www.ietdl.org

Published in IET Software
Received on 7th April 2012
Revised on 28th April 2013
Accepted on 3rd June 2013
doi: 10.1049/iet-sen.2012.0163

ISSN 1751-8806

Formal framework for specifying dynamic reconfiguration of adaptive systems

Jaber Karimpour, Robab Alyari, Ali A. Noroozi

Department of Computer Science, Faculty of Mathematics Science, University of Tabriz, 29 Bahman Blvd., 5166616471 Tabriz, Iran
E-mail: Karimpour@tabrizu.ac.ir

Abstract: In the real-world, there are many types of software systems and software engineers always deal with changes. The value of large systems decreases significantly as the requirements and operational environment change over time. Modern software systems are expected to have dynamic reconfigurations to cope with failure and changes. Software adaptation techniques try to overcome the change problem by reconfiguration. In this study, at first, the authors present a formal framework to represent the whole system and then, build a mathematical model called 'adaptor' based on adaptation contract and system architecture. The adaptor is used to define automatic fit between two different components of the system. Finally, for specifying the whole adaptor system the authors will introduce adaptor network using synchronisation vectors.

1 Introduction

New large computer systems are built from different components with uncertain behaviour. Behaviour of the components and the requirements of the system may change over time. The success of component-based systems depends on the capability of adapting to unpredictable situations or sudden changes. Software systems need to discover how they can exchange those parts which do not work properly or do not fit into requirements. It means that the system should reconfigure itself when it needs to upgrade or has an error that causes the system to fail in runtime.

An investigation in literature shows that the desirable properties of the adaptive systems architecture are as follows [1–3]:

- *Grounded*: The architecture needs to be able to take into account the 'real-world'. It needs to represent more than just logical relationship between or within abstract computational entities.
- *Exogenous*: The architecture cannot assume it has access to internal configuration or state of the components that build up the system. Coordination of behaviour or measurement of non-functional properties of the components must be external or exogenous to those components.
- *Self-managed*: It is commonly accepted that all parts of a complex system should be able to be independently described, implemented and deployed in modules [3]. The complexity of managing the relationship between entities in a system increases dramatically as the number of heterogeneous entities increases. In order to handle this complexity, management of the system should be distributed down to the level of modules rather than globally managing the system. In other words, these modular composites should be self-managed, as much as possible.
- *Recursive*: If description of the system at different levels of granularity and abstraction is based on the same architecture meta-model, the efficiency of the meta-model and comprehensibility of the design increases dramatically. For example, one of the strengths of the object-oriented methodology is that objects are composed of other objects, those which are also made of other objects too, and so on.
- *Practical*: The architectural meta-model concepts should be as simple as possible. Adaptive architectural model should ideally be based on a few powerful concepts that software developers can readily comprehend and apply.

1.1 Problem

The main problem addressed in this paper is how to build adaptive software systems that can reliably achieve system level goals in volatile environments, where the system itself may be built from components of uncertain behaviour, and where the requirements for software system may be changing.

To solve this problem, the system should be equipped with rich interfaces enabling external access to its functionality with the following specific characteristics:

1. Must have precise and sound syntax and semantics.
2. Must have interfaces description by signatures (operation names and types) and behaviours (interaction protocols).
3. Must solve the problem of consistency between two different parts.
4. Must be suitable for hierarchical description of component architecture.

© The Institution of Engineering and Technology 2013

5. Must allow the possibility of formally reasoning about the system behaviour.
6. Must allow the possibility of system modelling, simulation and verification.

Our solution to this problem, stated as our claim in Section 1.3, satisfies items 1–4, setting the stage for future work on the next two items.

1.2 Motivation

Beer's [4] viable system model (VSM) presents a control-theoretic approach to recursively structured adaptive systems, as well as adapting to changes in itself or its environment. We can say, VSM differentiates various types of control and also shows how control can be related to organisational structure, and how complexity can be managed by this structure. Structure means construction or framework of identifiable elements of a system (like components, entities, parts or …) and the way in which these elements are connected to each other. The structure defines: (i) which elements are connected to each other in a direct or indirect way and (ii) in which range, elements can communicate with each other. Tanenbaum and Steen [5] believe the main assumption in adaptive software is that the software should be allowed to modify itself with environmental changes. We know system engineers can halt the system and repair it but some kinds of systems cannot be shut down. Software adaptation and dynamic reconfiguration are the only way to overcome this problem and improve such systems.

Most of the works on adaptation use features like signature, protocol, quality of service and semantics. A lot of them prefer full automation of the process (restrictive approaches) [6–8] and the others support the specification of adaptation scenarios (generative approaches) [9–11]. The ones that advocate the first class believe in solving interoperability issues by decreasing the behaviour that may lead to mismatches but on the other hand, generative approaches build adaptors automatically from abstract specification, called adaptation contract to solve how the mismatches can be annihilated.

A good adaptation idea could be the ability of a system to regulate itself and change its structure as it interacts with the environment. This change of structure is performed in two ways. The first is the change or interchange of the elements within the structure like adding or removing them, the second way is modification of relationships between the elements that make up the system, like changing the ways in which these elements interact with each other. If the adaptive system has loosely coupled elements, it must have ways of determining these kinds of changes at runtime so that the system maintains its viability. A viable system is a system which could continue to survive (and, if required, continue to meet its goals) in uncertain and changing environments. Adaptation is the ability to not only continue to survive in uncertain and changing environments, but also to maintain the quality of meeting goals.

'Software architecture' is a high-level view of a software system as a configuration of components and connectors. In the software architecture context, adaptive architecture can be defined as dynamic structure and management where management activities to monitor the system, reconfigure the structure and regulate the behaviour over that structure.

We intend to develop a mathematical model called 'Adaptor' based on the adaptation contract and the system architecture to specify dynamic reconfiguration of a system.

1.3 Claim

We present a framework for satisfying items 1–4 of our problem and some properties for adaptive systems. To achieve this framework, we apply the work of Jin *et al.* [12] and the work of Isazadeh and Karimpour [13] to extend Cansado *et al.* work [14] about a formal framework for structural reconfiguration of components under behavioural adaptation.

1.4 Paper outline

The remainder of this paper is organised as follows. In Section 2, we give a brief review of the previous work and provide some background on formalisms that will be used throughout the paper. Section 3 describes our method by presenting the new notation. In this section, at first, we introduce the 'Adaptor' and then expand our method in hierarchical components by defining network of 'Adaptors'. In Section 4, a case study is presented. Finally, the paper is included in Section 5 with a summary of results.

2 Background and evaluation

In 1995, Shaw [15] proposed that, in some types of application, an architectural idiom based on control theory is appropriate. Since then many approaches have been proposed that develop ways to control or manage components. Just having a dynamic architecture with some management capability does not guarantee that the software system itself will be viable, that is, it will be able to survive and continue to meet its goals by maintaining its organisational integrity.

2.1 Approaches to representing adaptive architecture

Most of the research efforts in adaptive software architecture address one or more of the above properties of reconfigurable systems. Fig. 1 shows two categories of dynamic architectures, structure-centric and quality-centric. As illustrated in Fig. 1, structure-centric category shows the architectures that are primarily concerned with structure, and making sure that the dynamic structure is well-formed. Structure offers an account of what a system is made of, like a configuration of items, a collection of inter-related components or services. These approaches, which are primarily concerned with functional change, are largely formal. They are motivated both by the need to compose functionally well-formed systems, and the need to express the dynamic transformation of the structure. Bradbury [16]

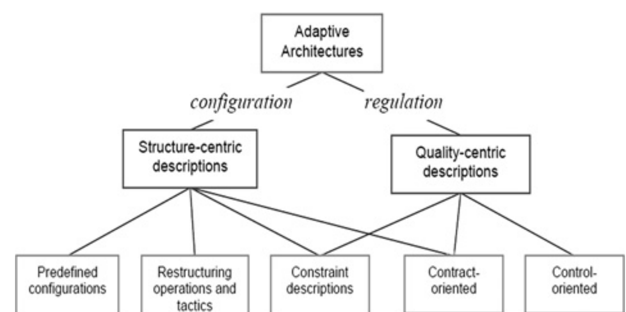

Fig. 1 *Representing dynamic architectures [1]*

© The Institution of Engineering and Technology 2013

surveys a number of formal dynamic architecture languages and evaluates the extent to which the formalisms support the specification of self-managing systems. The formalisms include graph-based, logic, process algebra and other approaches. Each approach has various strengths. Bradbury evaluates the languages in terms of component and connector addition/removal to/from the structure. Formalisms also vary according to whether they emphasise the behaviour of the system (as, say, naturally expressed in a process algebra) as opposed to an emphasis on the structure of the components and connectors (as, say, naturally expressed in a graph grammar).

The second category, that is, quality-centric descriptions, concentrates on the measurement of the qualities of interactions over a structure. Manager in these systems control the quality of interactions. These architectures are not mutually exclusive but, rather, represent different dominant themes for describing frameworks. The focus of these approaches to describing software architecture is non-functional transformation. These are concerned with representing structures in which changing performance and other qualities (reliability, resource allocation, security etc.) can be represented and managed. As shown in Fig. 1, classification of these descriptions can be according to the mechanisms for achieving configuration and/or regulation. Some of the frameworks, such as contract-oriented frameworks, can be seen as a mix of structure- and quality-centric descriptions. Also, a distinction can be made between contract-oriented and control-oriented approaches.

