the interface automata of Henzinger et al [3], with several significant differences. We expand their notion of input and output actions to combinations of input, output actions, and callable methods and method calls. Whereas interface automata have a refinement relation, we argue the crucial relation in pervasive systems is component *replacement*, which must include consideration of the services offered by a component and assumptions about the environment. We illustrate pervasive interface automata and component replacement with a small case study of a pervasive application for sports predictions.

1 Introduction

Pervasive systems are often context-dependent, component based systems that can evolve in unpredictable ways, through component addition (composition) and replacement. But unpredictability can have detrimental consequences for usability and for wide-spread adoption of systems: *how can we make component based evolution more predictable*?

The question is difficult because components are often designed and implemented incrementally by different development teams, or via end-user configurations, or they are mashups (e.g. make your own Android application mashup[7]). A traditional approach involving modelling and reasoning about full behavioural specifications is unlikely to be plausible. Consequently, we focus on the *interfaces* exposed by components and the *services* they offer. We define *pervasive interface automata* as a formalism for modelling the interfaces exposed by components and their composition. These automata are based on the interface automata of Henzinger et al [3], but there are several significant differences.

First, we expand their notion of input and output actions to combinations of input/output and calling and callable methods. We refer to the latter two

* This work was funded by the EPSRC grant, *Verifying Interoperability Requirements in Pervasive Systems* EP/F033206/1

** Corresponding author Muffy.Calder@glasgow.ac.uk

as master and slave actions, respectively. When components are composed, they synchronise on shared actions so that input/output and calling/called behaviour assumptions are met. Informally, this means that input/output and master/slave behaviours are (two-way) synchronised. We relax the synchronisation of components to allow a component offering a master action to wait until the appropriate slave action is offered. This allows both *busy send* and *busy receive*, where the original interface automata only allows a component to wait to receive, and not wait to send.

Second, we argue the crucial relation in pervasive systems is component *replacement*, which must include consideration of both services offered by a component and assumptions about the environment, where the environment is a composition of components. We include the environment because it may include actions that affect the interface of the component under consideration; specifically, it can cause some actions to become hidden (internal) or some choices to be removed.

As an example, consider a server component within a sports prediction application. The application keeps track of fixtures and results (e.g. a football league, or a tennis tournament) and allows users to make and share predictions in advance of actual events. The standard server component offers a service to *add predictions* and to *get predictions*. An online betting company might offer an alternative component that offers a service to *place a bet*, in addition to the previously mentioned services, the delivery of which relies on the availability of services *get data* and *add data* supplied by the betting company's online server component. Under what circumstances can we replace the standard component by the betting company's component? The latter relies on services from the company's online server component, which we call *environmental assumptions*; informally, we will allow replacement when environmental assumptions are met.

We introduce a linear temporal action logic to define service behaviour and define the satisfaction of the formulae by a pervasive interface automaton under assumptions about the environment. For a given environment, we can replace one component by another, with respect to a service, when both components satisfy the service in the environment. We judge the quality of a replacement of one component by another by the number of services that it preserves and any new services it may offer. We illustrate pervasive interface automata and component replacement with a small case study of an application for sports predictions, based on a real application.

In summary, the contributions of the paper are the following:

- definition of pervasive interface automata
- definition of action matching and algorithm for composition of pervasive interface automata
- logical specification of services
- satisfaction relation of pervasive interaction automaton and a service under environmental assumptions
- replacement relation between components, services and environment assumptions

– application of pervasive interaction automata and replacement relation to a case study involving sports predictions.

In the next section we give a brief overview of our case study, as motivation for pervasive interface automata, which we define formally in Sect. 3. In Sect. 4 we discuss action matching and we define automata composition by way of an algorithm. In Sect. 5 we introduce the concept of a service and define an action based logic for defining service behaviour; in the following section, Sect. 6 we define the replacement of one component for another, with respect to an environment and a set of services. Comparison of pervasive interface automata with interface automata and other related work is in Sect. 7. Our conclusions and directions for future work are in Sect. 8.

2 Case Study

We now present a case study which we use both as motivation and for explanation. The case study is a pervasive application for sports prediction, written within the Domino framework [2]. This application is mobile phone based and allows a user to see a list of upcoming sports fixtures and make predictions about the results. These predictions are then uploaded and stored on a remote server. A prediction league graphical interface allows a user to download predictions from multiple users (stored on a remote sever component) to compare them with the actual results in a league table. The architecture of this system is shown in Fig 1. Brief descriptions of the components follow.

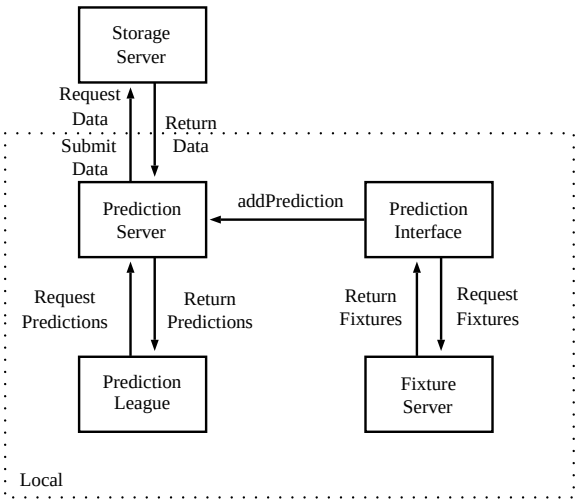

Fig. 1. Architecture of the sports prediction application.

Prediction Interface This component is a graphical user interface that allows a user to pull information about upcoming sporting fixtures from a fixture server. A user can input a prediction for one or more of the fixtures. The predictions are then sent to a prediction server for storage.

