Transactional Reduction of Component Compositions

Serge Haddad[1] and Pascal Poizat[2,3]

[1] LAMSADE UMR 7024 CNRS, Université Paris Dauphine, France
Serge.Haddad@lamsade.dauphine.fr
[2] IBISC FRE 2873 CNRS, Université d'Évry Val d'Essonne, France
[3] ARLES Project, INRIA Rocquencourt, France
Pascal.Poizat@inria.fr

Abstract. Behavioural protocols are beneficial to Component-Based Software Engineering and Service-Oriented Computing as they foster automatic procedures for discovery, composition, composition correctness checking and adaptation. However, resulting composition models (*e.g.*, orchestrations or adaptors) often contain redundant or useless parts yielding the state explosion problem. Mechanisms to reduce the state space of behavioural composition models are therefore required. While reduction techniques are numerous, *e.g.*, in the process algebraic framework, none is suited to compositions where provided/required services correspond to transactions of lower-level individual event based communications. In this article we address this issue through the definition of a dedicated model and reduction techniques. They support transactions and are therefore applicable to service architectures.

1 Introduction

Component-Based Software Engineering (CBSE) postulates that components should be reusable from their interfaces [27]. Usual Interface Description Languages (IDL) address composition issues at the signature (operations) level. However, compositions made up of components compatible at the signature level may still present problems, such as deadlock, due to incompatible protocols [13]. In the last years, the need for taking into account protocol descriptions within component interfaces through the use of *Behavioural IDLs* has emerged as a solution to this issue. BIDLs yield more precise descriptions of components. They support component discovery, composability and substitutability checking (see, *e.g.*, [6,24,3,15]) and, if mismatch is detected, its automatic solving thanks to adaptor generation (see, *e.g.*, [28,26,18,9,14]). With the emergence of Service Oriented Architectures (SOA) [22], behavioural techniques are also valuable, *e.g.*, to discover and compose services [10,4], to verify service orchestrations and choreographies [25,7,17] or to build adaptors [11]. BIDLs usually rely on Labelled Transition Systems (LTS), *i.e.*, finite automata-like models where transition labels correspond to the events exchanged between communicating components or

services. Several works rather use process algebras such as the π-calculus in order to ensure conciseness of behavioural descriptions, yet verification techniques rely on the process algebras operational semantics to obtain LTSs.

These behavioural techniques, grounding on operations such as LTS products, often yield big global (system-level) models for compositions or adaptations, *i.e.*, coordinators or adaptors, which also contain redundant or useless parts. This occurs even when all the basic component models are optimal with respect to some standard criterium, *e.g.*, their number of states. This problem limits the applicability of composition and adaptation techniques, especially in domains where they are to be applied at run-time on low-resources devices, *e.g.*, pervasive computing or ambient intelligence. *Reduction techniques supporting component and service*[4] *composition and adaptation are therefore required.*

Transactions are important in component composition, *e.g.*, for Web Services [19]. Here, we address transactions from an applicative point of view: to ensure a given high-level service (the transaction), components usually proceed by exchanging several lower-level events. For example, to book a tourism package, service bookTour, a client should give in sequence elements about the country, the hotel requirements, and eventually price constraints. At the same time, to achieve this, the service may itself communicate with external services such as bookHotel, bookPlane and rentACar. The overall complexity of the bookTour service is adequately encapsulated into the (application-level) transaction concept. Transactions are also important in adaptation where they correspond to long-run sets of exchanges one wants to ensure using adaptors. Deadlock-freedom adaptation [14] is closely related to component final states, which in turn enable the definition of transactions in component protocols. *This makes the support for transaction a mandatory feature of behavioural reduction techniques* since in order to consider behaviours equivalent, and thereafter remove duplicates, complete transactions should to be taken into account.

Related Work. The usual techniques to deal with complexity problems are abstraction, on-the-fly, compositional and equivalence techniques. *Abstraction*, *e.g.*, [8], is used to achieve behavioural descriptions at a high level, avoiding details. A problem with abstraction is that it can be difficult to relate abstract results (*e.g.*, a composition scenario or an adaptor) and lower-level models (*e.g.*, a Web Service orchestration in BPEL4WS). *On-the-fly techniques*, *e.g.*, [20], compute global LTSs not before but during a given process. Branches can be discarded if not relevant or not consistent with reference to the process issue. *Compositional techniques* rely on the fact that some properties can lift from the local level (in a component) to the global one (in a composite). However, many interesting properties such as deadlock freedom are not compositional [2]. Various *equivalences or reductions techniques* have been developped in the field of process algebras and reused afterwards for component models (see, *e.g.*, [24,25]). They are based on the hiding of internal or synchronized events (using τ transitions). A first problem is that the ability of two components to synchronize

[4] In the sequel, we use *component* as a general term covering both software components and services, *i.e.*, mainly an entity to be composed.

is an important element for the usefulness of a composition. Synchronizations also yield a structuring information supporting the implementation of coordinators or adaptors: to which subcomponent event they do correspond. Hence, they cannot just be hidden and removed.

Moreover, abstraction, on-the-fly, compositional and reduction techniques do not support transactions. Action refinement and related equivalences [5] should do. Yet, action refinement is not suited to component composition or adaptation. This is first because the relations between two connected components cannot be always seen as an unidirectional refinement. Moreover, action refinement relates two components while composition or adaptation may apply on a wider scale. In [15] an approach based on component interaction automata with generic (over a set of events) equivalence and substitutability notions is proposed. However, its absence of specific treatment for component final states may prevent its support for transactions and deadlock-freedom adaptation.