Frameworks that use control-oriented techniques regulate entities by monitoring the changes in the values of control variables, and then setting process variables. On the other hand, contract-oriented approaches regulate the interactions between entities by defining permissible types of interaction and performance levels of those interactions. Both approaches have some sort of monitoring mechanism to ensure that the required level of performance is being met, and take action to correct any underperformance. In control systems, this involves the controller changing some property of the controlled entity. In contracted systems, the entity needs to autonomously meet the requirement. Contracts can be viewed as connectors that define the structural relationships in a system, because they can be used to define relationships between components. Owing to this property, contracts (or connector types) can be used to describe both the structure and quality of relationships, as illustrated in Fig. 1.

2.2 Structure-centric frameworks

Structure-centric architectural descriptions can be classified as those that define reconfiguration operations and tactics, those that define constraints and the ones that define valid configurations. One of the structure-centric frameworks is ‘Plastik’ [17].

‘Plastik’ is a framework that supports a formally specified runtime reconfiguration. The architecture description language (ADL) has two sub-levels, as illustrated in Fig. 2. Definition of generic patterns (e.g. protocol stack style) by setting constraints so that, types of component, connectors and interface operations (‘properties’) can be composed, are in the style level. A configuration, defined by a style, is encapsulated in a ‘component framework’. The instance level particularises the style for a specific context.

The ‘system configurator’ is also divided into two levels. The singleton architectural configurator is responsible for

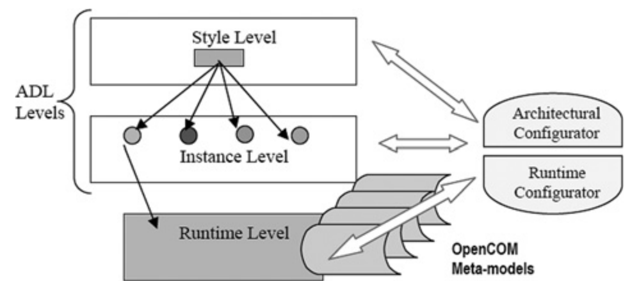

Fig. 2 *Plastik’s system architecture [17]*

accepting and validating reconfiguration requests from the ADL levels, whereas each deployed ‘component framework’ has a runtime configurator that manages the runtime level. Constraints at the ADL level are compiled into finite-state machines in the runtime configurators. These configurators are implemented in a scripting language that is generated by a compiler.

Plastik supports both the ‘programmed’ and ‘*ad hoc*’ reconfiguration. In *ad hoc* reconfiguration, system specifies certain invariants in which configurations cannot violate. This kind of configurations are not specified at the ADL level (only the constraints are specified), but change can be initiated from either the ADL and runtime levels. The other reconfigurations that can be foreseen at design time are programmed reconfigurations and are expressed as ‘predicate-action’ specifications.

Combining high-level reconfiguration concepts with a robust runtime component framework is the key advantage of Plastik’s approach. However, the problem is synchronisation between architectural layers and runtime layers that are separated into two representations connected by compiled scripts. This is particularly so because change can be initiated at either level. There is also no discussion about the runtime conditions that target reconfiguration can hold, nor is there a way to explicitly model non-functional requirement or change. The addition and removal of components are the only reconfiguration operations. Monitoring is not a part of the framework but is something that, it is claimed, can be provided by the components or a third party. Also, key aspects of the system design have been trialed, rather than being fully implemented.

2.3 Control-oriented frameworks

Control-oriented approaches to architectural adaptation adopt the paradigm of ‘software systems being control systems’. This paradigm is suggested by Shaw [15]. These frameworks define a control-loop with three phases: sense–evaluate–act. A separate control component(s) or layer that monitors the system and adapts the structure to changing environments or requirements is the main aspect of control-oriented approaches.

Policy-based-self-adaptive model (PobSAM) [18] is a control-oriented framework. A PobSAM model is composed of a collection of autonomous managers and managed actors. Autonomous managers are meta-actors responsible for monitoring and handling events by enforcing suitable policies. Each manager has a set of configurations containing adaptation policies and governing policies. A manager changes its configuration dynamically in response to the changing circumstances according to adaptation policies. The behaviour of managed actors is

© The Institution of Engineering and Technology 2013
IET Softw., 2013, Vol. 7, Iss. 5, pp. 258–270
doi: 10.1049/iet-sen.2012.0163

either governed by managers or cannot be directly controlled from outside. Governing policies are the rules that are applied while the system is in a stable state. Adaptation policies are rules that govern the transient states between two stable states while the system is changing. The set of manager's configurations is not fixed and may change dynamically by growing and evolving the system. In PobSAM, the managers monitor the actor's behaviour through the view layer and direct and adapt the system behaviour.

Another control-oriented approach is Rainbow [19]. This architecture assumes that the management framework has access to some measurement, resource discovery and affecting mechanism to observe and change the functional system. In Rainbow, adaptation strategies are globally defined across the architecture. The authors point out that having a central representation of the architecture makes the system subject to single-point failure. Application examples focus on network reconfiguration.

2.4 Contract-oriented frameworks

The control-oriented frameworks focus on the monitoring and control of components through the sensing and manipulation of control variables. Contract-based frameworks, on the other hand, control the system through the (dynamic) specification of the relationships which components must follow. There are differences in the types of contract between frameworks. Some contracts define the valid configuration(s) in the composites. Other contract-based frameworks view contracts as exercising control by constraining the interactions between components. In this sense, contracts are both structure-centric and quality-centric descriptions, as discussed (Fig. 3). Contracts have the ability to define the existence of relationships (hence structure), as well as the quality of those relationships. Some frameworks, such as ConFract [20] and the framework proposed in this paper, have contracts that perform both these functions. As illustrated in Fig. 3, Colman [1] believes there are two methods of defining and controlling the quality of relationships. The first method is to control the interface of the component involved in any association so that only behaviour acceptable to the contract can occur over that interface. This approach characterises the non-functional properties of the component interface irrespective of its actual relationships.

The second method focuses on characterising and controlling the connectors rather than characterising the components. This is the approach adopted in this paper. Non-functional relationships can always be reduced to an abstraction over functional relationships. Although we tend to think of a non-functional requirement as a property of an entity (role, object, component etc.), it is always a requirement in relation to some other entity (or entities). In terms of a contract, a non-functional property of a relationship has both a requirement (obligation) and a state-of-fulfilment of that obligation (performance). Non-functional properties can be viewed as abstractions across functional interactions, even though many such properties (e.g. availability, fault tolerance) may be invariant for all of a component's relationships.

In this area, a great deal of research has been done. Camara *et al*. [21] use symbolic transition system (STS) notation for developing adaptation contracts. STS provides protocol of a component or services on its interface to verify and specify the proposed adaptation contract. Also Salaün and coworkers [22] define a behavioural model inspired by the π-calculus to specify behavioural interfaces of components. They also choose to specify behavioural interfaces using labelled transition systems (LTSs) [2] as notation. This is very convenient for developing composition algorithms since they rely on the traversal of the different states of the components, and the transition function of LTS descriptions makes the access to the set of states and their connections straightforward. However, the other part of this framework is dynamic reconfiguration. After proper connections between components are established using contracts, reconfiguration abilities are added to the framework too. Similar to the contract specifications, there is a wide range of techniques that can support reconfiguration.

Huang *et al.* [23] facilitate the development of the dynamically partially reconfigurable system (DPRS), with a model-based platform-specific co-design methodology for DPRS. For DPRS analysis and validation, a model-based verification and estimation framework is proposed to make model-driven architecture more realistic and applicable to the DPRS design. In the system implementation phase, the OS4RS design is integrated into the DPRS design methodologies [24]. Thus, reconfigurable hardware functions are executed as software applications that can be dynamically created and removed. Cansado *et al*. [14] use LTS for their formal model that includes behavioural adaptation. Their framework provides structural reconfiguration of components as well. In this paper, we try to improve Cansado's work by using RTSs for the system and contracts specification. We use synchronisation vectors to composite 'Adaptors' in a hierarchical manner and achieve reconfiguration patterns.

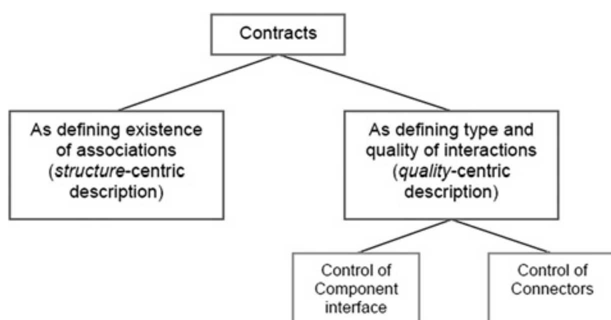

Fig. 3 *Aspects of contracts and methods of controlling interactions [1]*

2.5 Reactive transition system

A good idea to make use of contracts as connectors is to have an approach that includes: (i) a formal and sound language, (ii) a graphical notation to define transitions and (iii) composition and hierarchical techniques to facilitate interactions via several components and composites. In this paper, reactive transition system (RTS) is selected to achieve the above properties.

Definition 2.1: A RTS is defined as $L = (s^0, S, \Sigma, \Delta)$, where

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

© The Institution of Engineering and Technology 2013

RTSs are like LTSs. The main distinction is that RTSs have input, output and internal events. This distinction helps the system decide when to produce an output, whereas the environment decides when to provide an input.

The composition of RTSs is defined in terms of synchronisation vectors. Synchronisation vectors were introduced by Arnold and Nivat (Arnold 1994) [2]. In this way, not only peer-to-peer but also multicast and broadcast communication among processes can be described.