Fixture Server This is a passive component that responds to requests for fixture information. The data returned is a list of forthcoming sporting events, including information about times, dates, locations and the teams involved in the fixture.

Prediction Server This component accepts predictions from a prediction interface component and stores them in an external storage server component. The information sent to the server includes the fixture, the prediction and the user that made the prediction. Predictions are also retrieved from the storage server upon request.

Storage Server This is a web based generic data storage server component. It allows the storage of data and allows any user to retrieve any stored information.

Prediction League This component is a graphical user interface that allows a user to pull historical predictions from a prediction server along with the actual results to evaluate the predictions. The prediction league component can also retrieve and display the predictions from other users, allowing a user to compare their performance against that of their friends.

In this system most components are held on a local device (such as a mobile phone) only the storage server component is external. A user may wish to upgrade or replace any of the individual components to increase the overall systems functionality or to improve the user experience. However, any new component must be capable of providing all the services provided by the component it is replacing.

3 Pervasive Interface Automata

Pervasive Interface Automata are an extension of interface automata [3] with pervasive systems in mind. The main difference is the addition of annotations to actions. These annotations include ! and ? to indicate output and input respectively, and $\circ$ and $\star$ to represent slave and master actions respectively. Master actions are instigated by the component and slave actions are instigated by the environment (i.e. some other component). We have distinguished master/slave behaviour from input/output behaviour to ensure that we capture the notion of when a component requires external resources to function, i.e. to deliver a service. This is essentially the difference between *calling* a method and offering a method that *can be called*. For example, consider the four combinations of behaviour. If automaton P offers action $foo?^{\circ}$, then P is offering a callable method foo, which will receive data; if P offers action $foo?^{\star}$, then data is returned to P, from a method instigated by P. If automaton P offers action $foo!^{\circ}$, then P

is offering a callable method foo, which delivers data; if P offers action $foo!^{\star}$, then data is sent by P, from a method instigated by P.

Masters synchronise with slaves, and inputs synchronise with outputs. However, there is an asymmetry between masters and slaves. Whereas components with master actions (i.e. method calls) *require* slave actions (i.e. callable methods) in order to function properly, the converse is not true. More precisely, if a component reaches a state in which a master action is offered, a synchronising slave action is required at that point. On the other hand, if a component reaches a state in which a slave action is offered, and there is no corresponding synchronising master action, that slave action can be considered spare capacity; that is the action is on offer, but no other component requires it. In these circumstances, it can be ignored.

Definition 1. *A Pervasive Interface Automaton* $P = \langle V_P, V_P^{init}, A_P, T_P \rangle$ *where*

- $V_P = \{v_1, v_2, \ldots, v_{|V_P|}\}$ *is a finite set of states*
- $V_P^{init} \subseteq V_P$ *is the set of initial states*
- $A_P = \{a_1, a_2, \ldots, a_{|A_P|}\}$ *is the finite set of actions,*
  - *where an action* $a = name[?^{\star}|?^{\circ}|!^{\star}|!^{\circ}]$
- $T_P \subseteq V_P \times A_P \times V_P$ *is the set of steps (state transitions)*

Action annotations indicate the following. Annotations ! and ? denote input and output, respectively, and $\star$ and $\circ$ denote master and slave, respectively. An action that has no annotation is a hidden (internal) action. A_P° is defined to be the set of all slave actions, i.e. actions with a $\circ$ annotation, and $A_P^{\star}$ is defined to be the set of all master actions. We often use graphical representations of example automata.

To aid later description, a number of functions are now defined.

- $A_P(v)$ is the set of actions enabled at state v
  - an action a is enabled in state v if there exists $(v, a, v') \in T_P$
  - $A_P^{\circ}(v)$ is the set of slave actions enabled at state v
  - $A_P^{\star}(v)$ is the set of master actions enabled at state v
- $source(t)$ returns the state v, where transition $t = (v, a, v')$
- $act(t)$ returns the action a, where transition $t = (v, a, v')$
- $target(t)$ returns the state v', where transition $t = (v, a, v')$
- $\rho = \{t_1, t_2, \ldots\}$ is a path, defined as an ordered multiset of transitions
  - ρ_i the i^{th} transition in ρ
  - $\forall t_i, t_{i+1} \in \rho, target(t_i) = source(t_{i+1})$
  - $\rho \in P$ means that $\forall \rho_i \in \rho \implies \rho_i \in T_P$

An automaton P is said to be closed if it does not require any external resources to function. Input/output actions require external resources to function. However, slave actions (annotated with $\circ$) are assumed to be spare capacity, thus they will not be performed unless synchronised with some other component. Therefore, only master actions need be considered when determining if an

automaton is open or closed. However, it may be the case that some master actions are only enabled in states that require a transition involving a slave action to be taken to be reached from the initial state. Such unreachable actions will not be considered. The set of input/output requirements for an automaton is now defined.

Definition 2. *The set of input/output requirements for automaton P is*

$$req(P) = \{a \in A_P^\star | \exists \rho \in P, \exists t_i \in \rho, act(t_i) = a, \forall j < i, act(t_j) \notin A_P^\circ\}$$

If $req(P) = \emptyset$ then P is closed and P is open otherwise.