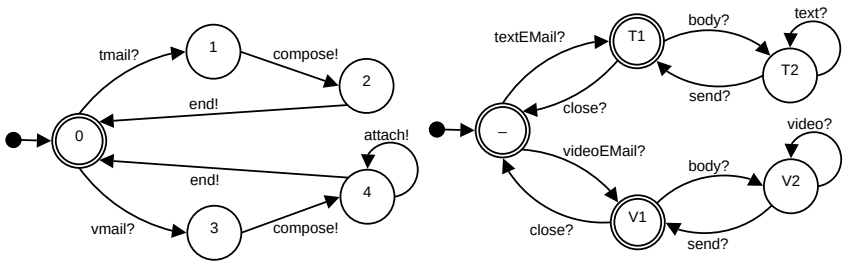
Contribution. The contributions of this article are twofold. First we propose a hierarchical component model with behavioural descriptions and expressive binding mechanisms combining different degrees of synchronization and encapsulation. These binding mechanisms enable one to define composition and adaptation contracts that would not be expressible in other component models such as the Fractal ADL [12] or UML 2.0 component diagrams [21] due to consistency constraints between component interface or port names. Our model supports open systems and enables one to achieve compositionality (composites are components). As discussed in related work, almost all reduction techniques forget the structure of composites and are thus inappropriate when *one does not want to (or cannot) re-design the subcomponents*. So our second contribution are reduction techniques which, on the one hand, take into account the transactional nature of communications between components and, on the other hand, do not modify the internal behaviour of the subcomponents. As a side effect, they also enforce deadlock-freedom adaptation between components that was previously handled by specific algorithms.

Organization. The article is organised as follows. In Section 2, we motivate the need for specific features in hierarchical component models and we informally introduce ours. It is then formalized in Section 3. Section 4 defines transaction-based reductions and corresponding algorithms are given. Finally, we conclude and present perspectives in Section 5.

2 Informal Presentation of the Model

Behaviours. As advocated in the introduction, component models should take into account means to define the behavioural interfaces of components through BIDLs. Let us introduce this part of our model using a simple example. Two components, an email server (SERVER) and an email composer (GUI) are interacting altogether and with the user. Their behavioural interfaces are described by LTSs (Fig. 1) where transition labels denote events that take place at the level of the component, either receptions or emissions. Receptions correspond to

provided services and emissions to required ones. LTSs can be obtained either at design-time as a form of behavioural contract for components, but may also be obtained by reverse engineering code.



(a) GUI (b) SERVER

Fig. 1. Mail System – Components (LTS)

At the composer level, one begins with a user opening a new window for a simple email (tmail?) or a video email (vmail?), which requires a special authentication not modelled here for conciseness. Afterwards, different user actions can be performed on the window. To keep concise, we only represent the corresponding triggered emissions in GUI. A text input (triggering compose!) is possibly followed by attachments (triggering attach!) for video email, and the mail is asked to be send (triggering end!). The server works on a session mode and allows, again with authentication, two kind of sessions: one dedicated to emails with text file attachments (textEMail?) and one dedicated to emails with video attachments (videoEMail?). Content is received using body?, attachments with either text? or video?, and the sending request with send?. Sessions are closed with close?. The usual notations are used for initial (black bullet) and final (hollow circles) states.

Architectural descriptions and hierarchical models. Simple components may be either the starting point of an architectural design process or the result of a discovery procedure [4]. In both cases, their composition has then to be described. This can take different forms, it can be directly given by the designer, or it can be computed from a high-level service or property description using conversation integration [4], service aggregation [10] or component adaptation [14]. However, in both cases, what one ends up with is an architectural description where correspondences between required and provided services are given. We advocate that, in order to deal with complexity, a composition model has to be hierarchical and to yield composites that can be related to components, and therefore, once defined, be reused in other higher-level composites. UML 2.0 component diagrams and Fractal ADL are such models and we refer to these for more details on the interests of hierarchical notations.

Expressive inter-component bindings. Even if components are meant to be reused, one may not expect all the components in an architecture have been

designed to match perfectly, neither at the signature, nor at the behavioural level [13]. Therefore, it is important to be able to describe *composition/adaptation mapping contracts* where correspondences between required and provided services may not correspond to name identity (*e.g.*, the end! required service in GUI corresponds to the send? provided service in SERVER), or could even be non one-to-one mappings (*e.g.*, the services related to the opening and closing of sessions provided by SERVER have no counterpart in GUI which is not session-oriented). These are current limitations of the UML 2.0 and Fractal ADL models which impose restrictions on bindings between components interfaces.

The role of transactions and their support. One may expect from a correct adaptation of the SERVER and GUI components that, of course they are able to communicate in spite of their incompatible interfaces, but also that the adaptor ensures the system eventually ends up in a global state where both SERVER and GUI are in a stable (final) state, namely _, T1 or V1, and 0. We base our approach on *implicit transactions* that correspond to this notion, and are sequences of transitions begining in an initial or a final state and ending in a final state. Explicit transactions are a complementary approach that could be supported with additional definitions (sequences of events) to be given for components. In GUI transactions are the sending of a text email and the sending of a video email. In SERVER, transactions correspond to the opening and the closing of sessions, to the creation of a text email and the creation of a video email. LTSs models provide the support for implicit transactions for free, however, to perform reduction more elements are to be taken into account.

The reduction of a composition model is a process that results in removing behaviours from it, mainly removing transitions. One expects from such a process that, in every context, replacing a component by one of its reductions can be achieved without hurt, *e.g.*, without introducing new deadlocks. This corresponds to the substitutability concept, *i.e.*, that all *useful* behaviours are still available for further composition. There is therefore a need for means to define the utility of transitions and sequences of transitions with reference to the available transactions in components. Reduction is usually enabled in composites by hiding and then removing synchronized events, we have seen in the introduction the problems with this. Here for example, this would apply to the synchronizing between end! and send?. Not only this synchronizing is an important information as it ends a global-level transaction (the sending of an email) hence it is a witness of it, but its removal also makes it impossible to implement it afterwards as a communication between end! in GUI and send? in SERVER.

Composition = synchronization + encapsulation. Using the basic components behavioural models and architectural descriptions one may obtain the behaviour of the global system using formal semantics (and hence, tools). However, as we have seen above, the correspondence between services is often confused with the encaspulation level of these services, *i.e.*, synchronized correspondences in-between the subcomponents of a composite are internal (hidden) while ports remaining free may eventually be exported so that the composite interface is defined in terms of its subcomponents ones. This misses distinction in

nature between *synchronization* (related to communication) and *encapsulation* (related to observability). We advocate that architectural composition should provide means to describe separately synchronization and encapsulation. We define *binding connectors* (or connectors for short) as an architectural level concept supporting the definition of both synchronization and encapsulation. Synchronization is defined thanks to *internal bindings* which relate a binding connector with at least one subcomponent port. Encapsulation is defined thanks to *external bindings* which relate a binding connector with at most one port of the composite. A component port can be either *synchronizable*, and in such a case it can be synchronized or not, or *observable*. Four different encaspulation levels are possible: inhibition, hiding, observability and synchronizability. Component ports not bound to some connector are inhibited. Connectors not bound to composite ports corresponds to the internal level and events which may be removed by reduction. Connectors bound to synchronizable ports of composites are synchronizable (support for n-ary synchronisation). In between, observability acts as an intermediate encapsulation level and is used to denote internal, yet useful, information for transactions. Binding connectors are a solution to the issues related to transactions and reductions presented above. They propose a good balance between the possible hiding of synchronizations (to enable reduction, yet stressing their possible utility thanks to the observability notion) and the possible retrieval of synchronized events (thanks to the internal bindings information).