Definition 2.2: Consider sets E_0, E_1, …, E_n, a relation $\boldsymbol{R} \subseteq \prod_{0\le i\le n} E_i$, and a set $E \subseteq \bigcup_{0\le i\le n} E_i$ such that $E \cap E_i \cap E_j \cap \emptyset$ for all i, j: $0 \le i$, $j \le n \wedge i = j$. Let projections π_i: $\boldsymbol{R} \to E_i$ for $0 \le i \le n$ and sets of keys $\kappa_{\boldsymbol{r}} = \{\pi_i(\boldsymbol{r}) \in E / 0 \le i \le n\}$ for $\boldsymbol{r} \in \boldsymbol{R}$. Then $\boldsymbol{R}$ is said to be indexed by E if there exists:

- Exactly one key per tuple: $|\kappa_{\boldsymbol{r}}| = 1$ for all $\boldsymbol{r} \in \boldsymbol{R}$;
- At most one occurrence per key: $\forall e \in E$, $\exists \boldsymbol{r} \in \boldsymbol{R}$, $e \in \kappa_{\boldsymbol{r}}$ implies $\forall r' \in R \ \{r\},\ e \notin \kappa_{r'}$.

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

Definition 2.3: A RTS 'network' is a tuple $N = (\Sigma, W, \boldsymbol{R})$, where Σ is a set of external events of the network, consisting of two disjoint sets of input events $\Sigma^{\rm I}$ and output events $\Sigma^{\rm O}$. We let $\Sigma^{\#} = \Sigma \cup \{\epsilon\}$ that ε is not an event of any RTSs.

- W is a finite set of RTSs. We let $\Sigma_l^{\#} = \Sigma_l^{\rm obv} \cup \{\epsilon\}$ for all $l \in W$;
- $\boldsymbol{R} \subseteq \Sigma^{\#} \times \prod_{l\in W} \Sigma_l^{\#}$ is a relation indexed by $\Sigma^{\rm I} \bigcup_{l\in W} \Sigma_l^{\rm O}$.

The relation $\boldsymbol{R}$ is called a set of synchronisation vectors. A synchronisation vector in the set describes a particular synchronisation pattern between the component RTSs. In order to give RTS networks a sound interleaving semantics, we require exactly one output event in each synchronisation vector, because output is non-blocking in asynchronous applications. We also require that at most one synchronisation vector can be matched for a particular output event. ($\Sigma^{\rm obv} = \Sigma^{\rm O} \cup \Sigma^{\rm I}$)

Definition 2.4: Consider a RTS network $N = (\Sigma_N, W, \boldsymbol{R})$. The synchronised product of N is a RTS $L = (s^0, S, \Sigma, \Delta)$, where

- $s^0 = \prod_{l\in W} s_l^0$ and $S \subseteq \prod_{l\in W} S_l$ is the smallest set such that $s^0 \in S$ and $\forall s \in S$, $(s, e, S') \in \Delta$ implies $S' \in S$. We assume projection π_l: $S \to S_l$ and let $s_l = \pi_l(s)$ and $s_l' = \pi_l(s')$ for $l \in W$, s, $s' \in S$;
- $\Sigma_L^{\rm I} = \Sigma_N^{\rm I}$, $\Sigma_L^{\rm O} = \Sigma_N^{\rm O}$ and $\sum_L^H \subseteq \prod_{l\in W} \Sigma_l^{\rm ctrl}$;
- Δ consist of input steps $\left\{(s,\ e,\ s')\ |e \in \Sigma_L^{\rm I},\ \exists \boldsymbol{r} \in \boldsymbol{R},\ \rho_{\boldsymbol{r}} = {\rm env}\ \wedge e = \pi_{\rm env}(\boldsymbol{r}) \wedge \delta(s,\ \boldsymbol{r},\ s')\right\} \cup \left\{(s,\ e,\ s)\ |e \in \Sigma_L^{\rm I},\ \nexists \boldsymbol{r} \in \boldsymbol{R},\ \rho_{\boldsymbol{r}} = {\rm env}\right\}$,

Output steps $\{(s,\ e,\ s')\ |e \in \Sigma_l^{\rm O},\ \exists\, l \in W,\ \boldsymbol{r} \in \boldsymbol{R},\ 1 = \rho_{\boldsymbol{r}} \wedge e = \pi_{\rm env}(\boldsymbol{r}) \wedge (s_l,\ \pi_l(\boldsymbol{r}),\ s_l') \in \Delta_l \wedge \delta(s,\ \boldsymbol{r},\ s')\}$,

And internal steps $\{(s,\ e,\ s')\ |\exists\, l \in W,\ e \in \Sigma_l^H \wedge (s_l,\ e,\ s_l') \in \Delta_l \wedge (\forall g \in W\backslash\{l\},\ s_g' = s_g)\} \cup \{(s,\ e,\ s')| \exists\, l \in W,\ e \in \Sigma_L^{\rm O} \wedge \exists r \in R,\ e = \pi_l(\boldsymbol{r})\ \wedge \pi_{\rm env}(\boldsymbol{r}) = \epsilon \wedge (s_l,\ e, s_l') \in \Delta_l \wedge \delta(s,\ \boldsymbol{r},\ s')\} \cup \left\{(s,\ e,\ s')\ |\exists\, l \in W,\ e \in \Sigma_l^{\rm O} \wedge (\nexists \boldsymbol{r} \in R,\ e = \pi_l(\boldsymbol{r})) \wedge (s_l,\ e,\ s_l') \in \Delta_l \wedge (\forall g \in W\backslash\{l\},\ s_g' = s_g\ \right\}$, $\delta(s,\ \boldsymbol{r},\ s') = (\forall g \in \iota_{\boldsymbol{r}},\ (s_g, \pi_g(\boldsymbol{r}),\ s_g') \in \Delta_g) \wedge (\forall h \in \iota_{\boldsymbol{r}},\ s_h' = s_h)$.

Note that, for a RTS network N and a synchronisation vector $\boldsymbol{r} \in \boldsymbol{R}$ and projection π_l: $\boldsymbol{R} \to \Sigma_l$, $\rho_{\boldsymbol{r}}$ is called producer of $\boldsymbol{r}$ and it is a unique RTS l such that $\pi_l(\boldsymbol{r}) \in \Sigma_l^{\rm O}$. Also the set of consumers of $\boldsymbol{r}$, denoted by $\eta_{\boldsymbol{r}}$, consists of all RTSs l such that $\pi_l(\boldsymbol{r}) \in \Sigma_l^{\rm I}$. In addition, an idler of $\boldsymbol{r}$ is a RTS l such that $\pi_l(\boldsymbol{r}) = \epsilon$ and denoted by $\iota_{\boldsymbol{r}}$.

Jin *et al.* [12] provided general definition for components by specialising reactive systems (RTSs). They described how to provide services of a component and how to consider environment assumptions of a component by means of the component protocols. Isazadeh and Karimpour [13] extended [12] to describe the system as a hierarchical composition of some components and proposed a compositional verification method to verify component-based systems.

3 New framework

In this section, we are going to extend Cansado *et al.*'s work [14] using Jin's work in his PhD [12], and a method proposed by Isazadeh and Karimpour [13] to present our framework. For this purpose, at first the association of components, as the infrastructure of the framework, will be described by contracts. Then, these associations will be connected to each other in a larger part, called 'Adaptor'. 'Adaptors' will specify the configurations of the system. Synchronisation vectors are exploited to composite and reconfigure 'Adaptors' in a network of 'Adaptors'.

3.1 Adaptation contract

Our work is based on the concept of contracts which associate two components. These components could have different behavioural interfaces or could be components from different systems or parts, for example, one from server and the other from client part. Contracts also regulate and help to monitor interactions between components. They define which interactions are permissible or required and set arbitrary performance conditions on the interactions of components and monitor these interactions for compliance with those conditions. In this paper, contracts build the structure of 'Adaptors'.

'Adaptors' work based on adaptation contracts and can automatically fit two different interfaces with each other. Adaptation contract $\mathcal{AC}$ will be used between components, based on vectors. A vector may involve any number of components and does not require interactions to occur with the same kind of interfaces. Moreover, a vector may synchronise events performed by sub-processes in a hierarchical style. With having contracts in vectors, they can be used to enforce sequences of interactions. The mathematical formalism of adaptation contracts that suite our work is presented as below.

Definition 3.1: Adaptation contract $\mathcal{AC}$ is a set of vectors $\boldsymbol{V}$ = [part 1: event (D), part 2: event (D)] that:

© The Institution of Engineering and Technology 2013

doi: 10.1049/iet-sen.2012.0163

- Parts 1 and 2 are events in the first and second system parts;
- $D \in (!, ?)$ is a tuple that stands for the communication direction. (! For emission and? for reception).

Each event in a vector is executed by exactly one component and the overall result has impacts on other components. Actions occur when exactly both events in parts 1 and 2 happen in the same time. This formalism also can be used in reusing components. First part can be interface of reusable components and the second part can be acceptable interface of the system.

3.2 Work of adaptation contract

We now establish a framework for reconfiguration and behavioural adaptation. At first an example is presented to show how adaptation contract can work between client and server then we define an 'Adaptor' and we work with it so that can easily handle mismatches, and after that we provide a formal model based on nets for this kind of systems.

We present a client/server system, in which the server may be substituted by an alternative server component. This example represents an online shopping center that has two parts, one for entertainment and another for home shopping. Owners want to have two servers that works separately in each part. The client wants to buy CD/DVD, Sport tools, books, clothes, foods and electronic tools as shown in its behavioural interface in Fig. 4*a*. The two servers E and H have behavioural interfaces depicted in Figs. 4*c* and 5*c*, respectively.