4 Composition

Before two automata can be composed, it needs to be established how they are to interact. This is a non-trivial problem, which will need a domain specific solution. For the purposes of this document, it will be assumed that automata will synchronise on action names. For example, if automaton P has an action $foo?^\circ$ and automaton Q has the action $foo!^\star$, the composition $P \otimes Q$ will have both actions combined into a single hidden action foo. The set $match(P, P \otimes Q)$ is used to show how actions in the automaton P are mapped to the actions in the product automaton $P \otimes Q$. For example in the case mentioned above, $(foo?^\circ, foo) \in match(P, P \otimes Q)$, meaning the slave input action foo in P is mapped to the hidden action foo in $P \otimes Q$. Similarly, $(foo?^\star, foo) \in match(Q, P \otimes Q)$. For each action $a \in A_P$ we assume there is a corresponding pair $(a, a') \in match(P, P \otimes Q)$, such that neither a nor a' appear in any other pair in $match(P, P \otimes Q)$. In other words, a matches, or synchronises uniquely with a' in $P \otimes Q$. Note, our notion of 2-way synchronisation is similar to CCS [10], where action a matches action $\overline{a}$. But, whereas in CCS a and $\overline{a}$ synchronise to become hidden τ, we assume the name of the hidden action is retained. Note that, $match(P, P \otimes Q)$ is used to map all actions from P to $P \otimes Q$ not just the synchronised actions.

For the default case of matching actions, if $a_P \in A_P$ and $a_Q \in A_Q$ are to be synchronised in $P \otimes Q$ then a_P and a_Q must be compatible. Meaning that master output actions match slave input actions and master input actions match slave output actions. Both a_P and a_Q are matched to the same hidden action in $P \otimes Q$. Alternately, if an action $a_P \in A_P$ is not to be synchronised then it will be unchanged in $P \otimes Q$.

Pervasive interface automata P and Q are composable if each automaton can perform its shared actions as required. That is, whenever P has an enabled shared master action either Q is also capable of performing the corresponding slave action, or P is able to wait until Q is ready to perform the slave action. In practice this means that if a shared master action a is enabled in a state $v \in V_P$ but not enabled in a product state $(v, u) \in V_{P \otimes Q}$ then all paths originating in (v, u) must include a state in which a is enabled. When allowing an automaton to wait we do not distinguish between input and output actions, therefore, we

allow both busy send and busy receive. Note that while P is waiting Q may need to interact with one or more other components, in which case the availability of these components will be a factor in the assessment of the composability of P and Q. Such requirements can be modelled as environmental assumptions, meaning that for P and Q to be composable the environment must meet these assumptions.

4.1 Composition

The composition of automata has two stages. The first is to generate the product of the two automata. The second attempts to validate the product as a valid composition.

We first define $\mathit{shared}(P, Q)$ as the set of shared actions in the product of $P \otimes Q$:

$$\mathit{shared}(P,Q) := \{a|(a_P, a) \in \mathit{match}(P, P \otimes Q)\} \cap \{a|(a_Q, a) \in \mathit{match}(Q, P \otimes Q)\}$$

Product We now define the product of automata $P \otimes Q$ as:

$$\begin{array}{l}
V^{init}_{P\otimes Q} = V^{init}_P \times V^{init}_Q \\
A_{P\otimes Q} = \{a'|(a,a') \in \mathit{match}(P, P\otimes Q)\} \cup \{a'|(a,a') \in \mathit{match}(Q, P\otimes Q)\} \\
T^{prov}_{P\otimes Q} = \\
\qquad \{((v_1,u),a',(v_2,u))|\ u \in V_Q, ((v_1),a,(v_2)) \in T_P, \\
\qquad\qquad\qquad (a,a') \in \mathit{match}(P, P\otimes Q), a' \notin \mathit{shared}(P,Q)\} \cup \\
\qquad \{((v,u_1),a',(v,u_2))|\ v \in V_P, ((u_1),a,(u_2)) \in T_Q, \\
\qquad\qquad\qquad (a,a') \in \mathit{match}(Q, P\otimes Q), a' \notin \mathit{shared}(P,Q)\} \cup \\
\qquad \{((v_1,u_1),a',(v_2,u_2))|\ ((v_1),a_1,(v_2)) \in T_P, ((u_1),a_2,(u_2)) \in T_Q, \\
\qquad\qquad\qquad\qquad (a_1,a') \in \mathit{match}(P, P\otimes Q), \\
\qquad\qquad\qquad\qquad (a_2,a') \in \mathit{match}(Q, P\otimes Q)\} \\
s \in V_{P\otimes Q} \iff \exists \rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ \mathit{source}(\rho_1) \in V^{init}_{P\otimes Q} \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad \mathit{target}(\rho_n) = s \\
\qquad\qquad\qquad\qquad\qquad\qquad \wedge \quad \forall \rho_i \in \rho.\ \rho_i \in T^{prov}_{P\otimes Q} \\
t \in T_{P\otimes Q} \iff t \in T^{prov}_{P\otimes Q} \wedge \mathit{source}(t) \in V_{P\otimes Q}
\end{array}$$

The set of initial states of $P \otimes Q$, $V^{init}_{P\otimes Q}$ is the product of the two sets of initial states V^{init}_P and V^{init}_Q. The action set $A_{P\otimes Q}$ is the union of the action sets of P and Q, respecting the matchings $\mathit{match}(P, P \otimes Q)$ and $\mathit{match}(Q, P \otimes Q)$ and $T^{prov}_{P\otimes Q}$ is the provisional set of transitions for $P \otimes Q$, which is used only as a construct to aid the definition of the product. A set of transitions is added to $T^{prov}_{P\otimes Q}$ for each non-shared transition in T_P; for a transition (v_1, a, v_2) this set consists of a transition $((v_1, u), a', (v_2, u))$ for each $u \in V_Q$, where $(a, a') \in \mathit{match}(P, P \otimes Q)$. Similarly, a set of transitions is added to $T^{prov}_{P\otimes Q}$ for each non-shared transition in T_Q. For every pair of transitions $t_p \in T_P$ and $t_q \in T_Q$,

where t_p and t_q involve matching shared actions, a transition t is added to $T^{prov}_{P\otimes Q}$, where $source(t) = (source(t_p), source(t_q))$, $target(t) = (target(t_p), target(t_q))$ and $act(t) = a$, where $(act(t_p), a) \in match(P, P \otimes Q)$. The set of states $V_{P\otimes Q}$ contains all states reachable via a path constructed of transitions from $T^{prov}_{P\otimes Q}$ and originating from a state in $V^{init}_{P\otimes Q}$. Finally, the set of transitions $T_{P\otimes Q}$ consists of all transitions $t \in T^{prov}_{P\otimes Q}$, where $source(t) \in V_{P\otimes Q}$.