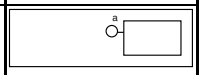
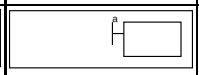
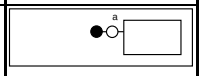
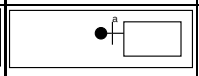
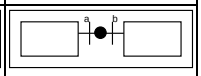
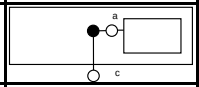
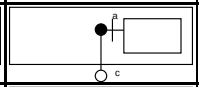
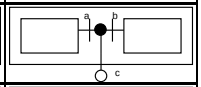
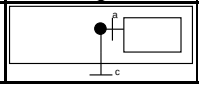
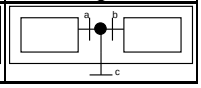
	observable	synchronizable	
		not synchronized	synchronized
inhibited			not applicable
internal			
observable			
synchronizable	forbidden		

Table 1. Architectural Notations

In Table 1 we present the graphical notation for our architectural concepts. Binding connectors are denoted with black bullets, observable component ports with white bullets and synchronizable ports with Ts as in Fractal ADL.

The architecture corresponding to the composition/adaptation contract for our example is given in Figure 2. Two ports of GUI are connected to the composite synchronizable ports in order to model the possible action of the user on this component. There are three connectors for internal synchronization between components: one for the body of emails (compose! with body?), one for

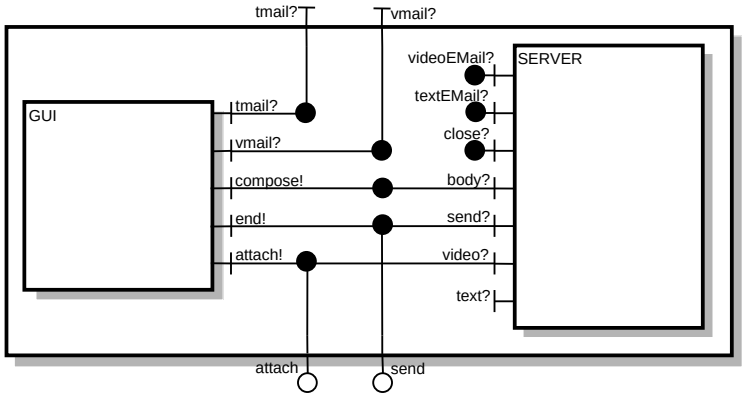


Fig. 2. Mail System – System Architecture

video attachments (attach! with video?) and one for the sending requests (end! with send?). In order to denote that the two latter ones should not be taken into account when reducing the composite behaviour (*i.e.*, should not be removed), they are made observable (using respectively observable ports attach and send in the composite interface). Three ports of SERVER are synchronizable but not synchronized. This means that in the composition corresponding events may be generated by an adaptor when needed (see [14] for more details on such adaptation contracts). Yet, they are hidden which means that the corresponding events are not observable and may be removed when reduction is performed. To end, one port of SERVER, namely text? is inhibited (not used in the composition/adaptation contract).

3 Formalization of the Model

We focus on events triggered by (basic or composite) components. There are two possible related views of such events: (i) the external view (encapsulation) which distinguishes events depending on the ability to observe them and to synchronize with them, and (ii) the internal view (synchronization) which additionally includes in case of a composite event, the activities of subcomponents and synchronizations between them that have produced the external event. The external view leads to *elementary alphabets* while the internal view leads to *structured alphabets*. Given an event of the latter kind we will obtain an event of the former one by abstraction. The next definitions formalize these concepts.

Definition 1 (Elementary Alphabet). *An elementary alphabet Σ is given by the partition $\Sigma = \Sigma^s \uplus \Sigma^o \uplus \{\tau\}$ where Σ^s represents the synchronizable events, Σ^o represents the observable (and non synchronizable) events and τ represents an internal action. Furthermore, Σ does not include $\perp$ (do-nothing event).*

In the sequel, we use the letter Σ for elementary alphabets.

Example 1. Let us describe the elementary alphabets of our example. The elementary alphabet of the GUI subcomponent, Σ_{GUI} is defined by $\Sigma^o_{\mathsf{GUI}} = \emptyset$ and $\Sigma^s_{\mathsf{GUI}} = \{\mathsf{tmail?}, \mathsf{vmail?}, \mathsf{compose?}, \mathsf{end!}, \mathsf{attach!}\}$. The elementary alphabet of the SERVER subcomponent, Σ_{SERVER} is defined by $\Sigma^o_{\mathsf{SERVER}} = \emptyset$ and $\Sigma^s_{\mathsf{SERVER}} = \{\mathsf{videoEMail?}, \mathsf{textEMail?}, \mathsf{close?}, \mathsf{body?}, \mathsf{send?}, \mathsf{video?}, \mathsf{text?}\}$. The elementary alphabet of the composite component (*i.e.*, its external view), Σ is defined by $\Sigma^s = \{\mathsf{tmail?}, \mathsf{vmail?}\}$ and $\Sigma^o = \{\mathsf{attach}, \mathsf{send}\}$.

A structured alphabet is associated with a possibly composite component.