Initially, the client is connected to server E, we shall call this configuration c_E. The client and the server agree on an adaptation contract $\mathcal{AC}_{C,E}$ (see Fig. 4*b*). Under configuration c_E the client cannot buy food because it is not supported by server E. The latter is implicitly defined in the adaptation contract (Fig. 4*b*), because there is no vector allowing the client to perform the action food!.

In an alternative configuration c_H the client is connected to server H whose protocol is depicted in Fig. 5*c*. Similarly, the client and the server agree on an adaptation contract $\mathcal{AC}_{C,H}$ (see Fig. 5*b*). Under configuration c_H, the client can buy food, clothes and electronic tools.

When the client is in server E and wants to buy something from server H, he/she should sign out from first server and enter his\her identity again into the second one. We want to reconfigure c_E to c_H in order that the client can buy anything without knowing about our two different servers. It means, we just substitute E by H in a way that client does not know about this reconfiguration. We shall study reconfiguration from c_E to c_H which substitutes E by H. It is worth noting that E and H do not have the same behavioural interfaces and thus we should connect client with new server that have no mismatches with former transaction.

3.3 Adaptor

After we modelled our system with RTS, we can substitute it with 'Adaptor' which contains both system and adaptation contract. In this way we can easily work on networks in hierarchical manner.

Definition 3.2: Let P be our system that contains two different parts with their adaptation contract $\mathcal{AC}$ between them. To build an 'Adaptor' we do as follows:

- For each event $a! \in \Sigma^{O}_{P_i}$, we define a step in Adaptor as $v = \{P_i: a!, A_p: a?\}$, that means if in state P_i a reception to our system occurs, in Adaptor that state change to emission.
- For each event $a? \in \Sigma^{I}_{P_i}$, we define a step in Adaptor as $v = \{P_i: a?, A_p: a!\}$, that means if in state P_i an emission to our system occurs, in Adaptor that state change to reception.
- We add an event $\boldsymbol{r}_{S_i^*}?$ that can substitute one Adaptor with another.

It means that with this definition client and server become a concrete system called 'Adaptor'. For this we start with client's first request from server and then we add the answer of server to this request in 'Adaptor', again we assume the clients requests and servers answers to those request and according to this cycle we built the 'Adaptor'. The main point here is that we change all the emissions and receptions of client and server in 'Adaptor'.

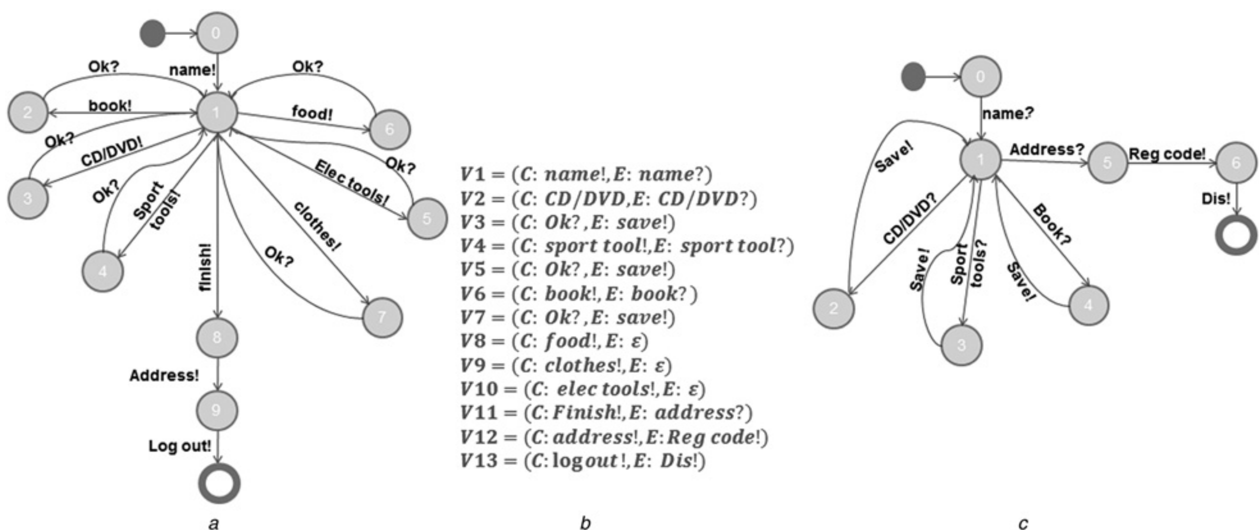

Fig. 4 *Client and server E RTS model and their configurations*
a RTS of client C
b Adaptation contract $\mathcal{AC}_{C,E}$
c RTS of server E

© The Institution of Engineering and Technology 2013

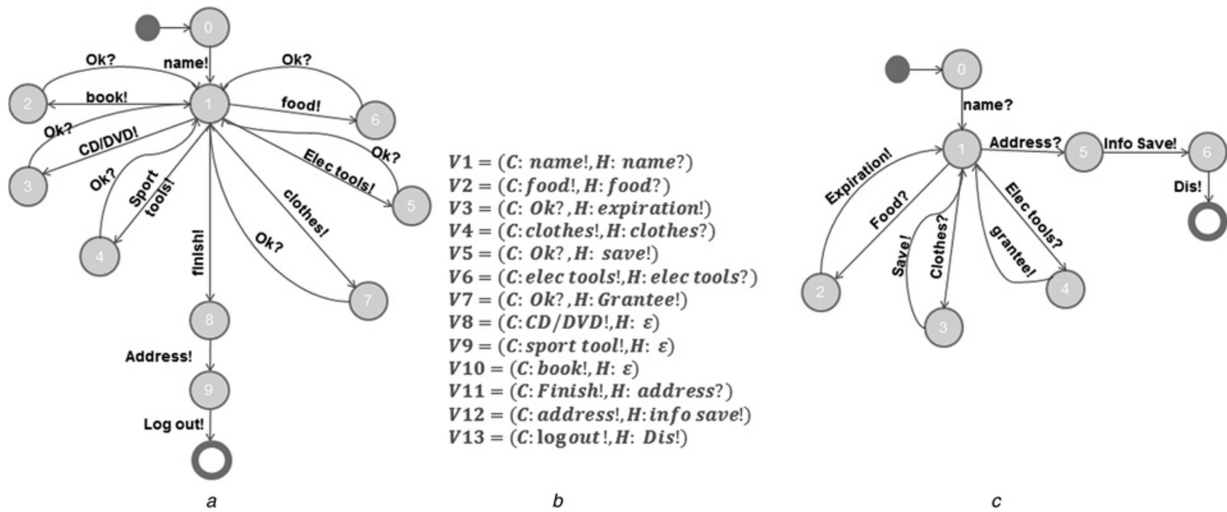

Fig. 5 *RTS model of client C and server H and their configurations*

a RTS of client *C*
b Adaptation contract $\mathcal{AC}_{C,H}$
c RTS of server *H*

In most of other approaches, 'Adaptors' are as third-party components that are in charge of coordinating the parts involved in the system. Those systems have three parts. In our approach, for decreasing the overall cost of monitoring, memory and complexity, we use just one part, that is, 'Adaptor'. Also, the scalability of the system increases because adding and removing some parts of the system can be easily done by just adding and removing producer vectors of 'Adaptor'. Adding posterior adaptors into a network can be done to improve the scalability of the framework too. It is like adding another vector to the synchronisation vectors that have mathematical base. These vectors described more details later.

Now, we build 'Adaptors' for the pervious example. In that example, there were two pairs of server and client with adaptation contract between them. Each vector in adaptation contracts generates two events in 'Adaptor'. For example, vector $\boldsymbol{V}1 = (C\text{: name!}, E\text{: name?})$ is divided into two states in 'Adaptor' $\mathcal{AC}_{C,E}$, that is, *C*: name! and *E*: name?. Event $\boldsymbol{r}_{S_i^*}?$ can be in any state that system engineer prefers (Fig. 6).

After creating 'Adaptor', the server and the client can work consistently with each other. Sometimes servers are huge and can be divided into some parts that are not similar to each other. This may make working with and finding errors in the system easier. In this paper, we assume servers are made from different parts that are irrelative to each other.

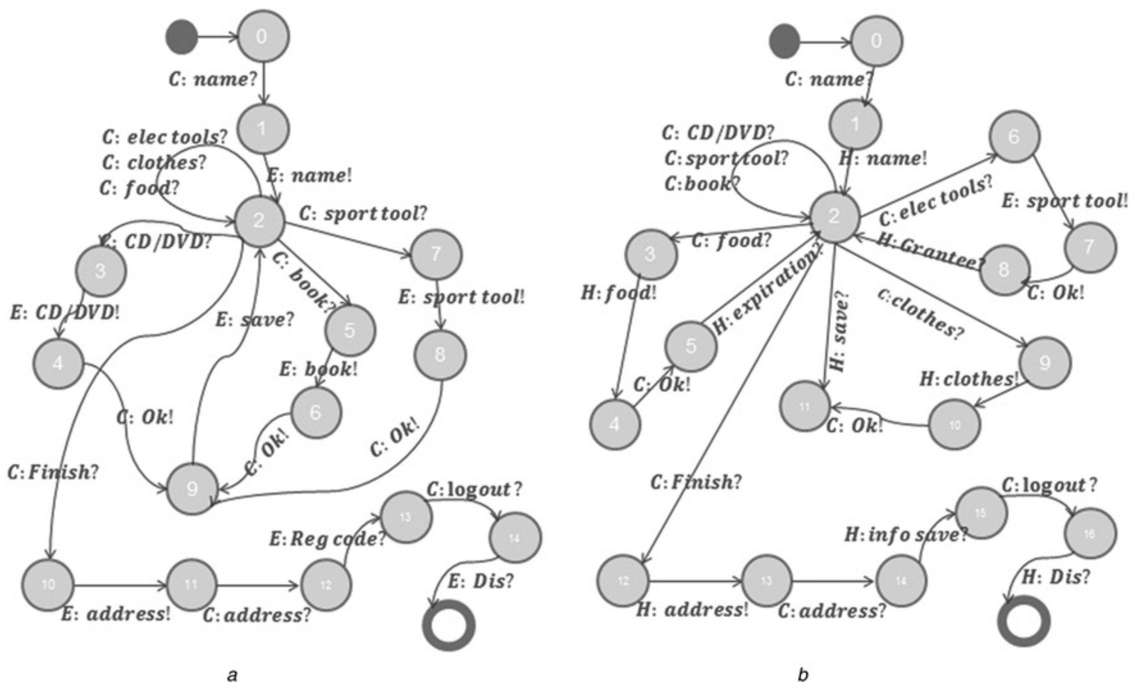

Fig. 6 *Adaptors between client and servers*

a Adaptor $A_{C,E}$
b Adaptor $A_{C,H}$

© The Institution of Engineering and Technology 2013
IET Softw., 2013, Vol. 7, Iss. 5, pp. 258–270
doi: 10.1049/iet-sen.2012.0163