Composition validation Informally, a product is a valid composition if the following two properties hold for master transitions in P (and Q, respectively). First, for every transition in P that involves a master action, there is a corresponding transition in $P \otimes Q$. Second, if v is a state of $P \otimes Q$ at which master action a is enabled, and a corresponds to a master action of P, then there is a path prefix in $P \otimes Q$ that contains a, all actions occurring prior to a are hidden, and they do not involve a state change for P.

The product $P \otimes Q$ is a valid composition iff:

$$
\begin{aligned}
&\forall t \in T_P.\ act(t) \in A^{\star}_P \implies \\
&\qquad \exists t' \in T_{P\otimes Q}.\ (t' = ((source(t), \_), a, (target(t), \_)) \\
&\qquad\qquad \wedge\ (act(t), a) \in match(P, P \otimes Q)) \\
&\quad \wedge \quad \forall v \in V_{P\otimes Q}.\ (v = (source(t), u)) \implies \\
&\qquad\qquad \exists path\rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ (\rho \in P \otimes Q \\
&\qquad\qquad\qquad \wedge\ source(\rho_1) = (source(t), u) \\
&\qquad\qquad\qquad \wedge\ act(\rho_n) = a \\
&\qquad\qquad\qquad \wedge\ \forall i : 1 \leq i < n. \\
&\qquad\qquad\qquad\qquad (source(\rho_i) = (source(t), \_) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ act(\rho_i)\ \textit{is hidden}))
\end{aligned}
$$

and

$$
\begin{aligned}
&\forall t \in T_Q.\ act(t) \in A^{\star}_Q \implies \\
&\qquad \exists t' \in T_{P\otimes Q}.\ (t' = ((source(t), \_), a, (target(t), \_)) \\
&\qquad\qquad \wedge\ (act(t), a) \in match(Q, P \otimes Q)) \\
&\quad \wedge \quad \forall v \in V_{P\otimes Q}.\ (v = (source(t), u)) \implies \\
&\qquad\qquad \exists path\rho = \{\rho_1, \ldots, \rho_n, \ldots\}.\ (\rho \in P \otimes Q \\
&\qquad\qquad\qquad \wedge\ source(\rho_1) = (source(t), u) \\
&\qquad\qquad\qquad \wedge\ act(\rho_n) = a \\
&\qquad\qquad\qquad \wedge\ \forall i : 1 \leq i < n. \\
&\qquad\qquad\qquad\qquad (source(\rho_i) = (source(t), \_) \\
&\qquad\qquad\qquad\qquad\qquad \wedge\ act(\rho_i)\ \textit{is hidden}))
\end{aligned}
$$

where _ is any state in the relevant component automaton.

4.2 Composition Examples

A simple example illustrates the role of master actions in composition. Consider the automata given in Fig. 2, assume a and b are shared actions and x is hidden (so not shared). In the composition, EX1 $\otimes$ EX2, master action $a!^{\star}$ waits for $a?^{\circ}$ at $(0, 0)$, i.e. they do not synchronise until $(0, 1)$. The slave $b!^{\circ}$ is never

synchronised. If however $b!^{\circ}$ is replaced by $b!^{\star}$ in $EX1$, then EX1 $\otimes$ EX2 would be *invalid*.

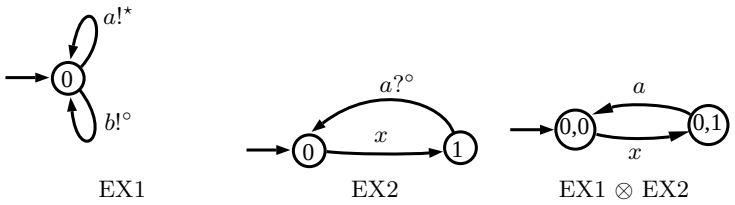

Fig. 2. Pervasive interface automata examples

As another example, consider the prediction server and storage server components described in Sect. 2, represented as pervasive interface automata in Fig. 3.

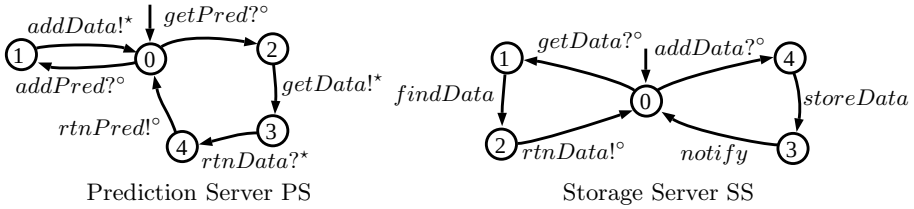

Fig. 3. Pervasive interface automata examples

The composition of these two automata will synchronise over the set of shared actions $\{getData, rtnData, addData\}$. The composition is shown in Fig. 4. Note the occurrence of a busy send in this example. From the product state (2,4), PS needs to be able to perform the master action $getData$ but in state 4 SS is not yet ready to provide the matching slave action. Therefore, PS will then wait in state 2 until SS performs the $storeData$ and $notify$ actions before it returns to state 0 in which it is ready to perform the matching $getData$ slave action. If SS was never enabled to receive the request, then the composition would be invalid.