Definition 2 (Structured Alphabet). *A structured alphabet* $A = \Sigma \times \prod_{i \in Id} (A_i \cup \{\perp\})$ *is recursively defined by:* Σ *an elementary alphabet,* Id *a (possibly empty) finite totally ordered set, and* A_i *a structured alphabet for every* $i \in Id$*. To denote an item of* A*, we use the tuple notation* $v = v_0 : \langle v_1, \ldots, v_n \rangle$ *with* $Id = \{id_1, \ldots, id_n\}$*,* $v_0 \in \Sigma$ *and* $\forall 1 \leq i \leq n, v_i \in A_i \cup \{\perp\}$*.*

Id represents the set of subcomponent identifiers and the occurrence of $\perp$ in v_i, where v belongs to the structured alphabet, means that subcomponent id_i does not participate to the synchronization denoted by v. Obviously Id is isomorphic to $\{1, \ldots, n\}$. However in the component-based framework, component identifiers are more appropriate. Note that every elementary alphabet can be viewed as a structured one with $Id = \emptyset$. The mapping $v \mapsto v_0$ corresponds to the abstraction related to the external view. Hence, in $v = v_0 : \langle v_1, \ldots, v_n \rangle$, v_0 plays a special role. We therefore introduce $root(v) = v_0$. Similarly, we note $root(A) = \Sigma$. The alphabets A_i are called subalphabets of A. We denote the empty word by ε. Let A be a structured alphabet and $w \in A^*$, the *observable part* of w, denoted $\lceil w \rceil$ is recursively defined by $\lceil \varepsilon \rceil = \varepsilon$, $\forall a \in A$, if $root(a) = \tau$ then $\lceil a \rceil = \varepsilon$ else $\lceil a \rceil = root(a)$ and finally $\lceil ww' \rceil = \lceil w \rceil \lceil w' \rceil$.

In our framework, component behaviours are described with *Labelled Transition Systems*.

Definition 3 (Labelled Transition System). *A Labelled Transition System (LTS)* $\mathcal{C} = \langle A, Q, I, F, \rightarrow \rangle$ *is defined by:* A*, a structured alphabet,* Q*, a finite set of states,* $I \subseteq Q$*, the subset of initial states,* $F \subseteq Q$*, the subset of final states, and* $\rightarrow \subseteq Q \times A \times Q$ *the transition relation. As usual,* $(q, a, q') \in \rightarrow$ *is also denoted by* $q \xrightarrow{a} q'$*. The observable language of* $\mathcal{C}$*,* $\mathcal{L}(\mathcal{C})$*, is defined as* $\mathcal{L}(\mathcal{C}) = \{w \mid \exists \sigma = q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_m} q_m \text{ s.t. } q_0 \in I, q_m \in F, w = \lceil a_1 \ldots a_m \rceil\}$.

We now introduce *mapping vectors* and *mapping contracts* which express component bindings in order to build a composite component. Mapping vectors are items of a specific structured alphabet whose root alphabet Σ corresponds to the interface of the composite and whose subalphabet indexed by i corresponds to the interface of component id_i which is (generally) different from the alphabet of this component. A mapping contract is a subset of mapping vectors representing all the possible "local" or "synchronized" events of the composite.

Definition 4 (Mapping Vectors and Mapping Contracts). *Let* Id *be a set of component identifiers,* $\mathcal{S} = \{\mathcal{C}_i\}_{i \in Id}$ *be a finite family of LTS, for* $i \in Id$*, let*

A_i denote the alphabet of $\mathcal{C}_i$ and let Σ be an alphabet. Then a mapping vector *v relative to $\mathcal{S}$ and Σ is an item of the structured alphabet $\Sigma \times \prod_{i \in Id} (root(A_i) \cup \{\bot\})$. Furthermore a mapping vector $v = v_0 : \langle v_1, \ldots, v_n \rangle$ fulfills the following requirements:*

- $\exists i \neq 0, v_i \notin \Sigma_i^s \cup \{\bot\} \Rightarrow v_0 \notin \Sigma^s \wedge \forall j \notin \{0, i\}, v_j = \bot$;
- $\exists i \neq 0, v_i = \tau \Rightarrow v_0 = \tau$.

A mapping contract *$\mathcal{V}$ relative to $\mathcal{S}$ and Σ, is a set of mapping vectors such that every mapping vector v with some $v_i = \tau$ belongs to $\mathcal{V}$.*

The requirements on mapping vectors and mapping contracts are consistent with our assumptions about the model (Tab. 1). Non synchronizable events of a subcomponent cannot be synchronized or transformed into a synchronizable event in the composite and internal events of a subcomponent cannot be made observable in the composite. The requirement about mapping contracts means that an internal event in a subcomponent cannot be inhibited in the composite.

The translation from the graphical notation to the formal model is straightforward. There is a mapping vector for each binding connector of the graphic, its root is either given by the composite port bound to this connector when it is present or τ. Each component of the mapping vector is either given by the corresponding component port bound to this connector when it is present or $\bot$.

Example 2. The mapping contract in Figure 2 is defined by: $\mathsf{tmail?} : \langle \mathsf{tmail?}, \bot \rangle$, $\mathsf{vmail?} : \langle \mathsf{vmail?}, \bot \rangle$, $\tau : \langle \bot, \mathsf{videoEMail?} \rangle$, $\tau : \langle \bot, \mathsf{textEMail?} \rangle$, $\tau : \langle \bot, \mathsf{close?} \rangle$, $\tau : \langle \mathsf{compose!}, \mathsf{body?} \rangle$, $\mathsf{send} : \langle \mathsf{end!}, \mathsf{send?} \rangle$, $\mathsf{attach} : \langle \mathsf{attach!}, \mathsf{video?} \rangle$ and the vectors relative to internal events of the subcomponents.

Synchronous product is used to give a formal semantics to composites.

Definition 5 (Synchronized Product of LTS). *Let $\mathcal{S} = \{\mathcal{C}_i\}_{i \in Id}$ be a finite family of LTS, Σ be an alphabet and $\mathcal{V}$ be a mapping contract relative to $\mathcal{S}$ and Σ, then the synchronized product of $\mathcal{S}$ w.r.t. $\mathcal{V}$ is the LTS $\Pi(\mathcal{S}, \mathcal{V}) = \langle A, Q, I, F, \rightarrow \rangle$ where:*

- $A = \Sigma \times \prod_{i \in Id}(A_i \cup \{\bot\})$, $Q = \prod_{i \in Id} Q_i$, $I = \prod_{i \in Id} I_i$, $F = \prod_{i \in Id} F_i$,
- $(q_1, \ldots, q_n) \xrightarrow{v_0 : \langle a_1, \ldots, a_n \rangle} (q'_1, \ldots, q'_n)$ *iff* $\exists v_0 : \langle v_1, \ldots, v_n \rangle \in \mathcal{V}$ *and* $\forall i \in Id$,
  - $v_i = \bot \Rightarrow a_i = \bot \wedge q'_i = q_i$,
  - $v_i \neq \bot \Rightarrow root(a_i) = v_i \wedge q_i \xrightarrow{a_i}_i q'_i$.