3.4 Reconfiguration

Reconfiguration is to substitute one part of system with another (here, system parts are 'Adaptors'). This substitution starts from one state in the first part and ends in another state in the target part. We denote these transitions with $\boldsymbol{r}^*$. The important key is to select reconfiguration states in a way that 'Adaptor' substitution is hidden from client. We assume designers of the system select those states.

Definition 3.3: A reconfiguration contract $\mathcal{R} = (A_p, a_{p_0}, \Delta_{\mathcal{R}})$ is defined as: A_p is the set of 'Adaptors' and a_{p_0} is the initial configuration. $\Delta_{\mathcal{R}} \subseteq A_p \times R_{\mathrm{O}} \times A_p$ is the set of reconfiguration operations such that $\boldsymbol{R}_{\mathrm{O}} \subseteq S_i{*} \times S_j{*}$, where $S_i^* \subseteq A_{p_i}$ and $S_j^* \subseteq A_{p_j}$. S_i^* is the state that reconfiguration starts and $S_j{*}$ is the state which the new part starts working from.
Selecting the states that reconfiguration starts and ends (S_i^*, S_j^*) is out of scope of this paper. It is notable that these states are selected in design time. To reach the target state in the second part of the reconfiguration, traces can be used.

Definition 3.4: Let $L = (s^0, S, \Sigma, \Delta)$ be a RTS. Trace ξ consists of a set of events such that $\xi = \{\sigma | \sigma = e.\sigma'; e \in \Sigma \wedge (s^0, e, s) \in \Delta, \sigma' \in \xi'\}$ where ξ' is a trace of $L' = (s, S, \Sigma, \Delta)$.

With this definition we can introduce ξ_s, that is, a trace of any particular RTS from its initial state till the special state$s$. To make these definitions clear, an example is given. Assume the 'Adaptors' in the previous example. Since the 'Adaptor' $\mathcal{AC}_{C,E}$ does not have permission to buy food, a reconfiguration step is added. In the 'Adaptor' $\mathcal{AC}_{C,H}$ a step, from which the new configuration will start, should be selected. In this example, we have: $\mathcal{R} = (A_p, a_{p_0}, \Delta_{\mathcal{R}})$ where $A_p = \{\mathcal{AC}_{C,E}, \mathcal{AC}_{C,H}\}$ and $a_{p_0} = \mathcal{AC}_{C,E}$ and $\Delta_{\mathcal{R}} \subseteq \mathcal{AC}_{C,E} \times R_{\mathrm{O}} \times \mathcal{AC}_{C,H}$ such that $\boldsymbol{R}_{\mathrm{O}} \subseteq s_1 \times s_1$. It means that when the configuration reaches to state 1 in the first 'Adaptor', the system can reconfigure to the 'Adaptor' $\mathcal{AC}_{C,H}$ with reconfiguration step $\boldsymbol{r}^*$ and then start the new configuration from state 1 of 'Adaptor' $\mathcal{AC}_{C,H}$. The system reaches to state 1 in the second 'Adaptor' with ξ_1.

3.5 Network of adaptors

Real scenarios for adaptation often involve several interacting parts or services that are called 'Adaptors' in this paper. With increasing the number of 'Adaptors', the complexity problem starts and hinders the developing task. We solve this problem using 'Adaptors' in a hierarchical network. If interactions can be encapsulated in different 'Adaptors', the developers can focus on the particular adaptation for any sub-problem. This encapsulation has important advantages in design, development and debugging time. Expanding, testing and replacing 'Adaptors' in required time can be possible as well. Networks are built by composition of 'Adaptors'. This composition is based on synchronisation vectors that orchestrate the whole network with different synchronisation patterns called relations.

This section represents how a network of 'Adaptors' can be built. We previously defined the system with RTS and modelled it. Now we generate a complete system that can be fed into model-checking tools. We also show how two 'Adaptors' can be exchanged in runtime, in a way that system does not need any shutting down for this reconfiguration.

Definition 3.5: A tuple as $\mathcal{A}_{\mathrm{Net}} = (\Sigma, W, \boldsymbol{R})$ defines a network of 'Adaptors' such that:

- $\Sigma = \Sigma^{\mathrm{I}} \cup \Sigma^{\mathrm{O}} \cup \Sigma^{H}$.
- W is a set of 'Adaptors' 'that are in RTS form. For all $A_p \in W$, $\Sigma^{\#}_{A_p} = \Sigma^{\mathrm{obv}}_{A_p} \cup \{\varepsilon\}$.
- $\boldsymbol{R} \subseteq \left\{\Sigma^{\#} \times \prod \Sigma^{\#}_{A_p}\right\} \cup \{\boldsymbol{r}^*\}$ is indexed by $\Sigma^{\mathrm{I}} \cup \bigcup \Sigma^{\mathrm{O}}_{A_p}$.

In the definition above, Σ shows all the events that affect the system, they can be an input event from the environment or input event from internal 'Adaptors' $\left(\Sigma^{\mathrm{I}} = \Sigma^{\mathrm{I}} \cup \Sigma^{\mathrm{I}}_{\mathrm{env}}\right)$ or maybe an output event from other 'Adaptors' (Σ^{O}) or internal event in any 'Adaptor' (Σ^{H}) (Fig. 7).

$\boldsymbol{R}$ is a synchronisation vector that describes a particular synchronisation pattern between the two 'Adaptors'. The first part, $\left\{\Sigma^{\#} \times \prod \Sigma^{\#}_{A_p}\right\}$, shows the vector can include configurations from current 'Adaptor'. For example, when the client connects to the network for the first time, this vector selects the proper 'Adaptor' that should work with the client or when several clients work with the network in parallel, this vector synchronises the overall network at each transaction. When an 'Adaptor' needs to reconfigure to

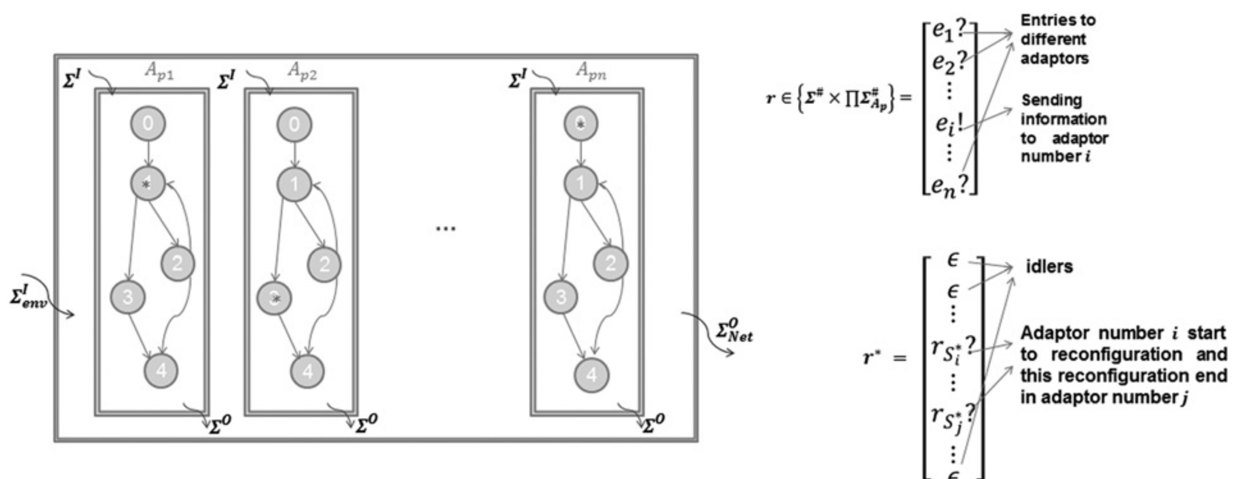

Fig. 7 *Network of Adaptors*

© The Institution of Engineering and Technology 2013

another, builds the synchronisation vector $\{\boldsymbol{r}^*\}$. With this event, the system knows which part requested for reconfiguration and also the target 'Adaptor' in this reconfiguration.

For making sure that any changes in the network affect the 'Adaptors', we present the following theorem.