5 Services

A service[1] is something that a component can do, such as respond to a data request, distribute information, store and retrieve data, etc; internal detail is not

[1] Similar to a web service [1].

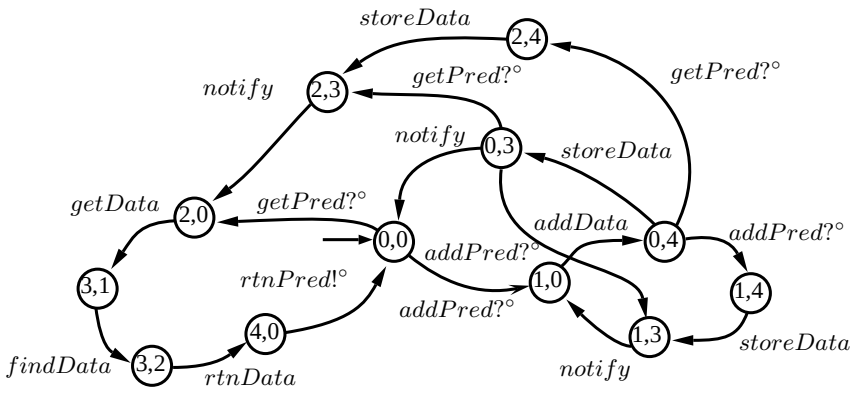

Fig. 4. The composition of PS and SS, PS ⊗ SS.

relevant. For example, a fixture server component P may offer the $getFixture$ service, a service that always responds to the method call $getFix()$ by returning a list of fixtures via a return method $rtnFix()$. In component P, the service is offered in a straightforward way because the fixture list is held internally. Another component, P' say, may also offer the same service, even though it obtains the list of fixtures from a third component. For example, P' may respond to the $getFix()$ call by calling a third component to obtain the fixture list, which it then returns via the method $rtnFix()$. In both cases, P and P' offer the same $getFixture$ service.

In terms of pervasive interface automata, a service can be described as a property defined in our own simple custom logic, defined below.

5.1 Logic for Services

We now describe a linear temporal action logic for defining services (a simplification of that in [6]). The logic is defined over paths ρ of a pervasive interface automaton P. Here a and b are (annotated) actions.

Syntax:

$\phi = tt \mid \mathit{offer}\ a\ \phi \mid a \rightsquigarrow b\ \phi$	path formulae
$\Sigma = \forall\phi \mid \exists\phi \mid \Sigma \wedge \Sigma \mid \Sigma \vee \Sigma$	service formulae

Semantics:

$\rho \models tt$	*always*
$\rho \models \mathit{offer}\ a\ \phi$	iff $\exists n \geq 1.\ \rho = \{\dots, t_n, \dots\}$
	and $act(t_n) = a$
	and $\forall_{i<n}\ act(t_i)$ are hidden actions
	and $\{t_{n+1}, \dots\} \models \phi$
$\rho \models a \rightsquigarrow b\ \phi$	iff $\exists n \geq 1, \exists m \geq n.\ \rho = \{\dots, t_n, \dots, t_m, \dots\}$

$$
\begin{array}{lll}
 & \text{and} & act(t_n) = a \\
 & \text{and} & act(t_m) = b \\
 & \text{and} & \forall_{i<n}\ act(t_i) \text{ are hidden actions} \\
 & \text{and} & \forall_{n<i<m}\ act(t_i) \text{ are hidden actions} \\
 & \text{and} & \{t_{m+1}, \dots\} \models \phi \\
\text{or } \nexists n \geq 1.\ \rho\{\dots, t_n, \dots\} & & \\
 & \text{and} & act(t_n) = a \\
P \models \forall\phi & \text{iff} & \forall \rho \in P.\ \ \rho \models \phi \\
P \models \exists\phi & \text{iff} & \exists \rho \in P.\ \ \rho \models \phi \\
P \models \Sigma_1 \wedge \Sigma_2 & \text{iff} & P \models \Sigma_1 \wedge P \models \Sigma_2 \\
P \models \Sigma_1 \vee \Sigma_2 & \text{iff} & P \models \Sigma_1 \vee P \models \Sigma_2
\end{array}
$$

Note that the actions preceding a are hidden in both *offer* $a\ \phi$ and $a \rightsquigarrow b\ \phi$. This expresses the requirement that the initiating action a of a service is available (e.g. cannot be blocked by waiting on another component), and if a service is initiated, then it is completed, e.g. a service is not abandoned.

5.2 Typical services

A service often involves a response to a request (a liveness property). More precisely, after possible hidden actions, it offers the *request*, which is a callable method. After further possible hidden actions, it either *sends* a response, which is a method call, or it offers to *respond*, which is a callable method. The initial request may be accompanied by data (an input); the response may also be accompanied by data (an output).

Formally, assuming actions *request* (slave), *send* (master), and *respond* (slave), these two service are expressed in our logic by:

1. request/send: $\forall(request?^\circ \rightsquigarrow send!^\star\ tt) \wedge \exists(\mathit{offer}\ request?^\circ\ tt)$
2. request/respond: $\forall(request?^\circ \rightsquigarrow respond!^\circ\ tt) \wedge \exists(\mathit{offer}\ request?^\circ\ tt)$

The second conjunct: $\exists(\mathit{offer}\ request?^\circ\ tt)$, serves to ensure the service is not trivially satisfied, i.e. the first action is offered by at least one path. As examples, the first type of service is offered by the prediction server PS (Fig. 3): $addPred?^\circ$ is always followed by $addData!^\star$, and the second is offered by the storage server SS: $getData?^\circ$ is always followed by $rtnData!^\circ$.