This semantics is supported by the ETS plugin [23]. It can be obtained in an *on-the-fly* way to be more efficient. Thus in practice, we reduce Q to be the set of reachable states from I.

Example 3. The synchronized product of the GUI LTS and the SERVER one is described in Figure 3. For sake of readability, for each (structured) event occurring in this LTS, we have only represented its root and when this root is τ we have not represented it. For instance, the arc from $(2, \mathsf{T2})$ to $(0, \mathsf{T1})$ should be labelled $\mathsf{send} : \langle \mathsf{end!}, \mathsf{send?} \rangle$ instead of send. The size of this LTS is $[13; 27]$ where 13 is the number of states and 27 is the number of transitions.

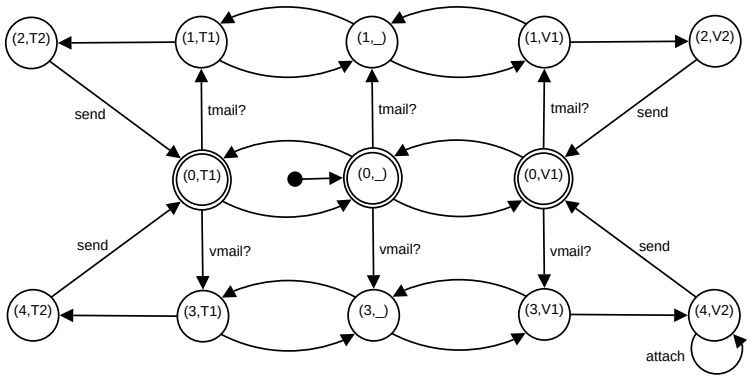


Fig. 3. Mail System – Resulting Adaptor/Coordinator ([13; 27] LTS)

4 Transaction-Based Reductions

The goal of this section is the design of algorithms which reduce the LTSs associated with compositions. Due to our assumptions about components, two requirements must be fulfilled by such algorithms: the reduction only proceeds by transition removals (and as a side effect possibly by state removals) and the reduction must preserve the capabilities of the composite w.r.t. its transactions. We introduce first the transaction concept.

Definition 6 (Transactions of an LTS). *Let $\mathcal{C} = \langle A, Q, I, F, \rightarrow \rangle$ be an LTS, a transaction $tr = (s, w, s')$ of $\mathcal{C}$ is such that $s \in I \cup F$, $s' \in F$, $w \in (root(A))^*$ and there exists a* witnessing sequence $\sigma = q_0 \xrightarrow{a_1} q_1 \ldots \xrightarrow{a_m} q_m$ *with $s = q_0$, $s' = q_m$, $\forall 0 < i < m, q_i \notin F$ and $\lceil a_1 \ldots a_m \rceil = w$. We denote by $Seq(tr)$ the set of witnessing sequences of transaction tr and by $\mathcal{L}(s, s') = \{w \mid (s, w, s') \text{ is a transaction}\}$, the language generated by transactions from s to s'.*

Example 4. Below, we exhibit the regular expresssions associated with every transaction language of the Figure 3 LTS:

$$\mathcal{L}((0,\_),(0,\_)) = \mathcal{L}((0,\mathsf{T1}),(0,\_)) = \mathcal{L}((0,\mathsf{V1}),(0,\_)) = \{\varepsilon\}$$

$$\mathcal{L}((0,\_),(0,\mathsf{T1})) = \mathcal{L}((0,\mathsf{T1}),(0,\mathsf{T1})) = \mathcal{L}((0,\mathsf{V1}),(0,\mathsf{T1})) = \mathsf{tmail?} \cdot \mathsf{send} + \mathsf{vmail?} \cdot \mathsf{send}$$

$$\mathcal{L}((0,\_),(0,\mathsf{V1})) = \mathcal{L}((0,\mathsf{T1}),(0,\mathsf{V1})) = \mathcal{L}((0,\mathsf{V1}),(0,\mathsf{V1})) = \mathsf{tmail?} \cdot \mathsf{send} + \mathsf{vmail?} \cdot \mathsf{attach}^* \cdot \mathsf{send}$$

Since we want to preserve the transaction capabilities, we introduce a specific notion of simulation between states, where only initial and final states are considered and the transactions are viewed as atomic transitions.

Definition 7 (Transaction Simulation Relation between States). *Let $\mathcal{C}$ and $\mathcal{C}'$ be two LTS and let $\mathcal{R}$ be a relation, $\mathcal{R} \subseteq (I \cup F) \times (I' \cup F')$. $\mathcal{R}$ is a transaction simulation relation iff for every pair (q_1, q_1') of $\mathcal{R}$ and every transaction $tr = (q_1, w, q_2)$ of $\mathcal{C}$, there is a transaction $tr' = (q_1', w, q_2')$ of $\mathcal{C}'$ with $(q_2, q_2') \in \mathcal{R}$.*
We define $\sqsubseteq$ by $\sqsubseteq = \bigcup\{\mathcal{R} \mid \mathcal{R} \text{ is a transaction simulation relation}\}$.

Algorithm 1 `transSimulation`

computes the transaction simulation relation between states of $\mathcal{C}$
inputs LTS $\mathcal{C} = \langle A, Q, I, F, \rightarrow \rangle$
outputs Relation $\mathcal{R} \subseteq (I \cup F) \times (I \cup F)$

1: **for all** $(i, j) \in (I \cup F) \times (I \cup F)$ **do** $\mathcal{R}[i, j] :=$ **true end for**
2: **repeat** *// fixed point algorithm for* $i \sqsubseteq j$
3: $end :=$ **true**
4: **for all** $(i, j) \in (I \cup F) \times (I \cup F)$ s.t. $i \neq j \wedge \mathcal{R}[i, j] =$ **true do**
5: **for all** $k \in F$ **do**
6: // *is* $\mathcal{L}(i, k) \subseteq \bigcup_{k' \text{ s.t. } \mathcal{R}[k,k']} \mathcal{L}(j, k')$*?*
7: $K' := \{k' \in F \mid \mathcal{R}[k, k'] =$ **true**$\}$
8: $\mathcal{R}[i, j] :=$`lgInclusion(transLTS`$(\mathcal{C}, i, \{k\})$`,transLTS`$(\mathcal{C}, j, K'))$
9: $end := end \wedge \mathcal{R}[i, j]$
10: **end for**
11: **end for**
12: **until** *end*
13: **return** $\mathcal{R}$

In the Figure 3 LTS, any final state simulates the other ones:

$$\forall s, s' \in \{(0, \_), (0, \mathsf{T1}), (0, \mathsf{V1})\}, s \sqsubseteq s'$$

Based on state simulation, the simulation of an LTS $\mathcal{C}$ by an LTS $\mathcal{C}'$ is defined. We require that every initial state of $\mathcal{C}$ is simulated by an initial state of $\mathcal{C}'$.