Theorem 3.6: Consider a RTS network $N = (\Sigma_N, W, \boldsymbol{R})$ and $L_N = (s_N^0, S_N, \Sigma_{L_N}, \Delta_N)$ be the synchronisation product of N, and ξ be a trace of any RTS ℓ from s_ℓ^0. We prove any event $e \in \Sigma_N$ leads RTS components to take a step. This step could be configuration (internal in RTSs) or reconfiguration (causes change in RTSs).

Proof: We prove this theorem by induction on set of events and steps.

1. If $e \in \Sigma_N^{\mathrm{I}}$ then $\forall \ell \in W$, if $\exists s^0 \in S \wedge \pi_\ell(s^0) = e$, if $(s^0, e, s') \in \Delta_\ell \Rightarrow (s^0, e, s') \in \Delta_N$. That means if e be an input event of the network and this event causes any RTS $\ell \in W$ to take a step, then this step is a step of synchronisation product of the network.
2. If $\exists e_1, e_2, \ldots, e_{m-1} \in \Sigma_N^H$, if $e_m \in \Sigma_N^H$ then $\forall \ell \in W$, $\pi_\ell(s) = e$ and $(s, e, s') \in \Delta_\ell \Rightarrow (s, e, s') \in \Delta_N$. That means if we have a set of internal events of the network and if the next event is also an internal event, then this event causes the responsible RTS to take a step and this step is considered as a step of synchronisation product of the network.
3. If $\exists e_1, e_2, \ldots, e_{m-1} \in \Sigma_N^H$, if $e_m \in \Sigma_N^{\mathrm{ctrl}}$, if $\exists \ell \in W \wedge s_i^* \in S_\ell$ then $\exists \boldsymbol{r}^* \in R$, $\rho_{\boldsymbol{r}^*} = \ell \wedge e = \pi_{\rho_{\boldsymbol{r}^*}}(\boldsymbol{r}^*)$ and $\exists \ell' \in W, \eta_{\boldsymbol{r}^*} = \ell' \wedge s_j^* \in S_{\ell'} \Rightarrow (s_i^*, e, s_j^*) \in \Delta_N$.

That means if we have a set of internal events of the network and if the next event is from controllable event of L_N, and if we have a RTS $\ell \in W$, in which we have a reconfiguration step s_i^*, from which ℓ wants to start reconfiguration, then ℓ creates a synchronisation vector $\boldsymbol{r}^*$ that sends event e to its consumer ℓ'. After reconfiguration, ℓ' starts to work from state s_j^*. We note that trace ξ_s can be used to reach this state in ℓ'.
4. If $\exists e_1, e_2, \ldots, e_{m-1} \in \Sigma_N^H$, if $e_m \in \Sigma_N^{\mathrm{O}}$ then $\exists \ell \in W$, $\exists \boldsymbol{r} \in \boldsymbol{R}$, $\rho_r = \ell \wedge e = \pi_{\mathrm{env}}(\boldsymbol{r}) \Rightarrow (s, e, \mathrm{env}) \in \Delta_N$.

That means if we have a set of internal events in the network and the next event is an output event of the network, then we have a RTS $\ell \in W$ and a synchronisation vector $\boldsymbol{r}$ that ℓ produces. This vector leads the network to have an output to the environment. As a result $(s, e, \mathrm{env}) \in \Delta_N$.
5. If $\exists e$, such that $e \notin \Sigma_N$ then $\forall \ell \in W$, if ℓ is in state s then $\pi_\ell(s) = \varepsilon$.

That means if an event is not a part of the network, then it is idler for the network and causes no effect. □

Theorem 3.6 indicates that any event in a network has a response that affects the overall RTSs. In that case we can ensure that in a network with proper synchronisation vectors, the system meets its goals. Lemma 3.7 will show a detailed version of Theorem 3.6 that uses trace projection to represent what happened in a specific RTS composite in a synchronised network.

Lemma 3.7: Consider a RTS Network $N = (\Sigma, W, \boldsymbol{R})$ and a RTS $\ell \in W$. Let L_N be the synchronisation product of N and $\boldsymbol{r}^*$ be a synchronisation vector for reconfiguration that maps an output event of any RTS ℓ', which is other than ℓ, into the corresponding input event of ℓ. RTS ℓ' causes the RTS ℓ to take an input step. Let $i, j \in$ reconfiguration states. The trace projection of σ in the network with respect to ℓ is defined as below:

1. $\pi_\ell(\sigma_n) = \pi_\ell(\sigma_{n-1})$, if $e \in \lambda\pi_\ell$
2. $(\sigma_n) = e.\pi_\ell(\sigma_{n-1})$, if $e \in \Sigma_\ell^H$
3. $\pi_\ell(\sigma_1) = e.\pi_\ell(\sigma_i)$, if $e \in \Sigma_{\ell^*}^{\mathrm{O}}$ and, $\exists \boldsymbol{r}^*, e = \pi(\boldsymbol{r}^*)$ and $\rho_{\boldsymbol{r}^*} = \ell'$, $\eta_{\boldsymbol{r}^*} = \ell$
4. $\pi_\ell(\sigma_n) = e.\pi_\ell(\sigma_{n-1}) = \pi_{\ell'}(\sigma_j)$, if $e \in \Sigma_\ell^{\mathrm{O}}$ and, $\exists \boldsymbol{r}^*$, $e = \pi(\boldsymbol{r}^*)$ and $\eta_{\boldsymbol{r}^*} = \ell'$
5. $\pi_\ell(\sigma_n) = \pi_\ell(\sigma_{n-1})$, if $\in \Sigma_\ell^{\mathrm{O}}$ and, $\nexists \boldsymbol{r}^*, e = \pi_{\rho_{\boldsymbol{r}^*}}(\boldsymbol{r}^*)$

The trace projection $\pi_\ell(\sigma_n)$ on ℓ is an event sequence consisting of events that ℓ takes while the network N follows the trace ℓ from its initial state s_N^0. It is computed recursively by considering each event e in σ sequentially. If e is an internal event of ℓ then it is appended to the trace; if e is an output event of any RTS ℓ', other than ℓ, with a synchronisation vector $\boldsymbol{r}^*$, and ℓ is the consumer of this vector then the trace σ continues its sequence in state i of RTS ℓ; if e is an output event of ℓ and ℓ should take a reconfiguration with respect to synchronisation vector $\boldsymbol{r}^*$, then the trace σ follow its sequence from state j of consumer of $\boldsymbol{r}^*$; if e is an output event of ℓ and there is no reconfiguration, then e is an output event of the network; otherwise e is ignored.

4 Case study

In this section, we discuss a case study of 'self-adaptive behaviours in a decentralised system'. Iftikhar and Weyns [25] presented a case study in which they used model checking to verify behavioural properties of a decentralised self-adaptive system. They formalised architectural model of a decentralised traffic monitoring system and modelled it with timed automata. The rational for choosing this case study is that this system is able to adapt autonomously to internal dynamics and changing conditions in the environment. With our framework, this traffic monitoring model can have dynamic reconfigurations and self-compositions to achieve particular quality goals.

4.1 System specification

Traffic monitoring system provides information about traffic jams. The main challenges of the system are: (i) inform clients of dynamic changing traffic jams, (ii) realise this functionality in a decentralised way and (iii) makes the system robust to camera failures. The system consists of a set of intelligent cameras which are distributed along the road. An example of a highway is shown in Fig. 8.

The system components are:

1. Cars, that are in environment and move along the subsequent viewing ranges of the cameras;
2. Cameras, which have three basic states. In normal operation, the camera can be master with no slaves, master of an organisation with slaves or it can be slave; and
3. Traffic monitor, that keeps track of the actual traffic conditions based on the signals it receives from the cars and determines traffic congestion. For each camera, a traffic monitoring process instance is running all the time.

© The Institution of Engineering and Technology 2013

doi: 10.1049/iet-sen.2012.0163

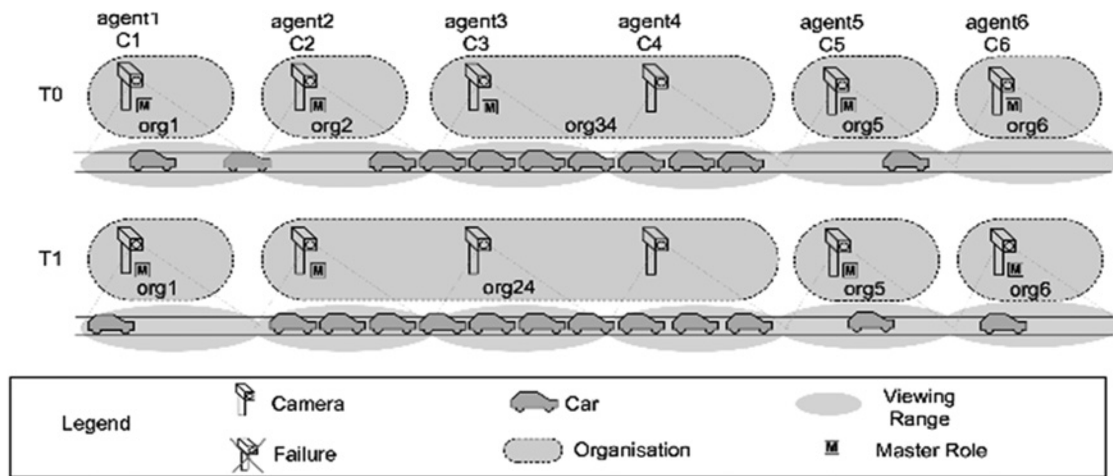

Fig. 8 *Traffic monitoring system [25]*

4.2 System operation

When the system is initialised, each camera belongs to a single member organisation. However, when a traffic jam is detected that spans the viewing range of multiple neighbouring cameras, organisations on these cameras will merge into one organisation. To simplify the management of organisations and interactions with clients, the organisations have a master/slave structure. The master is responsible for managing the dynamics of that organisation (merging and splitting) by synchronising with its slaves and neighbouring organisations and reporting traffic jams to clients. Therefore the master uses the context information provided by its slaves about their local monitored traffic conditions. At T0, the example in Fig. 8 shows four single member organisations, org1 with agent1, org2 with agent2 and similar for org5 and org6. Furthermore, there is one merged organisation, org34 with agent3 as master and agent4 as slave. At T1, the traffic jam spans the viewing range of cameras 2, 3 and 4. As a result, organisations org2 and org34 have merged to form org24 with agent2 as master. When the traffic jam resolves, the organisation is split dynamically.