5.3 Components, environments and services

Our notion of a component automaton *offering* a service is not simply satisfaction of a service formula, we also take into account the impact and requirements placed upon the *environment*. We assume an environment is itself a composition of components. How to quantify the impact is subtle; it is not sufficient to check $P \models \Sigma$ (does a component P offer the service Σ), nor is it appropriate to check $P \otimes E \models \Sigma$ (does the composition of P with environment E offer Σ). In the

former, Σ may not be satisfied because there are non-hidden actions that do not occur in Σ, but they will be hidden when the component is composed with the environment and in the latter, after composition, the actions occurring in Σ may have become hidden. Instead we consider an *abstraction* of environments that, if fulfilled, means that P offers the service Σ *to* E.

Specifically when checking if P offers the service Σ to E, we consider the availability of actions we require E to offer. We refer to the set of requirements for the availability of actions as A. The availability set A contains (action,state) pairs. A is split into two subsets A^+ and A^-. If $(a, s) \in A^+$ then the environment must offer action a in state s, if $(a, s) \in A^-$ then the environment must not offer action a in state s.

We define a set *cpath* of modified paths of a component automaton. There are two cases to consider. First, if an action (slave or master) can *always* be matched by the environment, then we assume the actions represent *hidden activity*. Second, if a slave action can *never* be matched with the environment, and it is offered from a component state that offers any other type of action (slave or hidden), then the slave action represents *spare capacity*. Note, we assume angelic non-determinism; this means that paths representing spare capacity will not be taken.

In the following, we write f°, g^* etc. to stand for any slave or master action respectively, that is we ignore input and output annotations. We call these abstract actions and when we say that abstract action f° matches f^*, we assume the underlying input and output annotations match as required. We refer to the set of actions in a service formula by $\alpha(\Sigma)$; for example, $\alpha(\forall a \rightsquigarrow\ b\ tt) = \{a, b\}$.

Definition 3. *The alphabet of a service formula, $\alpha(\Sigma)$, is the set of all actions occurring in the path sub-formulas of Σ.*

Definition 4. *Given component automaton P, service Σ and environment assumptions A, define the set $cpath(P, \Sigma, A^+, A^-)$, as all the paths of P constructed in the usual way* except*, when constructing the paths*

- *for transition t, if $act(t) = f^*$, $f^* \notin \alpha(\Sigma)$, and $(f^\circ, source(t)) \in A^+$, then replace $f^\star$ by f in t (hide master),*

- *for transition t, if $act(t) = f^\circ$, $f^\circ \notin \alpha(\Sigma)$, and $(f^\star, source(t)) \in A^+$, then replace f° by f in t (hide slave),*

- *for transition t, if $act(t) = f^\circ$,$f^\circ \notin \alpha(\Sigma)$, and $(f^\star, source(t)) \in A^-$, then any (sub)path beginning with t is excluded (remove spare capacity).*

Note that, under the conditions given above, some slave actions can be safely excluded from the set of paths. This is not the case for master actions as, by definition, a component requires to be able to perform them in order to function correctly.

As illustration of spare capacity and hidden activity, consider the two automata in Fig. 5 and the service $\Sigma = \forall(g1^\circ \rightsquigarrow g2^\circ\ tt)$. The alphabet of Σ is $\{g1^\circ, g2^\circ\}$. Both P_1 and P_2 include one hidden action, I.

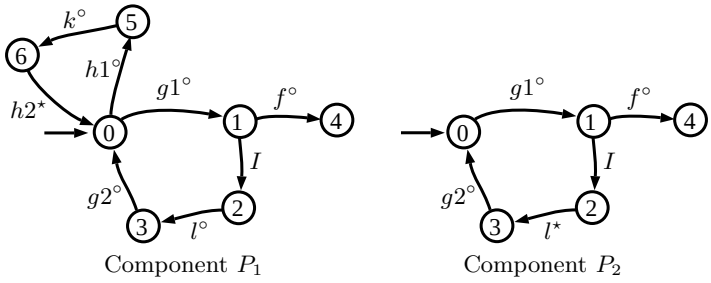

Fig. 5. Component automata, with abstract actions

The paths in $cpath(P_1, \Sigma, \{(l^\star, 2)\}, \{(f^\star, 1)\})$ start with either prefix $\rho_1 = \{h1^\circ, k^\circ, h2^\star\}$ or $\rho_2 = g1^\circ, I, l, g2^\circ\}$. As both $\rho_1 \models \Sigma$ and $\rho_2 \models \Sigma$, P_1 can offer Σ in an environment which offers $l^\star$ whenever P_1 is in state 2 and does not offer $f^\star$ when P_1 is in state 1. All the paths in $cpath(P_2, \Sigma, \{(l^\circ, 2)\}, \{(f^\star, 1)\})$ start with prefix $\rho_2 = \{g1^\circ, I, l, g2^\circ\}$. As $\rho_2 \models \Sigma$, P_2 can offer Σ in an environment which offers l° whenever P_2 is in state 2 and does not offer $f^\star$ when P_2 is in state 1.

Note that in both cases the path prefix $g1^\circ, f^\circ$ is excluded because the action f° is forbidden in state 1 (the environment will never offer a matching master action), whereas in component P_1, the path prefix $h1^\circ, k^\circ$ is included as spare capacity. In both cases the action l° has become hidden in *cpath*, because it matches an action offered by E.