Definition 8 (Transaction Simulation between LTS). *Given two LTS* $\mathcal{C}$ *and* $\mathcal{C}'$*,* $\mathcal{C}'$ *simulates* $\mathcal{C}$ *denoted by* $\mathcal{C} \sqsubseteq \mathcal{C}'$ *iff* $\forall i \in I$, $\exists i' \in I'$, $i \sqsubseteq i'$*.*

We are now a position to define when an LTS $\mathcal{C}_{red}$ is a reduction of an LTS $\mathcal{C}$: it is obtained from $\mathcal{C}$ by removal of transitions and states and still simulates it.

Definition 9 (Reduction of a LTS). *Given two LTS* $\mathcal{C}$ *and* $\mathcal{C}_{red}$*,* $\mathcal{C}_{red}$ *is a reduction of* $\mathcal{C}$ *iff:* $Q' \subseteq Q$*,* $I' \subseteq I$*,* $F' \subseteq F$*,* $\rightarrow' \subseteq \rightarrow$*, and* $\mathcal{C} \sqsubseteq \mathcal{C}_{red}$*.*

Let us describe the principles of our reduction algorithm for an LTS $\mathcal{C}$:

1. Algorithm 1 computes the simulation relation between initial and final states of $\mathcal{C}$. It proceeds by iterative refinements of a relation until a fixed point has been reached. The number of iterations of this algorithm is polynomial w.r.t. the size of the LTS and every iteration involves a polynomial number of calls to the language inclusion procedure applied to simple transformations of $\mathcal{C}$.
2. Then, based on the simulation relation between states, Algorithm 2 computes a subset of initial states and a subset of final states such that the LTS, obtained by deleting the other initial and final states, simulates the original one.
3. At last Algorithm 3 examines every transition of the step 2 LTS whose label $\tau : \langle v_1, \ldots, v_n \rangle$ is such that $\exists i, v_i \in \Sigma_i^s$ and removes it if the resulting LTS

Algorithm 2 `stateReduction`

state-based reduction, constructs reduced LTS $\mathcal{C}'$ from LTS $\mathcal{C}$ with $\mathcal{C} \sqsubseteq \mathcal{C}'$
inputs LTS $\mathcal{C} = \langle A, Q, I, F, \rightarrow \rangle$
outputs reduced LTS $\mathcal{C}' = \langle A', Q', I', F', \rightarrow' \rangle$

1: $\mathcal{R}$:=`transSimulation`($\mathcal{C}$)
2: $heap$:= `getAMaximal`($\mathcal{R}, I$)
3: $front := heap$
4: **repeat**
5: extract some s from $front$
6: $candidates := F \cap$ `reach`(`transLTS`($\mathcal{C}, s, F$), $\{s\}$)
7: **for all** $f \in candidates \setminus heap$ **do**
8: $dom := \{f' \in candidates \setminus \{f\} \mid \mathcal{R}(f, f')\}$
9: **if** `lgInclusion`(`transLTS`($\mathcal{C}, s, \{f\}$),`transLTS`($\mathcal{C}, s, dom$)) **then**
10: remove f from $candidates$
11: **end if**
12: **end for**
13: $front := front \cup (candidates \setminus heap)$
14: $heap := heap \cup candidates$
15: **until** $front = \emptyset$
16: $I' := I \cap heap$; $F' := F \cap heap$
17: $Q' :=$ `reach`($\mathcal{C}, I'$) $\cap$ `coreach`($\mathcal{C}, F'$)
18: $I' := I' \cap Q'$; $F' := F' \cap Q'$; $\rightarrow' := \rightarrow \cap Q' \times A \times Q'$
19: **return** $\langle A, Q', I', F', \rightarrow' \rangle$

simulates the current one. The condition on labels ensures the approach is compatible with a grey-box vision of components where components are composed and/or adapted externally *without* modifying the way they internally work (*i.e.*, without removing internal or observable events).

The different steps of our reduction involve calls to `transLTS`. Given an LTS $\mathcal{C}$, an arbitrary state s of $\mathcal{C}$ and a subset of final states S, this function produces an LTS $\mathcal{C}'$ whose observable language is the set of suffixes of transactions in $\mathcal{C}$, starting from s and ending in S. After every reduction, we "clean" (in linear time) the LTS by eliminating the states that are not reachable from the initial states using the `reach` function and the ones that cannot reach a final state using the `coreach` function. This ensures deadlock-freedom adaptation. The (observable) language inclusion check between two LTS is performed by the `lgInclusion` function. It is the main factor of complexity as language inclusion is a PSPACE-complete problem. However the design of (empirically) efficient procedures is still an active topic of research with significant recent advances [16]. The procedure includes some non deterministic features (for instance the examination order of "τ transitions"). Thus it could be enlarged with heuristics in order to empirically improve its complexity but this is out of the scope of the current paper.

Algorithm 1 is based on a standard refinement procedure for checking simulation or bisimulation. Its specific feature is that it checks inclusion of languages rather than inclusion of set of labels (which entails an increasing of complexity).