4.3 System in our proposed framework

Iftikhar and Weyns use two scenarios that require adaptation. Here, we just focus on one of them that needs dynamic reconfiguration and composition. The scenario concerns the dynamic adaptation of an organisation from T0 to T1, where camera 2 joins the organisation of cameras 3 and 4, after it monitors a traffic jam. For this purpose, the template of cars, traffic monitor and cameras are needed. As shown in Fig. 9*a*, cars enter and leave the viewing range of cameras. Fig. 9*b* shows the traffic monitor in which whenever a car enters into the viewing range of a camera, the traffic monitor detects the car via the *carEnt* channel. Similarly, when a car goes out of the range of a camera, the traffic monitor detects this through the *carLeave* channel. The traffic monitor determines a traffic jam by comparing the total number of cars in its viewing range with the 'Capacity'. Fig. 9*c* also shows the process of the cameras. Each camera works with signals that receives from the cars and gives to the traffic monitor.

To have safe connections between the three components of the system, the adaptation contracts are added to the traffic system. Contracts can regulate the transitions between

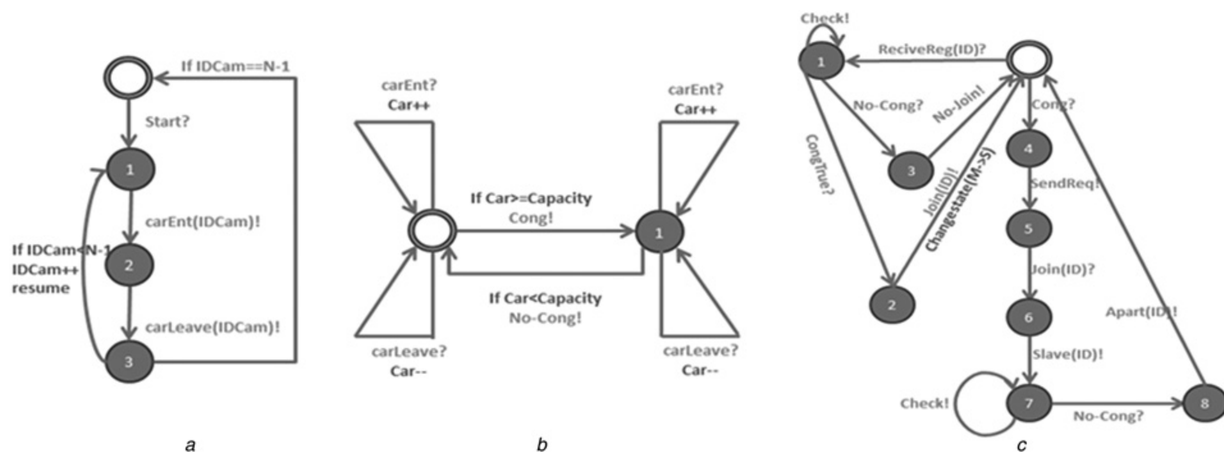

Fig. 9 *Templates*

a Car processes
b Traffic monitor
c Camera processes

© The Institution of Engineering and Technology 2013

Algorithm 1

inputs car processes, traffic monitor and camera processes, signals of help from other *Adaptor*s

outputs signals of help or information about traffic to other *Adaptor*s

1: set the first state M
2: observe cameras rang of view with vectors of traffic monitor
3: **if** $\exists\mathcal{V}_{TM} =: \langle \text{TM: Cong!} \rangle$ && the first state == M **then**
4: send signals to adaptor with ID n-1, n+1 and request help // by $r \in \Sigma^O$
5: **if** $\exists\mathcal{V}_{env} =: \langle \text{env: Join!} \rangle$ **then** // by $r \in \Sigma^I$
6: compose this adaptor to your own network and reconfigure to a Master with Slave organisation and set the first state to MWS
7: **end if**
8: **do** observe cameras range of view
9: **while** receive $\mathcal{V}_{TM} =: \langle \text{TM: No} - \text{Cong!} \rangle$
10: send signal Apart to slaves and return to configuration Master of single organisation and set the first state to M //by $r \in \Sigma^O$
11: **end if**
12: **goto** 1
13: **else if** $\exists\mathcal{V}_{env} =: \langle \text{env: Join!} \rangle$ **then** // by $r \in \Sigma^I$
14: **goto** 1
15: **end if**
16: **if** receive signal $\mathcal{V}_{env} =: \langle \text{env: ReceiveReq(ID)!} \rangle$ && the first state == M **then** // by $r \in \Sigma^I$
17: observe cameras rang of view with vectors of traffic monitor
18: **if** $\exists\mathcal{V}_{TM} =: \langle \text{TM: Cong!} \rangle$ **then**
19: send signal $\mathcal{V}_{cam} =: \langle \text{C: Join!} \rangle$ to adaptor (ID) and change the first state to S
20: **else if** $\exists\mathcal{V}_{TM} =: \langle \text{TM: No} - \text{Cong!} \rangle$ **then**
21: send signal $\mathcal{V}_{cam} =: \langle \text{C: No} - \text{Join!} \rangle$ to adaptor (ID) and **goto** 1
22: **end if**
23: **else if** receive signal $\mathcal{V}_{env} =: \langle \text{env: ReceiveReq(ID)!} \rangle$ && the first state == MWS **then** // by $r \in \Sigma^I$
24: **goto** 6
25: **end if**

Fig. 10 *Algorithm 1: Adaptor of traffic system*

components. The adaptation contracts between these three components are as below:

$$\text{AC}_{\text{CCME}} = \{\mathcal{V}_{\text{car,env}} =: \langle C\text{: start!}\ , E\text{: start?} \rangle$$

$$\mathcal{V}_{\text{cam,TM}} =: \langle C\text{: check!}\ , \text{TM: Cong!} \parallel \text{No} - \text{Cong!} \rangle$$

$$\mathcal{V}_{\text{TM, cam}} =: \langle \text{TM: Cong?}\ , C\text{: Cong!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: SendReq}(\text{ID}_{n-1,n+1})?, E\text{: ReciveReq}(\text{ID}_{n-1,n+1})! \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: Join(ID)?}, E\text{: Join(ID)!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: Slave(ID)!}, E\text{: Slave(ID)?} \rangle //\text{change state } \mathcal{V}_{\text{TM,cam}} =: \langle \text{TM: No} - \text{Cong?}, C\text{: No} - \text{Cong!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: Apart!}, \text{env: Apart?} \rangle //\text{change state}$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: } \varepsilon, E\text{: SendReq!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: ReciveReq(ID)!}, E\text{: } \varepsilon \rangle$$

$$\mathcal{V}_{\text{cam,TM}} =: \langle C\text{: check!}\ \ \text{TM: Cong!} \parallel \text{No} - \text{Cong!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: No} - \text{Join?}, E\text{: } \varepsilon \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{:Join?}\ , E\text{: } \varepsilon \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{: } \varepsilon, E\text{: Slave!} \rangle$$

$$\mathcal{V}_{\text{cam,env}} =: \langle C\text{:Slave?}, E\text{: } \varepsilon \rangle //\text{change state}$$

$$\}$$

The adaptation contract consists of car processes, camera, traffic monitor and environments. Here, 'environments' mean other 'Adaptors' that receive and send reconfiguration signals and information about changing the 'Adaptors' into single member organisation or organisations with master and slaves. These signals are sent with synchronisation vectors in the network. Algorithm 1 (see Fig. 10) shows the way 'Adaptors' work.

Fig. 11 shows the 'Adaptor' of the traffic system. Reconfiguration states are states 2, 7, 10 and 17, in which 'Adaptors' send and receive reconfiguration signals via synchronisation vectors. Theorem 3.6 assures safe signal transformation between several 'Adaptors' in networks.

Therefore, applying our framework to the common modelling example, shows that the formal constructs introduced in this paper are sufficient to specify the requirements of the common component modelling example. We were also able to reconfigure 'Adaptors', in which the execution is not interrupted and the whole system keeps running.

5 Conclusion and future work

This paper proposes a new framework for dynamic reconfiguration of adaptive systems. We also claim that the new framework is formal because we use a formal language, that is, RTS to model the framework.