In summary, the set *cpath* allows us to specify services that are offered by a component, in the context of an environment that meets assumptions A^+ and A^-. Together, A^+ and A^- are an abstraction of a class of environments. From here on we assume quantification over paths in $cpath(P, \Sigma, A^+, A^-)$ and we denote satisfaction with respect to A^+ and A^- by $\models_A$, defined thus.

Definition 5. *Given component automaton P, environment assumptions A, and path formula ϕ,*

$$
\begin{array}{lll}
P \models_A \forall\phi & \textit{iff} & \forall\rho \in cpath(P, \phi, A^+, A^-).\rho \models \phi \\
P \models_A \exists\phi & \textit{iff} & \exists\rho \in cpath(P, \phi, A^+, A^-).\rho \models \phi \\
P \models_A \Sigma_1 \wedge \Pi_2 & \textit{iff} & P \models_A \Sigma_1 \wedge P \models_A \Sigma_2 \\
P \models_A \Sigma_1 \vee \Pi_2 & \textit{iff} & P \models_A \Sigma_1 \vee P \models_A \Sigma_2
\end{array}
$$

By abuse of notation we extend satisfaction to sets of services S, and write $P \models_A S$, when $\forall\Sigma \in S.P \models_A \Sigma$.

Note that P offers Σ to E can mean either P offers a service that E needs, or P offers a service that persists after P is composed with E. In the latter case we have $P \otimes E \models \Sigma$, but in the former case this is not true because the actions in Σ have become hidden.

5.4 Service Example

Consider again the *getFixture* service example. If we translate the offering of method *getFix*() as an action $getFix?^\circ$ and the *rtnFix*() method as action $rtnFix!^\circ$, we can express the *getFixture* service as the property:

$$getFixture = \forall(getFix?^\circ \rightsquigarrow rtnFix!^\circ\ tt) \wedge \exists(\mathit{offer}\ getFix?^\circ\ tt)$$

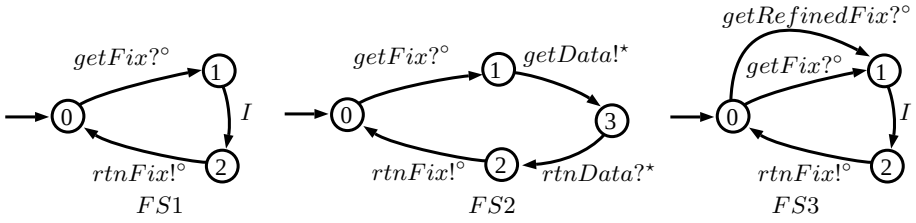

Fig. 6. Three example automata.

Figure 6 shows three example automata. The first, $FS1$, represents a component that can supply a fixture list without referring to any other component. $FS2$ represents a component that, upon request, requires a third party component (i.e. a storage server) before the requested fixture list can be returned. The third, $FS3$, offers the same basic functionality as $FS1$ (and thus offers the *getFix* service), however, it also offers the addition option of requesting a refined fixture list. More formally, we have:

- $FS1 \models_A getFixture$, where $A^+ = \emptyset$ and $A^- = \emptyset$.
- $FS2 \models_A getFixture$, where $A^+ = \{(getData?^\circ, 1), (rtnData!^\circ, 3)\}$ and $A^- = \emptyset$.
- $FS3 \models_A getFixture$, where $A^+ = \emptyset$ and $A^- = \emptyset$.

6 Replacement

Pervasive systems are adaptive and evolutionary by definition, meaning that components will be updated and replaced over time. Therefore, it is useful to know what effects such replacements will have on the system. Will the new component have the same functionality as the component it is replacing? To this end, we now define the replacement relation, which allows us to check if one component can be replaced by another in a given environment, with respect to a given set of services.

Definition 6. *For given component automata P, P', service Σ and environment requirements A, if $P \models_A \Sigma$, then P may be replaced by P' if $P' \models_{A'} \Sigma$ and $A' \subseteq A$.*

Replacement of a component with respect to a set of services is the natural extension of replacement with respect to a single service. This can be qualified: the *quality* of a replacement depends upon which services are preserved in a replacement, i.e.

Definition 7. *Given component automata P, P_1, P_2, sets of services SS, S_1, and S_2, and environment assumptions A, if $P \models_A SS$, $P_1 \models_A S_1$, and $P_2 \models_A S_2$, we say that component P_1 is a better replacement than component P_2 for P when $(S_2 \cap SS) \subset (S_1 \cap SS) \subseteq SS$.*

Note, we consider intersections of services, since the new components may offer additional services.

6.1 Replacement Example

Consider fixture server component $FS1$ from Fig. 6 that offers the $getFixture$ service defined as $\forall(getFix?^\circ \rightsquigarrow rtnFix!^\circ) \wedge \exists(\mathit{offer}\ getFix?^\circ\ tt)$, with no environment assumptions. Which components could replace $FS1$?

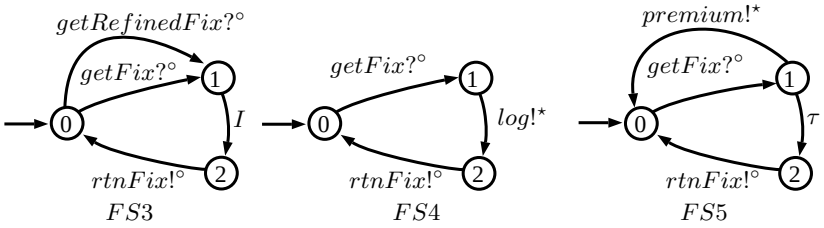

Fig. 7. Three potential replacement automata for FS1.