Algorithm 2 starts with a maximal set of initial states given by function `getAMaximal` (line 2). The *heap* variable contains the current set of initial and final states that should be in the reduced LTS whereas the $front$ variable contains the subset of *heap* whose "future" has not yet been examined. The main loop (lines 4–15) analyzes the transactions initiated from a state s extracted from $front$. In line 6, it computes the final states reached by such a transaction and stores them in variable *candidates*. For every f, *candidate* not already present in *heap*, it looks whether the language of transactions $\mathcal{L}(s, f)$ is included in the union of the languages of transactions $\mathcal{L}(s, f')$ with f' a candidate simulating f. In the positive case, it removes f from *candidates* (lines 7–12). At the end of loop, the remaining *candidates* not already present in *heap* are added to *heap* and $front$. For the Figure 3 LTS, the algorithm starts with $front = heap = \{(0, \_)\}$. During the first loop, *candidates* is set to $\{(0, \_), (0, \mathsf{T1}), (0, \mathsf{V1})\}$, and then $(0, \mathsf{T1})$ is removed. Therefore, at the beginning of the second loop, $front = \{(0, \mathsf{V1})\}, heap = \{(0, \_), (0, \mathsf{V1})\}$. During the second loop, *candidates* is set to $\{(0, \_), (0, \mathsf{T1}), (0, \mathsf{V1})\}$, again $(0, \mathsf{T1})$ is removed and at the end of second loop, $front = \emptyset, heap = \{(0, \_), (0, \mathsf{V1})\}$. The resulting LTS is represented on the left-hand side of Figure 4.

Algorithm 3 main loop tries to remove (one by one) transitions which are unobservable at the composite level but are observable at the component level (lines 2–18). When the state reached s' from some state s by such a transition is a final state, then the subset of transaction suffixes that reach s' from s is reduced to the singleton $\{\varepsilon\}$. So, in order to remove the transition, the algorithm checks whether the empty word may be the suffix of a transaction starting in s, ending in a final state simulating s' without using this transition (lines 4–7). Otherwise it performs a similar test for every final state f reached from s (inner loop 10–15) comparing the languages of transaction suffixes. Starting from the LTS on the left-hand side of Figure 4, Algorithm 3 produces the right-hand side LTS. Note that even for such a small example, the reduction is significant w.r.t. the original LTS (*i.e.*, the size is approximatively divided by two).

A comparison with the usual reduction techniques (Fig. 5) demonstrates their inadequacy in our context: they are based on equivalences which are either too strong – bisimulation treats τs as regular transitions hence only removes few of them – or too weak – too many τ transitions are removed (*e.g.*, $\tau : \langle \mathsf{compose!}, \mathsf{body?} \rangle$ between states $(1, \mathsf{V1})$ and $(2, \mathsf{V2})$) which makes it impossible afterwards to implement the composition between components. Moreover, acting only at the composition level, these reduction techniques may make it necessary to change the subcomponent protocols in order to implement compositions, while we want to support a non intrusive approach for composition and adaptation.

5 Conclusion

In order to build efficient composite components, one needs both efficient basic components (which can be expected from, *e.g.*, Commercial-Off-The-Shelf) and efficient composition or adaptation techniques. This last constraint is related

Algorithm 3 `transReduction`

transaction reduction, constructs reduced LTS $\mathcal{C}'$ from LTS $\mathcal{C}$ with $\mathcal{C} \sqsubseteq \mathcal{C}'$
inputs LTS $\mathcal{C} = \langle A, Q, I, F, \rightarrow \rangle$
outputs reduced LTS $\mathcal{C}' = \langle A', Q', I', F', \rightarrow' \rangle$

1: $\mathcal{C}' := \mathcal{C}$
2: **for all** $t = s \xrightarrow{\tau:\langle v_1,\ldots,v_n\rangle} s'$ s.t. $\exists i, v_i \in \Sigma_i^s$ **do**
3: **if** $s' \in F'$ **then**
4: $dom := \{f \in F' \mid \mathcal{R}(s', f)\}$; $\mathcal{C}'' := \mathcal{C}'$; $\rightarrow'' := \rightarrow'' \setminus \{t\}$
5: **if** `lgInclusion`(`emptyWordLTS`(),`transLTS`($\mathcal{C}'', s, dom$)) **then**
6: $\rightarrow' := \rightarrow' \setminus \{t\}$
7: **end if**
8: **else**
9: del :=**true**
10: **for all** $f' \in F' \cap$ `reach`(`transLTS`($\mathcal{C}', s, F'$), $\{s\}$) **do**
11: $dom := \{d \in F' \mid \mathcal{R}(f', d)\}$; $\mathcal{C}'' := \mathcal{C}'$; $\rightarrow'' := \rightarrow'' \setminus \{t\}$
12: **if not** `lgInclusion`(`transLTS`($\mathcal{C}', s', \{f'\}$),`transLTS`($\mathcal{C}'', s, dom$)) **then**
13: del :=**false**; **break**
14: **end if**
15: **end for**
16: **if** del **then** $\rightarrow' := \rightarrow' \setminus \{t\}$ **endif**
17: **end if**
18: **end for**
19: $Q' :=$ `reach`($\mathcal{C}', I'$) $\cap$ `coreach`($\mathcal{C}', F'$)
20: $I' := I' \cap Q'$; $F' := F' \cap Q'$; $\rightarrow' := \rightarrow \cap Q' \times A \times Q'$
21: **return** $\mathcal{C}'$

to the basic techniques which underpin the composition or adaptation process, *e.g.* [10,4,14], but also to efficient reduction procedures for the resulting behavioural models. We have addressed this issue with techniques that take into account the transactional nature of communications between components. Reduction is supported by a component model with expressive binding mechanisms and different levels of synchronization and encapsulation.

A first perspective of this work concerns the integration of our reduction algorithms in a model-based adaptation tool [1] we have developed and assessment on real size case studies from the pervasive computing area. A second perspective is to relate our model-based reduction technique with adaptor implementation issues, mainly taking into account the controllability of events (*e.g.*, viewing emissions as non controllable events). Other perspectives are related to the enhancement of our reduction technique, addressing on-the-fly reduction (reduction while building the compositions or adaptors) and optimizing algorithms thanks to recent developments on language inclusion [16].