This framework has some properties to assure its performance. They are listed as below:

1. *Reconfiguration possible at runtime*: The framework creates dynamic reconfigurations. These reconfigurations start from one 'Adaptor' and end in another. Since we define particular states for reconfigurations, our framework

© The Institution of Engineering and Technology 2013

doi: 10.1049/iet-sen.2012.0163

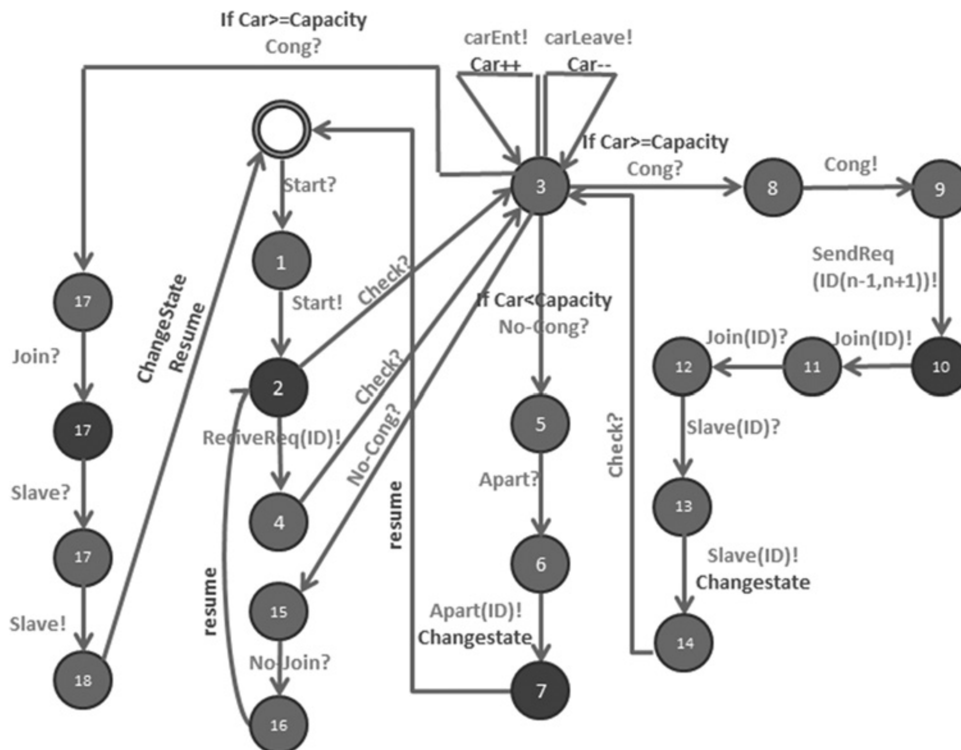

Fig. 11 *Adaptor of the traffic system*

supports runtime reconfiguration. In Theorem 3.6, part 3, we describe this reconfiguration in detail. However, we assume system designers select the reconfiguration states, but the system can use history states to reconfigure automatically, any time needed. The latter proposition is beyond the scope of this paper and can be considered as a future work.

2. *Elements can be substituted*: If a software system is viewed in a hierarchical manner, some parts of it can be seen as independent composites. We can substitute these composites by synchronisation vectors within the system network. Runtime substitution of one component with another one in lower level of composites is considered as a future work. We can also easily substitute components in earlier time of design by just adding or removing some vectors of reconfiguration contract. Modelling with reconfiguration contract and using a formal language guarantees the safety of these substitutions.

3. *Non-functional regulation possible*: we know non-functional properties can be reduced to functional ones. Since contracts are based on ‘Adaptors’, this framework takes account of non-functional properties too. We discussed about this property in Section 2.4.

4. *Formal reasoning about structure possible*: our framework is made of components that are connected to each other in a formal way. All the relations of the framework have mathematical descriptions. As a result, the structure can be formally represented.

5. *Management exogenous against endogenous*: The base of our framework is to use contracts and they just need components and their response to actions. Since contracts are between black box components, system management, which is based on contracts, imposes without needing access to the internal implementation of those components. Moreover, with extending the system network, the only responsible events would be just input and output ones. In this case the use of RTS model emboss itself because, RTS composites decide when to produce outputs while the environment gives them inputs.

6. *Supervisory control possible*: As defined in Section 3.7, system designers select reconfiguration states. These reconfigurations can substitute composites or servers. Since designers can easily add or remove components by just adding or removing adaptation contract vectors, then it is possible to have external management in situations that it needs an external control in additional of its original actions. It means that, it is possible for the system designers to add a reconfiguration that is not defined in earlier time.

We discussed some properties that the new framework supports. Also somehow this framework can make reusing components easier. On the other hand, we can improve software adaptation to repair or update systems with reconfiguring components and systems in urgent situations.

One can work on the traces of the system to provide reconfiguration in any state to have more flexibility. History states can be used to exchange ‘Adaptors’ whenever system needs or wants to upgrade itself. Since this framework use RTS meta-model and Jin introduces formal composition of RTSs in his theses, we could say maybe it is possible to work on formal composition of our framework.

6 References

1 Colman, A.: ‘Role-oriented adaptive design’. PhD thesis, Swinburne University, Melbourne, Australia, 2007
2 Arnold, A.: ‘Finite transition systems. Semantics of communicating systems’ (Prentice-Hall, 1994)
3 Simon, H.: ‘The sciences of the artificial’ (M.I.T. Press, Cambridge, 1969)
4 Beer, S.: ‘The viable system model’, *J. Oper. Res. Soc.*, 1984, **35**, (1), pp. 7–25
5 Tanenbaum, A.S., Steen, M.V.: ‘Distributed systems principles and paradigm’ (Prentice-Hall, 2002)

© The Institution of Engineering and Technology 2013

6 Autili, M., Inverardi, P., Navarra, A., Tivoli, M.: 'A tool for automatically assembling correct and distributed component-based systems'. Proc. Int. Conf. Software Engineering (ICSE'07), 2007, pp. 784–787
7 Brogi, A., Popescu, R.: 'Automated generation of BPEL adapters'. Proc. Int. Conf. Service-Oriented Computing (ICSOC'06), Springer, 2006 (*LNCS*, 294), pp. 27–39
8 MotahariNezhad, H.R., Benatallah, B., Martens, A., Curbera, F., Casati, F.: 'Semi-automated adaptation of service interactions'. Proc. World Wide Web Conf. (WWW'07), 2007, pp. 993–1002
9 Bracciali, A., Brogi, A., Canal, C.: 'A formal approach to component adaptation', *J. Syst. Softw.*, 2005, **74**, (1), pp. 45–54
10 Canal, C., Poizat, P., Salaün, G.: 'Model-based adaptation of behavioral mismatching components', *IEEE Trans. Softw. Eng.*, 2008, **34**, (4), pp. 546–563
11 Dumas, M., Spork, M., Wang, K.: 'Adapt or perish: algebra and visual notation for service interface adaptation'. Proc. Business Process Management (BPM'06), Springer, 2006 (*LNCS*, 4102), pp. 65–80
12 Jin, Y., Lakos, C., Esser, R.: 'Modular consistency analysis of component-based designs', *Proc. Res. Pract. Inf. Technol.*, 2009, **36**, (3), pp. 186–208
13 Isazadeh, A., Karimpour, J.: 'A new formalism for mathematical description and verification of component-based systems', *J. Supercomput.*, 2009, **49**, (3), pp. 334–353
14 Cansado, A., Canal, C., Salaün, G., Cubo, J.: 'A formal framework for structural reconfiguration of component under behavioral adaptation'. Proc. Electronic Notes in Theorical Computer Science, 2010, vol. 263, pp. 95–110
15 Shaw, M.: 'Software design paradigm based on process control', *ACM Softw. Eng. Notes*, 1995, **20**, (1), pp. 27–39
16 Bradbury, J.S.: 'Organization definition and formalisms of dynamic software architecture'. Technical report, Queen's University, 2004
17 Batista, T., Joolia, A., Coulson, G.: 'Managing dynamic reconfiguration in component-base systems'. Proc. European Workshop on Software Architecture Pisa, Italy, 2005
18 Khakpour, N., Jalili, S., Talcott, C., Sirjani, M., Mousavi, M.: 'Formal modeling of evolving self-adaptive systems', *J. Sci. Comput. Program.*, 2011, **78**, (1), pp. 3–26
19 Garlan, D., Cheng, S., Huang, A., Steenkiste, P.: 'Rainbow: architecture-based self-adaptation with reusable infrastructure', *Computer*, 2004, **10**, (37), pp. 46–54
20 Collet, P., Rousseau, R., Coupaye, T., Rivierre, N.: 'A constructing support for component-oriented programming'. SIGSOFT Symp. Component-based Software Engineering (CBSE'05), Springer Verlag, St-Louis, Missouri, USA, 2005 (*LNCS,* 3489)
21 Cámara, J., Martın, J.A., Salaün, G., Canal, C., Pimentel, E.: 'Semi-automatic specification of behavioural service adaptation contracts'. Proc. Electronic Notes in Theorical Computer Science, 2010, vol. 264, pp. 19–34
22 Cámara, J., Salaün, G., Canal, C.: 'Composition and run-time adaptation of mismatching behavioural interfaces', *J. Univers. Comput. Sci.*, 2008, **14**, (13), pp. 2182–2211
23 Huang, C.H., Hsiung, P.A., Shen, J.S.: 'Model-based platform-specific co-design methodology for dynamically partially reconfigurable systems with hardware virtualization and preemption', *J. Syst. Archit.*, 2010, **56**, pp. 545–560
24 Morandi, M., Novati, M., Santambrogio, D., Sciuto, D.: 'Core allocation and relocation management for a self dynamically reconfigurable architecture'. Proc. 2008 IEEE Computer Society Annual Symp. Very Large Scale Integration (VLSI), 2008, pp. 286–291
25 Iftikhar, M.U., Weyns, D.: 'A case study on formal verification of self-adaptive behaviors in a decentralized system'. 11th Int. Workshop on Foundations of Coordination Languages and Self Adaptation (FOCLASA'12), 2012, pp. 45–62

© The Institution of Engineering and Technology 2013

doi: 10.1049/iet-sen.2012.0163