Component $FS3$ offers the $getFixture$ service with no environmental assumptions: $FS3 \models_A getFixture$ where $A^+ = \emptyset, A^- = \emptyset$. It also offers the additional option of requesting a refined fixture list. Therefore $FS3$ can replace $FS1$ in any environment; as $FS3$ offers additional functionality, this replacement would be referred to as an upgrade. Component $FS4$ also offers the $getFixture$ service, however, it also logs each use of the service. Therefore, $FS4 \models_A getFixture$, where $A^+ = \{(log?^\circ, 1)\}$, $A^- = \emptyset$. That is, $FS4$ offers $getFixture$ only if a logging service is available in the environment. So $FS4$ can only replace $FS1$ in environments offering a logging service. A fifth component, $FS5$, may be offered by some commercial organisation that requires a subscription to have access to some premium content. In which case, a $getFix?^\circ$ request may result in an error message notifying the environment that the premium content is inaccessible. As $premium!^\star$ is a master action, it cannot be ignored. Therefore, $FS5$ would not be a viable replacement for $FS1$, as $FS5 \not\models_A getFix$, for any A.

7 Comparison with Interface Automata and Session Types

Pervasive Interface Automata are based on Interface automata [3–5]. However, the addition classification of non-hidden actions as either master (*) or slave ($^\circ$), results in a richer action set. This combined with the more relaxed definition of composability of pervasive interface automata make them more appropriate for the context of pervasive systems.

A state $s = (f, g)$ in the composition of two interface automata F and G is said to be an *error state* if there is a shared action a that is an output action in F and enabled in f, but the corresponding action is not enabled in g. This disallows a component to wait for another to be ready before it performs an output action. Such conditions are prevalent and desirable in pervasive systems. Our notion of master and slave actions allow us to both define and embrace this kind of behaviour.

A crucial aspect of pervasive interface automata is that they allow us to formally define the notion of *replacement* of components, with respect to services. This concept relies heavily on our categorisation of actions as either master or slave, and so is not possible with interface automata.

The stated objectives in [11] are similar to our own, however the authors do not differentiate between master and slave actions. They also have a less rich definition of services.

Pervasive interface automata (and indeed interface automata) also bear a superficial resemblance to *session types* [8, 9]. A session represents possible sequences of communication events between processes. Communication events include synchronous message passing and the passing of channel names. Again there is no notion of master/slave actions, and properties are restricted to the matching of communication events.

8 Conclusion and Future Work

We have introduced pervasive interface automata as a formalism for modelling interfaces offered by components. Our motivation is managing predictability in component-based systems, especially in pervasive systems where components are regularly composed and replaced. Distinctive features of our automata include the separation of actions according to input or output, and method call or callable method. Composition of automata involves synchronisation on input/output and calling/called actions.

We do not just model interfaces, but also reason about services, which we define using a linear temporal action logic. We define the notion of a component offering a service *to* an environment (a composition of components) by considering the assumptions we need to make about the environment. If an environment meets those assumptions, then we can be assured that either the component meets the (service) needs of the environment or the service will persist after composition. In either case, we can proceed with composition. A key relation

is replacement, which may add new functionality, but ensures that services are still offered, given environment assumptions. Key concepts are illustrated with a mobile phone based application for sports predictions. Our long term goal is to derive pervasive interface automata automatically from code; this is future work.

References

1. Alonso, G., Casati, F., Kuno, H., Machiraju, V.: Web Services: Concepts, Architectures and Applications. Data-centric systems and applications, Springer-Verlag (2004)
2. Bell, M., Hall, M., Chalmers, M., Gray, P., B., B.: Domino: Exploring mobile collaborative software adaptaion. In: Fishkin, K., Scheile, B., Nixon, P., Quigley, A. (eds.) Proceedings of the 4th international conference on Pervasive Computing (PERVASIVE 2006). Lecture Notes in Computer Science, vol. 3968, pp. 153–168. Springer-Verlag, Dublin, Ireland (May 2006)
3. de Alfaro, L., Henzinger, T.: Interface automata. SIGSOFT Software Engineering Notes 26(5), 109–120 (2001)
4. de Alfaro, L., Henzinger, T.: Interface theories for component-based design. In: Henzinger, T., Kirsch, C. (eds.) Proceedings of the 1st International Workshop on Embedded Software (EMSOFT). Lecture Notes in Computer Science, vol. 2211, pp. 148–165. Springer-Verlag, Tahoe City, CA, USA (October 2001)
5. de Alfaro, L., Henzinger, T.: Interface-based design. Engineering Theories of Software-intensive Systems 195, 83–104 (2005)
6. de Nicola, R., Vaandrager, F.: Action versus state based logics for transition systems. In: Guessarian, I. (ed.) Semantics of Systems of Concurrent Processes. Lecture Notes in Computer Science, vol. 469, pp. 407–419. Springer-Verlag, La Roche Posay, France (April 1990)
7. Google: App inventor for android. http://appinventor.googlelabs.com (July 2010)
8. Honda, K.: Types for dyadic interaction. In: Best, E. (ed.) Proceedings of the 4th International Conference on Concurrency Theory (CONCUR '93). Lecture Notes in Computer Science, vol. 715, pp. 509–523. Springer-Verlag, Hildesheim, Germany (August 1993)
9. Honda, K., Vasconcelos, V., Kubo, M.: Language primitives and type discipline for structured communication-based programming. In: Proceedings of the European Symposium on Programming (ESOP'98). Lecture Notes in Computer Science, vol. 1381, pp. 123–138. Springer-Verlag, Lisbon, Portugal (March-April 1998)
10. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer-Verlag (1980)
11. Černá, I., Vařeková, P., Zimmerova, B.: Component substitutability via equivalencies of component-interaction automata. Electronic Notes in Theoretical Computer Science (ENTCS) 182, 39–55 (2007)