References

1. Adaptor, January 2007 distribution (LGPL licence). http://www.ibisc.univ-evry.fr/Members/Poizat/Adaptor, 2007.

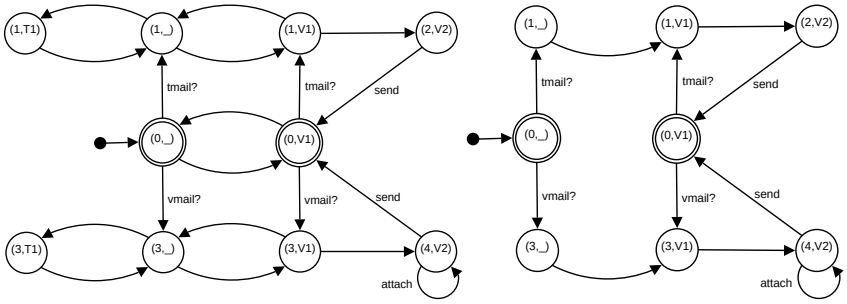


Fig. 4. Mail System – Reduced Adaptor/Coordinator (left: state reduction, [10; 19] LTS; right: transition reduction, [8; 11] LTS])

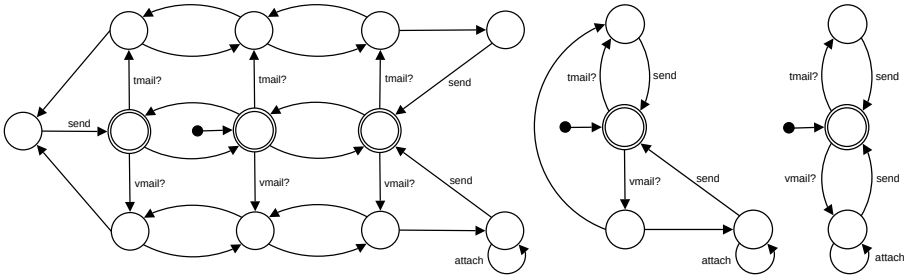


Fig. 5. Mail System – Reduced Adaptor/Coordinator (left: strong bisimulation reduction, [12; 26] LTS; center: weak bisimulation or branching reduction, [4; 7] LTS; right: trace or $\tau * a$ reduction, [3; 5] LTS)

2. F. Achermann and O. Nierstrasz. A calculus for reasoning about software composition. *Theoretical Computer Science*, 331(2–3):367–396, 2005.
3. C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In *Software Composition*, volume 4089 of *LNCS*, pages 18–33. Springer, 2006.
4. S. Ben Mokhtar, N. Georgantas, and V. Issarny. Ad Hoc Composition of User Tasks in Pervasive Computing Environments. In *Software Composition*, volume 3628 of *LNCS*, pages 31–46. Springer, 2005.
5. J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra*. North-Holland, Elsevier, 2001.
6. M. Bernardo and P. Inverardi, editors. *Formal Methods for Software Architectures*, volume 2804 of *LNCS*. Springer, 2003.
7. A. Betin-Can, T. Bultan, and X. Fu. Design for Verification for Asynchronously Communicating Web Services. In *International Conference on World Wide Web*, pages 750–759, 2005.
8. D. Beyer, T. Henzinger, R. Jhala, and R. Majumdar. Checking Memory Safety with Blast. In *International Conference on Fundamental Approaches to Software Engineering*, volume 3442 of *LNCS*, pages 2–18. Springer, 2005.
9. A. Bracciali, A. Brogi, and C. Canal. A Formal Approach to Component Adaptation. *Journal of Systems and Software*, 74(1):45–54, 2005.
10. A. Brogi, S. Corfini, and R. Popescu. Composition-Oriented Service Discovery. In *Software Composition*, volume 3628 of *LNCS*, pages 15–30. Springer, 2005.

11. A. Brogi and R. Popescu. Automated Generation of BPEL Adapters. In *International Conference on Service Oriented Computing*, volume 4294 of *LNCS*, pages 27–39. Springer, 2006.
12. E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and Its Support in Java. *Software Practice and Experience*, 36(11-12):1257–1284, 2006.
13. C. Canal, J. M. Murillo, and P. Poizat. Software Adaptation. *L'Objet*, 12(1):9–31, 2006. Special Issue on Coordination and Adaptation Techniques for Software Entities.
14. C. Canal, P. Poizat, and G. Salaün. Synchronizing Behavioural Mismatch in Software Composition. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, volume 4037 of *LNCS*, pages 63–77. Springer, 2006.
15. I. Cerná, P. Vareková, and B. Zimmerova. Component Substitutability via Equivalencies of Component-Interaction Automata. In *International Workshop on Formal Aspects of Component Software*, Electronic Notes in Theoretical Computer Science. Elsevier, 2006.
16. M. De Wulf, L. Doyen, T. Henzinger, and J.-F. Raskin. Antichains: A new algorithm for checking universality of finite automata. In *Computer-Aided Verification*, volume 4144 of *LNCS*, pages 17–30. Springer, 2006.
17. H. Foster, S. Uchitel, J. Magee, and J. Kramer. LTSA-WS: a tool for Model-Based Verification of Web Service Compositions and Choreography. In *International Conference on Software Engineering*, pages 771–774. ACM, 2006.
18. P. Inverardi and M. Tivoli. Deadlock Free Software Architectures for COM/DCOM Applications. *Journal of Systems and Software*, 65(3):173–183, 2003.
19. M. Little. Transactions and Web Services. *Communications of the ACM*, 46(10):49–54, 2003.
20. R. Mateescu and M. Sighireanu. Efficient On-the-Fly Model-Checking for Regular Alternation-Free Mu-Calculus. *Science of Computer Programming*, 46(3):255–281, 2003.
21. Objet Management Group. Unified Modeling Language: Superstructure. version 2.0, formal/05-07-04, August 2005.
22. M. P. Papazoglou and D. Georgakopoulos. Service-Oriented Computing. *Communications of the ACM*, 46(10):25–28, 2003.
23. P. Poizat. Eclipse Transition Systems. French National Network for Telecommunications Research (RNRT) STACS Deliverable, 2005.
24. P. Poizat, J.-C. Royer, and G. Salaün. Formal Methods for Component Description, Coordination and Adaptation. In *International Workshop on Coordination and Adaptation Techniques for Software Entities at ECOOP*, pages 89–100, 2004.
25. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. *International Journal of Business Process Integration and Management*, 1(2):116–128, 2006.
26. H. W. Schmidt and R. H. Reussner. Generating Adapters for Concurrent Component Protocol Synchronization. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 213–229, 2002.
27. C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley, 1998.
28. D. M. Yellin and R. E. Strom. Protocol Specifications and Components Adaptors. *ACM Transactions on Programming Languages and Systems*, 19(2):292–333, 1997.